Eric A. Wohlstadter
December 2004
Computer Science

Aspect-Oriented Development of Distributed Object Applications

Abstract

It it is often useful to distinguish between the definition of a core software system and features that may apply to it. A feature is an optional extension of the original system. Systems need to be *adapted* in order to take on new *features.* Distributed heterogeneous software is particularly hard to adapt for deployment in disparate execution environments. We introduce the concept of an adaptation *service* to coordinate crosscutting modifications to standard program components, transparently. This is achieved through Aspect-Oriented Programming. A service is made up of client- and server-side components called *adaptlets.* At run-time a protocol, *GlueQoS*, can be used to determine what adaptlets are activated. The goal is to reduce development costs associated with middleware application maintenance and deployment.

Professor Premkumar Devanbu
Dissertation Committee Chair

Aspect-Oriented Development of Distributed Object Applications

By

ERIC A. WOHLSTADTER
B.S. (University of California at Davis) 1998

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Committee in charge

2004

Aspect-Oriented Development of Distributed Object Applications

Contents

List of Figures

Abstract

It it is often useful to distinguish between the definition of a core software system and features that may apply to it. A feature is an optional extension of the original system. Systems need to be *adapted* in order to take on new *features*. Distributed heterogeneous software is particularly hard to adapt for deployment in disparate execution environments. We introduce the concept of an adaptation *service* to coordinate crosscutting modifications to standard program components, transparently. This is achieved through Aspect-Oriented Programming. A service is made up of client- and server-side components called *adaptlets*. At run-time a protocol, *GlueQoS*, can be used to determine what adaptlets are activated. The goal is to reduce development costs associated with middleware application maintenance and deployment.

Professor Premkumar Devanbu
Dissertation Committee Chair

Chapter 1

Introduction

Both business and military scenarios demand custom distributed applications to meet specific software requirements [1, 64]. Designers of these applications develop software to coordinate the activities of participating network hosts. Often business pressures force the need for cooperation between *heterogeneous* network hosts. Heterogeneity arises when software must be written in different programming languages, for different operating systems, or for different hardware architectures. Since languages, operating systems, and architectures all provide a *platform* for building applications we can view heterogeneity as arising from platform variability. Integration of legacy assets, contractual obligations, resource constraints, and performance requirements are also examples of pressures that cause heterogeneity. These distributed heterogeneous ($\mathcal{D}H$) systems are now embedded in the crucial infrastructure of our society. They are pervasive as a part of the telecommunications industry, financial sector, and military complex. A process of software development based on *middleware* is often used to ease the burden of building and maintaining these large $\mathcal{D}H$ systems.

Middleware The National Science Foundation of the U.S. defines middleware [61] as:

Middleware is software that connects two or more otherwise separate ap-

> plications across the Internet or local area networks. More specifically, the term refers to an evolving layer of services that resides between the network and more traditional applications for managing security, access and information exchange.

We see that the purpose of middleware is *connection* between applications in a distributed setting. This is a challenge because we need to connect applications built on heterogeneous platforms. So, middleware must translate between platform specific details to common standards of communication.

Software developers need only be concerned with developing the separate application software without regard for how an application will be integrated into a larger $\mathcal{D}H$ system. Viewed in this light, building systems becomes a matter of developing or purchasing application pieces and architecting the appropriate topology of connections. To support the exchange and reuse of application pieces between third parties, standards-based middleware may support a well defined component model [88]. A component model defines the interface between application pieces (the component) and the middleware (the connector) [21].

For reuse to be cost-effective it must be possible to use a component in many systems. This is problematic since the original component developer cannot predict all of the requirements induced by the environment in which components are deployed. This problem can be remedied in many cases by adding features to components at deployment time.

Adding Features at Deployment Time Sometimes distribution forces applications to manage more details than are present in local computations. Information about security [62], distributed transactions [63], or real-time scheduling [75] must be shared between components deployed across the network. Modifying the components to handle many concerns decreases cohesiveness by forcing the component implementation to handle several types of functionality. Current middleware solutions only offer a fixed set of features [87, 62]. However, hand tailoring features that are not offered directly by middleware tangles the implementation of several requirements and destroys program modularity (see Chapter 4 for examples). Furthermore, components are no longer usable when environmental conditions (such as security policies) change. So, hand-tailoring to different operating conditions increases production costs and requires repeated program modification which may decrease reliability.

As we demonstrate in this thesis, distributed heterogeneous software is particularly hard to adapt for deployment in disparate execution environments. We introduce the concept of an adaptation *service* to coordinate cross-host modifications to standard program components, transparently. A service is made up of client- and server-side components called *adaptlets*. At run-time a protocol, *GlueQoS*, can be used to determine what services are activated.

1.1 Crosscutting Features in Distributed Applications

Crosscutting features[1] [45] are those whose implementations stubbornly resist confinement within the bounds of modules. Features such as logging, transactions, security and fault-tolerance typically have implementations that straddle module boundaries even within the most sensible decompositions of systems. This issue has been discussed widely in the literature (see, for example, References [90, 45, 57, 17]; the sample security policy in the next chapter illustrates this issue). The scattered implementation of such features makes them difficult to develop, understand, and maintain. To worsen matters, the requirements of such features are often *late bound*. They are locality dependent, discovered late, and change often—security policies again being a

[1]We use the term “feature” and “concern” interchangeably although there are subtle differences in their connotation. Whereas a feature is perceived as an added value to the user, a concern is more likely to be associated with a software development task.

prime example. Programmers may be confronted with the difficult challenge of making a scattered set of changes to a broad set of modules, often late in the development process.

Software engineers must also consider *non-functional* [58] requirements, such as security, performance, reliability, and quality of service (QoS) requirements when designing component connections. These requirements may require support by software on both the client and server in order to operate properly. For example, software that checks passwords on the server side should be complemented by software that provides passwords for the client. Security requirements, however, may vary with deployment context and even at run-time. We must ensure that components have compatible non-functional features.

This thesis describes middleware services and development tools for the design, development, and deployment of new features for distributed heterogeneous software components. The overall designs are general enough to be applicable for a wide range of implementation languages/platforms. Additionally, we provide support for cooperation between features on heterogeneous systems. Thus, it must be ensured that features to manage concerns that span hosts are coordinated. This is especially important in cases where features interact. At run-time a protocol is used to ensure that composed layers do not conflict with one another. We show that this approach adds some performance overhead but is relatively small when used as part of typical distributed applications.

The services and tools presented in this thesis depend on two key representations about the features added to the system. First, a description of the feature's interface is required so that a feature can be integrated at deployment time with an existing application. This description requires changes to traditional interface description languages (IDLs).

Second, policies are used to describe allowable combinations of features so that agreement of feature usage can be made between network hosts at run-time. We seek to understand whether providing these representations to the middleware reduces the difficulty of adding features to middleware applications at deployment time. This is demonstrated through several example use-cases.

Middleware has often been used as a vehicle for adding features, however, previously no platform has existed for supporting programming and administration of cross-host features. In the next chapter we describe these types of features in more detail.

1.2 Overview of Thesis Goals

As we have discussed, late-bound, crosscutting features such as security require extra functional elements (which can be implemented as *adaptlets*) to be located together with distributed application software components. A client-server pair of adaptlets constitutes an adaptation service. Here we present a discussion of the main goals of the thesis.

Heterogeneity and Communication Adaptlets may need to exchange information and coordinate with each other, and/or with the application components. The adaptation mechanisms may depend on the platform; even so, heterogeneous adaptlets should co-exist and inter-operate correctly.

Binding and Deployment It is desirable to support *late binding* and *flexible deployment* of features. Consider that industry standards such as Java Enterprise Edition allow a fixed set of features to be customized for specific applications at deployment time. Likewise, we would like to allow vendors to build features consisting of adaptlets, independently of application builders, and then allow deployment experts to combine features and applications to suit their needs.

Flexible Communication and Coordination The interaction between a matched pair of client and server adaptlets may not be simple and contained. Under different circumstances, the client adaptlet may require and request different functions (with different parameters) that are supported by a server adaptlet (just as a distributed object can support several distinct methods). Likewise, the server adaptlet may request different post-processing functions on the client side.

Dynamic Service Recognition Several adaptlets, supporting different features, may be associated with an application component; clients and servers must deploy matching sets of adaptlets. In a dynamic, widely distributed context, clients may become aware only at run-time of the adaptlets associated with a server object. Thus adaptlets may need to be deployed at runtime.

Thesis Overview The rest of the thesis proceeds as follows: Chapter 2 presents fundamental material for an understanding of our research, Chapter 3 presents the programming model for developing new features, Chapter 4 presents two examples, Chapter 5 defines the protocol for deploying features at run-time, Chapter 6 discusses related work, and Chapter 7 ends with conclusions.

Chapter 2

Background

2.1 Middleware Features

It is often useful to distinguish between the definition of a core software system and features that may apply to it [102, 5, 45]. A feature is an “optional and incremental unit of functionality” [102]. This is analogous to features in other areas of engineering. For example, it may be more cost effective to engineer an automobile assembly plant in terms of a base product and optional features. An area of software engineering known as product-line architectures [5] is founded on this principle. Other areas such as aspect-oriented software development (AOSD) focus on creating new languages [45] and tools [71] to integrate features into products or identify concerns in existing programs.

We use the term *adaptation* to refer to the addition or removal of software code to extend or modify an existing system. Thus, features are an artifact of requirements, such as security or fault tolerance; adaptations are an artifact of implementation such as code transformations or software wrappers. Systems need to be *adapted* in order to take on new *features*. When software is adaptated by tools (statically) we often say that it has been “instrumented”.

In distributed software development, it is popular to use middleware as a mechanism where features are added to the connection between distributed components [83].

2.1.1 Examples of Features

The kinds of features supported by middleware naturally reflect the concerns relevant to software in a distributed setting. Here we preview two types of features, security and fault-tolerance, as they have been used in existing industry and research middleware platforms.

Distributed components often need to be protected from attacks launched by malicious users of the system. Thus, security in its many forms, including authentication and privacy, is a valuable feature to support in the middleware. Sterne et al. [85] introduced fine-grained access control as a means of applying discretionary controls to individual distributed object method definitions. Intrusion detection can be used to detect attacks launched by so-called "insiders" and was applied to object-oriented middleware [86]. Privacy can be maintained through encryption and most platforms now provide support for standard mechanisms such as transport layer security [26]. Systems can fail for many reasons including hostile users. Next is a description of features used for fault-tolerance.

Several approaches exist to adapt distributed applications for fault-tolerance. The Immune system [59] makes use of a group multicast protocol in order to maintain replicas of application state. However, since application state is captured as uninterpreted streams, Immune does not support heterogeneity. ITDOS [73] builds on the approach and provides support for replication of typed data values to support heterogeneity. The ITUA [69] project builds on intrusion detection technology to provide more efficient recovery. ITUA assumes that replicas will fail according to a particular attack profile and not arbitrarily.

From this small set of examples we can already begin to see that the set of features (and choices of feature implementations) useful to developers is much larger than one can expect to be supported in a monolithic middleware product. Therefore, researchers have relied on other means for extending middleware implementations with new features.

2.1.2 Implementation of Features in Middleware

Layered composition is a popular way to provide new middleware features, when layers can be implemented as separate components. Many language extensions support layered composition, including abstract subclasses [96], mixin layers [81], interceptors [60], and aspects [45].

It is useful to distinguish layered composition from a *layered architecture* [77], for example, the OSI seven layer model [84]. In a layered architecture, each layer provides one set of interfaces, and relies on a (usually) different set of interfaces from the layer below it. The goal is to provide an increasing level of abstraction, and hide details from higher layers. In the case of features presented in the previous section, *all layers* essentially provide a similar interface; layers that add new features are architecturally orthogonal, although semantic interactions may occur. Now we describe how features have been added to middleware using both the DECORATOR [35] design pattern and middleware interceptors.

A decorator layer, also called a wrapper, exposes the same interface to layers above as the interface onto which it is composed. This allows client code (the above layers) to remain unaffected by layer composition. A decorator approach is used in Lasange [94] to provide client customizable remote method invocations. Examples are given for client specific security and business rules. Quality of Objects [65] (QuO) uses decorators that are dynamically chosen based on runtime conditions. This allows QuO to provide services relating to intrusion detection, network bandwidth management, and fault-tolerance.

Interceptors also provide layered composition, but they are completely generic, relying on run-time type information [25]. Information about other layers (including the original application components) is gained dynamically via reflection, and used to monitor and modify application behavior. This provides flexibility at the expense of static type-checking. Many CORBA-based QoS features are implemented using interceptors including security, fault-tolerance, transactions, and real-time features.

This thesis describes an Aspect-Oriented approach to add incremental features. This approach provides the benefits of both decorator and interceptor based approaches because composition with an application can be checked for static type-safety. Additionally crosscutting deployment is simplified, as motivated in the following scenario.

2.2 Security Example

This example illustrates the issues of development and deployment of features in a setting where clients and servers may have different policies with regards to features. The Remote Collaboration Tool [4] (*RCT*) is a distributed multi-user interactive environment for academic and research settings. The architecture is based on client-server remote procedure-call (RPC) through the CORBA object-oriented middleware. Users on the RCT can self-organize into different groups to engage in private chat-room sessions or file-sharing. Groups can be used to support classroom settings where each student is a member of a group for each registered class. Multimedia functionality such as real-time voice and collaborative workspaces are supported.

Consider deployment of the RCT in an environment where two security features are required. The first feature is *authentication*. The RCT must protect certain services from unauthorized access; so client requests must be preceded or accompanied by an authentication step involving the presentation of credentials in order to gain group membership for those services. Credentials can be based on a password, or on public-key signatures [40] In this case, a feature on the server side is responsible for checking credentials, and the corresponding feature on the client-side is required to present the appropriate credentials.

The second feature, the *client-puzzle protocol* (CPP) [24], defends against denial-of-service (DoS) attacks. A DoS attack occurs when a malicious client (or set of malicious clients) overloads a service with requests, hindering timely response to legitimate clients. Certain components of the RCT are prone to DoS attack because of the amount of computation required by the components. CPP protects a component by intercepting client requests and refusing service until the client provides a solution to a small mathematical problem. We discuss the details in Chapter 4.

Adding these two features requires changes to many different components in the RCT implementation. Clients now have to authenticate themselves for certain group related services. Authentication must be added to these client-service requests. Additionally, the CPP protocol must be enabled to protect the DoS prone RCT components.

The changes are crosscutting: programs running on different platforms and in different languages might need changing. Since some platforms may have performance or battery limitations (e.g., PDAs or laptops) or be remotely located, different evolution strategies should be allowed and allowed to inter-operate. Changes to different elements must be made consistently to ensure correct interaction. Changes must be properly deployed in the different elements, otherwise versioning errors may result. Since the functions for Authentication and CPP may apply to other applications, it would be desirable to reuse the same implementations, should the platforms be compatible.

2.2.1 Composing CPP and Authentication

CPP and Authentication interact in interesting ways. For example, suppose the server's only requirement is to prevent DoS attacks. If we trust authenticated clients not to mount DoS attacks, then the authentication and client-puzzle features are equivalent and one can be substituted for the other; it would be redundant to use both. However, sometimes authentication may not imply a decreased risk of DoS attacks, so these features would be viewed as orthogonal. In other situations, we may require both authentication and DoS defense.

Client-side preferences must also be considered when selecting the features that govern a client-server interaction. A client may consider CPP and Authentication to be equivalent, and express a policy that it can use either. A client with a performance requirement, however, would naturally prefer to employ authentication to avoid computing puzzle solutions. A client who values its privacy would prefer to expend CPU cycles in order to not have to reveal their identity; this client may prefer to use CPP rather than provide identity-revealing credentials.

To appreciate our approach to address this type of scenario an understanding of both aspect-oriented software development and distributed object computing is necessary. We provide this background next.

2.3 Aspect-Oriented Software Development

AOSD is a general technique for effecting crosscutting program changes. This is most often achieved by encapsulating the changes to the programs within the lexical confines of a module called an `aspect`.

Aspect-Oriented Programming [45] (AOP) provides an abstraction of the execution of a program and techniques for altering its control and data flow. Although there are many methods for program transformation, AOP is particularly suitable for adaptations on distributed object software. Program execution as defined in AOP matches closely with the notion of interfaces, functions, and objects in distributed object computing (DOC). Thus, we choose to incorporate the terminology and techniques of AOP in our framework. The following definitions help to clarify how program events are used in AOP.

1. Joinpoint

   In AOP the execution of distinguished parts of a program is viewed to generate a meta-program event known as a *joinpoint*. As an example, a "method call joinpoint" includes information such as method name, argument types, or argument values.

2. Pointcut

   A *pointcut* is a query written to select some number of joinpoints out of the execution of a program. A pointcut selects a subset of joinpoints by matching against the information available in the joinpoint events.

3. Advice

   Advice are functions associated with some pointcut and executed at the joinpoints matched by the pointcut. Advice implementations can adapt program behavior by making decisions based on information available at the joinpoints or recorded by previous advice executions.

4. Introduction

   An *introduction* allows new fields (or methods) to be added to existing classes for use by the advice. For our purposes, adding new fields is useful when an advice must associate some state with application objects.

5. Aspect

An `aspect` is a program module used to encapsulate closely related advice and aspect specific statements (such as introductions). The `aspect` scope is used by advice to store information that should be saved between advice triggering.

(1) Aspect	::	**aspect** AspectName **{**(AspectDef | *JavaDef*)* **}**
(2) AspectDef	::	Pointcut | Advice | Introduce
(3) Pointcut	::	**pointcut** PointcutName **(** *ParamList* **)** **:** PCExpr **;**
(4) PCExpr	::	PCD | PCD **and** PCD | **!**PCD | PCD **or** PCD
(5) PCD	::	CallPCD | PCApp
(6) CallPCD	::	**call (** *TypeQuery ClassQuery* **.**(MethodName | Wildcard)
(7)		(PCDParam* **))**
(8) PCDParam	::	**..** | *Wildcard* | *Type* | ParameterName
(9) PCApp	::	PointcutName **(** ParameterName* **)**
(10) Advice	::	(**before** | **after** | **around**) **(** *ParamList* **)** **:** PCExpr { *JavaCode* }
(11) Introduce	::	*Type* ClassName **.** FieldName **;**

Figure 2.1: Mini AspectJ EBNF : Bold font represents terminals, productions in italic font are not shown and productions ending in the suffix "Name" are simply identifiers.

Figure 2.1 shows the syntax for a simplified version of the AspectJ language. An `aspect` can be declared (line 1) much like a `class` in Java. It may contain standard Java definitions or specialized aspect definitions. An aspect definition (line 2) is a `pointcut`, an advice, or an introduction.

A `pointcut` (line 3) is a named query over joinpoints. The `pointcut` has a formal argument list that is used to capture information from a *pointcut expression* (line 4). This expression may combine primitive queries using standard logical operators. A primitive query is known as a *pointcut designator* (line 5). We show only two designators.

The first, `CallPCD` (line 6), identifies calls to particular methods. Methods can be chosen based on (in the order given): the return type, enclosing class, method name, or parameter list.

The second, `PCApp` (line 9), is the application of a named `pointcut`. Here free variable names are used in place of the actual arguments. This is used to bind the information expressed in the pointcut designator to the arguments of an advice (line 10).

An advice (line 10) is a pairing of a pointcut expression and the Java code to be executed when the pointcut is matched. Advice may be declared to execute `before`, `after`, or `around` (in place of) the original joinpoint. The body of the distinguished `around` advice can use a special keyword, `proceed`, to allow execution to flow to the original joinpoint.

The last production in the figure is an introduction (line 11).

AOP is a general technique for encapsulating crosscutting concerns. Our research goals have applied these abstractions for coordinated adaptations of distributed object applications.

2.4 CORBA

A popular form of distributed software design has come to be known as Distributed Object Computing. Many different DOC standards are in existence and DOC software is being used for applications in both business and military applications. DOC extends the traditional RPC primitive by managing a dynamic number of stateful program entities at one traditional application endpoint (IP Address/Port). Because our research has focused on this type of middleware we present in detail some definitions and concepts from this area.

1. Interface

An interface is a named collection of procedure signatures (RPC signatures) including procedure return values, formal parameters, and exception conditions. An application component is said to *implement* an interface by creating a mapping from all the procedures in some interface to programming language procedures.

2. Application Component

   We have used the term application component thus far without defining it. An application component is a set of programming language procedures that share the same static data scope. Different application component implementations can be used to implement the same interface.

3. Distributed Object

   An application component that implements an interface can be registered with a local middleware daemon server (called an ORB, for Object Request Broker) to become a *distributed object.* Any number of application components can be registered at run-time. Each registration creates a new *instance* with its own globally unique name and memory storage scope.

4. External Reference

   The external reference is a globally unique name that can be used by client programs to initiate communication with distributed objects.

5. Server Stub

   A server stub is a component in some programming language that exports the same interface as a server object. Client side middleware can convert an external reference into a stub to give the client access to the server's services. The stub contains generated code to handle *marshaling.*

Marshaling (also known as serialization) is the process of converting data types from a programming language to a stream of bytes for network transmission. This process is inherently not statically type-checkable. Therefore it is advantageous to prevent programmers from explicitly writing marshaling code.

6. Server Skeleton

   In DOC a server side RPC stub is known as a *skeleton*. When application components implement an interface, the skeleton is used to connect programming language procedures for dispatch by the middleware.

DOC middleware only offer a fixed set of features with limited customizability of feature semantics. The next chapter presents our approach to address this limitation by incorporating AOP.

Chapter 3

Dado

This chapter describes Dado, an approach to developing features in distributed systems that require code changes, in a heterogeneous setting, to both client- and server-side of a distributed interaction.

Currently, the process of building $\mathcal{D}H$ systems using middleware such as CORBA begins by describing the high-level interfaces using IDL. IDL specifications are then implemented by developers on different platforms and perhaps in different languages. When implementation is complete, the users of the distributed system can run ORBs on a network as suited to the application and organizational needs, and deploy the constituent application objects.

Dado brings three new roles into this process (see Figure 3.1): a *service architect*, *service programmer*, and a *service deployment expert*.

This service architect can design a $\mathcal{D}H$ service that implements a crosscutting feature. The process begins with the description of a Dado `service` in an enhanced IDL (known as DDL). A `service` is a pair of client/server *adaptlet interface* descriptions, which consist of several operations, just like an IDL interface. These interfaces are then compiled using DDL compilers for a target implementation language (e.g., Java or C++), producing marshaling routines and typing environments. The imple-

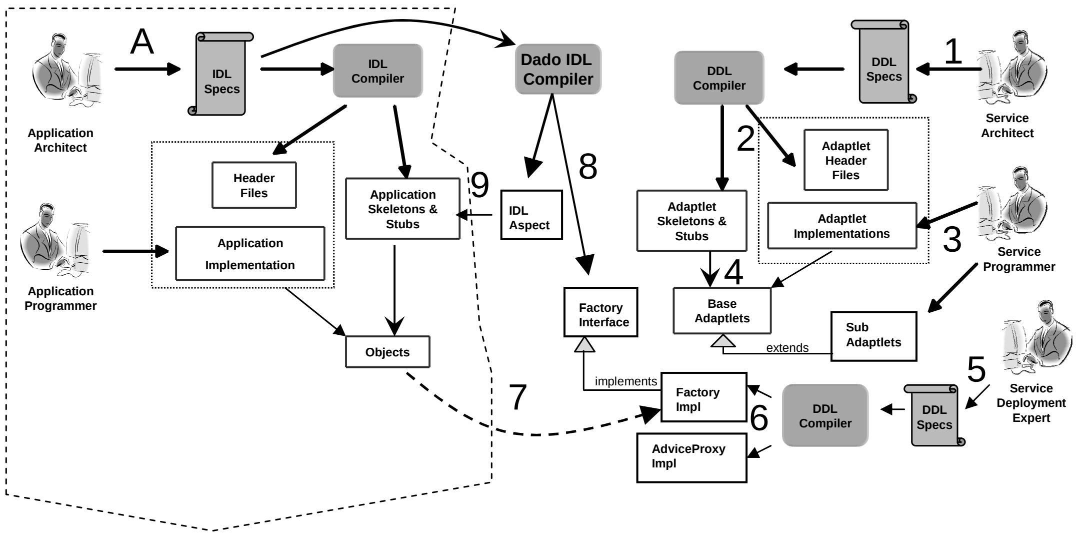

Figure 3.1: *Dado Development Process* The left hand side (within the dotted lines) indicates the conventional CORBA process. On the right, the Dado service development begins (1) with modeling the interface of Dado adaptlets using DDL; from this the DDL compiler generates (2) marshaling code, and typing environments for adaptlet implementations. The programmer writes (3) the adaptlet implementations and links to get (4) the adaptlets. Now, the deployment expert extends adaptlets using (5) pointcuts, and these are used to generate (6) a FACTORY for obtaining `AdviceProxy` implementations at run-time (7). Applications are instrumented independently by the Dado IDL compiler. An AspectJ aspect or TAO PROXY is generated (8) and used to modify the existing application (9).

mentation then proceeds by service programmers.

The deployment expert binds an implemented service to a given application by specifying bindings using a pointcut language. The deployment expert will need to understand both the application and the service, and select the bindings based on the specific installation. Applications are instrumented by modifying the application stubs and skeletons. A specialized CORBA IDL compiler (the Dado IDL compiler) is provided for this purpose.

This chapter presents the details of the Dado features outlined in the Introduction chapter. The current Dado experimental implementation is based on the CORBA standard. It includes the DDL language for adaptlets, a DDL compiler for C++ and Java, and the Dado IDL compiler to modify applications for two different open-source ORBs (JacORB [8] and the TAO [74] ORB).

Timing Example This chapter uses the running example of a remote invocation timing monitor. This simple example is used to ground the discussion of newly introduced concepts. Here we discuss the high-level requirements for the timing implementation.

One could write code (e.g., using interceptors [60, 62] or aspects) to attach to the client that will record the time each invocation leaves and each response arrives. However, the client may also want the invocation arrival-time at the server, and the reply sending-time in order to compute the actual processing time. This scenario demands more coordination between interacting client and server adaptations.

This chapter, through the example, demonstrates how Dado provides four critical elements. First, clients must be able to ask the server for timing statistics; only some clients will request this feature. Second, servers may return data through a type-checked interface. Third, clients need some way to modify existing software to add logic for requesting timing statistics; different means should be allowed. Finally, client and server adaptations should be coordinated; clients will not request timing statistics from servers unable to provide them.

Note on presentation syntax Throughout this chapter fragments of our Java and AspectJ based implementation are presented. The pseudo code is written in Java 1.5 and AspectJ syntax. Many details have been elided to showcase the relevant design choices in our implementation. Also, we do not show `try/catch` blocks, default constructor calls, exception handling code, visibility modifiers, or packages.

The chapter begins with a summary of the DDL language. Section 3.2 describes the instrumentation of CORBA applications. Section 3.3 describes adaptlet advice. Section 3.4 describes adaptlet communication. Section 3.5 describes deployment of adaptlets. Section 3.6 presents measurements. Finally, Section 3.7 concludes with an overall view of the work.

3.1 Dado Description Language

Dado adopts the philosophy that IDL-level models provide an excellent software engineering methodology for distributed systems. In addition to promoting better conceptualization of the design, one can construct tools to generate useful marshaling code and typing environments for static type-checking.

To gain an understanding of how a service architect works with DDL, it is useful to examine the elements of the DDL language presented in Figure 3.2. DDL introduces the notion of a *service* (line 1) that refers to a crosscutting feature. A service comprises a client and/or server *adaptlet* (lines 3 and 4).

A client and server adaptlet can be described together in a shared scope by declaring both inside the `service` element. The service can be viewed as a collaboration [82] of client and server adaptlets. The client and server adaptlets described in this way share a common name, used to coordinate run-time deployment. The scope also pro-

(1) Adaptlets	:: **service** Name Inherits? **{** Client? Server? **};**
(2) Inherits	:: **:** Name*
(3) Client	:: **client** **{** Element* **};**
(4) Server	:: **server** **{** Element* **};**
(5) Element	:: AdviceOp | Request | *Pointcut* | *Advice*
(6) AdviceOp	:: Around | Operation
(7) Around	:: **around** *Type* Operation
(8) Request	:: **request** Operation
(9) Operation	:: Name **(** *Parameter** **);**
(10) Name	:: *String*

Figure 3.2: Dado Description Language (DDL) EBNF : Used by Service Architects (item 1) and Deployment Experts (item 5) in Figure 3.1. Bold font represents terminals; productions in italic font are not shown but follow from Figure 2.1.

vides a way to coordinate client and server adaptlets at the interface level. This may be needed at deployment time, through shared pointcuts, and at run-time, through `request` messages.

3.1.1 Adaptlet Inheritance

Adaptlet interfaces can be extended through object-oriented multiple interface inheritance (line 2). Client adaptlets in a sub-`service`[1] implicitly inherit from the client adaptlets in the super-services; server adaptlets are similarly extended.

Inheritance provides for the description of generic services that are subsequently tailored for specific applications. The service-architect will write interfaces for client and server adaptlets, which are then implemented by service-programmers. A de-

[1]We use the terms sub-,super-, and base-service/adaptlet in analogy to sub, super, and base classes.

ployment expert can then add an adaptlet to an application by extending either the adaptlet interfaces (through DDL multiple inheritance), adaptlet implementations (through Java single inheritance or C++ multiple inheritance), or both. The inheritance model expands the notion of AspectJ's `aspect` inheritance to client/server adaptlet collaborations.

3.1.2 Adaptlet Operations

Each adaptlet may support several operations, which may be of two different kinds. Advice operations (line 5) are specified as standard operations or optionally tagged as `around` (line 6). Advice operations provide only the signature of an advice; the body is implemented in a specific programming language. Advice operations may be bound via `pointcut` patterns (like AspectJ advice) to application interfaces. They basically provide additional code that is run *every time* certain operations defined in an IDL interface are invoked. Code in the distinguished `around` advice have full access to modify intercepted application information, whereas other advice do not (as in AspectJ).

In addition to advice operations, DDL services can also include `request` operations (line 7). These are a form of asynchronous messages that may be triggered by any adaptlet advice. Advice and `request` are explained in more detail in Sections 3.3 and 3.4.

3.1.3 DDL for Timing Example

In Figure 3.3, we see the base-service for the `client` and `server` Timing adaptlets. Each adaptlet includes one `request` operation.

When `timeRequest` is invoked by the client adaptlet, a `request` message is added to the application invocation and dispatched to the server-side adaptlet. The server adaptlet can respond to a `timeRequest` by taking two timing measurements to deter-

```
service Timing {
   client {
      request timeResult(long received, long sent);
   };

   server {
      request timeRequest();
   };
};
```

Figure 3.3: Timing `service : client` and `server` adaptlets. Written by Service Architect as in item 1, Figure 3.1.

mine the actual execution time for that application method invocation. It can then report the results back to the client, using the client request `timeResult`.

In order to trigger this exchange of `request` messages some advice must be added to the client side. Later Section 3.3 shows how a deployment expert can extend the client adaptlet to include the advice.

3.1.4 DDL Language Mapping

The DDL compiler generates typing environments, as well as stub and skeleton functions. The generated typing environments (C++ header files or Java interfaces) ensure that the pieces of a `service` can safely inter-operate. Adaptlets can currently be implemented in either C++ or Java but must be written in the same language as the application. This is primarily for performance reasons; if adaptlets are in a different language, it would be necessary to endure an expensive traversal of middleware to get from an application object to an adaptlet.

Each adaptlet interface in a `service` is used to generate the appropriate programming language interfaces and communication code. These are then used by service programmers to build the adaptlet implementations. Figure 3.4 shows the Java implementation code for the client base-adaptlet.

```
(1) class Timing_Client_Impl implements Timing_Client {
(2)
(3)    void initialize(Timing_Server_Stub that) {
(4)    }
(5)    void timeResult(long received, long sent) {
(6)       System.out.println(recevied + " " + sent);
(7)    }
(8)}
```

Figure 3.4: Timing Implementation. Written by Service Programmer as in item 3, Figure 3.1.

From the `service` in Figure 3.3, the DDL compiler can generate the interface `Timing_Client`[2] which is referenced in Figure 3.4, line 1. The declaration of the `interface` (not shown, however implied by the implementation) includes two methods.

The first, `initialize`, allows the implementation to obtain a reference to the server side stub. The reference could be used to communicate through `request` operations declared on the server side, such as `timeRequest` (Figure 3.3, line 7). In the implementation code (lines 3 and 4) the reference is simply discarded. In the next section, we will see that for the `Timing service`, the reference is only relevant to a client sub-class.

The second method defined in the interface is `timeResult` (lines 5 and 6). This method will be invoked whenever the server-side adaptlet adds a `timeResult request` by using a `client` adaptlet stub. Here, the implementation simply prints out the results.

In this section we provided an introduction to the adaptlet interface language called DDL. We saw how the DDL is used to generate interfaces implemented by `service` programmers. The implementation could make use of stubs that refer to the matched `client` or `server` adaptlet (described in more detail in Section 3.4). The

[2]The suffix _Client is appended for a client adaptlet and _Server for a server adaptlet.

details presented were motivated by three concepts: that adaptlet implementation should be provided a statically typed-environment for communication, that communication details would be automatically generated, and that generic and application specific components could be separated through inheritance.

3.2 Application Instrumentation

In order to trigger adaptlet behavior at runtime, application code must somehow be modified, or execution intercepted to capture the right events. A wide range of binary and source-code, static and dynamic instrumentation mechanisms have been reported [14, 91, 45, 72]. Middleware, also, can support highly dynamic reflective mechanisms [6, 28]. In keeping with the $\mathcal{D}H$ philosophy, we have explored heterogeneity in the implementation of the triggering mechanism. Thus, while the pointcut specifies the high-level design of the binding, different implementation strategies are possible.

Applications are prepared statically, for adaptation by adaptlets at run-time. This is done in a generic fashion. The preparation steps do not depend on the adaptlets that will be added. Instrumentation takes place on the client and server CORBA stubs and skeletons. Each interface in an application has a corresponding stub and skeleton. Consider the simple `interface StockServer` in Figure 3.5. It provides an operation, `getQuote`, for retrieving a stock price for a given ticker symbol. The other operation, `setQuotes`, sets the current price for multiple stock quotes.

Now we present some details of the AspectJ code generated by the Dado IDL compiler. Our C++ implementation will be explained in terms of its differences from what we have already presented.

```
struct Quote {
  float price;
  string symbol;
};
interface StockServer {
  float getQuote(in string symbol);
  void setQuotes(in sequence<Quote> list);
};
```

Figure 3.5: Stock Interface : Written by Application Architect in item A, Figure 3.1.

```
(1) aspect StockServer_Client_Aspect {
(2)     AdviceProxy StockServer_Client._getQuotes;
(3)     AdviceProxy StockServer_Client._setQuotes;
(4)
(5)     //Invocation interception advice (see Figure 3.8)
(6)
(7)     //Constructor introduction (see Figure 3.14)
(8)}
```

Figure 3.6: Aspect for instrumenting application. Generated by Dado IDL compiler as in item 7, Figure 3.1.

3.2.1 IDL Interface Aspect

In Figure 3.6, the `aspect StockServer_Client_Aspect` (line 1) facilitates transformation (item 8, Figure 3.1) of a generated client stub. The code is generated by the Dado IDL compiler (item 7, Figure 3.1), directly from the CORBA IDL. An `aspect` is also used for transformation of server side skeletons. The code is symmetric, involving only changes to naming conventions, so the details are not presented.

On lines 2 and 3 a field introduction is generated for each operation in the `interface`. This field can be bound to a list of `AdviceProxy`s by the Dado runtime. An `AdviceProxy` manages the interception of an application operation for a particular adaptlet advice.

Next, an AspectJ `advice` is generated for each operation in the `interface` (comment on line 5, presented in Figure 3.8). This advice will intercept operation invocations and pass control to the `AdviceProxy` that is the head of the appropriate adaptlet advice list. This indirection is necessary so that advice may be installed dynamically. This is necessary because AspectJ does not support dynamic installation of advice. We return to the `AdviceProxy` in Section 3.3.

Finally, line 7 provides an AspectJ introduction to intercept stub creation. The body of an introduced constructor binds relevant `AdviceProxy` lists (i.e., the head of the list) to each of the generated `AdviceProxy` fields as specified by any DDL pointcuts. In this way, our dynamic Dado advices are bootstrapped into the application by the static AspectJ advice.

Before describing more details of our implementation we present an example flow of client to server communication for the `getQuote` operation where two adaptlet advices are in use. Each advice is represented by an `AdviceProxy`.

First, the client application would make an invocation by calling a method on a local stub. Assume that the stub has been instrumented by our Dado tools with an AspectJ aspect such as that shown in Figure 3.6. An advice in this aspect would delegate control to the head of the `AdviceProxy` list represented by the field in Figure 3.6, line 2. Next, the proxy will call an advice operation on the adaptlet that it stands in for. The code in the advice, implemented by a Service Programmer, can queue request messages to be delivered at the server. After the advice has executed, the `AdviceProxy` delegates to the next `AdviceProxy` in the list. Finally, at the end of each list is a special `AdviceProxy` that delegates to the *actual* CORBA operation invocation.

On the server side, the designated advice adaptlet operations for each adaptlet get executed, as are the queued requests. The server side adaptlets may also queue requests to be executed by the client side. This feature can be used to pass information back to client-side adaptlets; we illustrate this in the monitoring example where server-side time-stamps are passed back to the client via a `request` adaptlet operation.

3.2.2 AdviceProxy

```
(1)abstract class AdviceProxy {
(2)   AdviceProxy next;
(3)   Object instance;
(4)   abstract void run(AdviceContext context);
(5)}
(6)class AdviceContext {
(7)   Object[] arguments;
(8)   Object returnValue;
(9)}
```

Figure 3.7: Abstract AdviceProxy class and the AdviceContext data structure. Dado framework classes.

At run-time, each pairing of adaptlet advice to `interface` operation, matched by DDL `pointcut` bindings, is represented as an instance of an `AdviceProxy` class, shown in Figure 3.7.

`AdviceProxy`s are linked through the `next` field (as in line 2) into AdviceProxy lists. Each `AdviceProxy` is responsible for dispatching to the appropriate advice operation of the adaptlet instance (line 3) to which it is associated, and then delegating to the `next AdviceProxy` on the list. This allows multiple advices to affect a single joinpoint.

The implementation of `run` (line 4) accepts an `AdviceContext` and adapts its contents for the invocation of an adaptlet advice. The `AdviceContext` (line 6) carries the arguments of the application operation invocation to replace the formal arguments of an advice. The implementation of `run` (in a generated subclass) is responsible for the mapping. The mapping is determined by matching the adaptlet provided `pointcut`s with the application interfaces. Likewise, the `returnValue` is made available by generated type casts.

By generating a specific `AdviceProxy` for each advice/joinpoint pairing we ensure that the `AdviceContext` is typed according to the DDL `pointcut` bindings.

3.2.3 IDL Operation Advice

```
(1) Object around(StockServer_Client thiz) :
(2)     execution(* StockServer_Client.getQuotes(..)) && this(thiz)
(3){
(4)     AdviceContext context;
(5)     context.args = thisJoinPoint.getArgs();
(6)     CDROutputStream buffer;
(7)     RequestStack.push(buffer);
(8)     _getQuotes.run(context);
(9)     RequestStack.pop();
(10)    return context.returnValue;
(11)}
```

Figure 3.8: Instrumention for a specific interface operation. Generated by Dado IDL compiler as in item 7, Figure 3.1.

In order to begin dispatch to the adaptlet advice list, the operations in the generated IDL stubs must be modified. This modification is enabled by a generated AspectJ `advice` as shown in Figure 3.8. On line 2, a `pointcut` is generated to intercept a specific operation in the particular IDL stub class. The arguments of the operation are fed to the `AdviceContext` structure in line 5. The call to `thisJoinPoint.getArgs()` is a reflective means, available in AspectJ, of obtaining all the arguments to the method as an array of `Objects`. This was not available for our C++ implementation so these details are accounted for through additional generation of code. In either implementation, the array is mapped to adaptlet advice operations in a statically type safe manner by means of the generated `AdviceProxys`.

Now, a new *request* message buffer is created and pushed onto a stack of request buffers (line 7). The buffer is of the type `CDROutputStream`. The `CDROutputStream` stores *request* messages encoded in the CORBA Common Data Representation (CDR). A `RequestStack` is required for pushing multiple outstanding *request* buffers, since adaptlet advice may themselves invoke remote operations. When adaptlet advice makes remote invocations, it may recursively trigger other `request` messages *only* relevant for the *new* invocation. The buffers are popped and attached to invocation messages as they are sent out. The stack references are `ThreadLocal`; the Java run-time associates an instance to each thread, so concurrent threads only see a stack relevant to the `request` messages triggered on that thread.

In line 8, the dispatch is made to the head of the `AdviceProxy` list for this operation. Finally, the used buffer is popped from the stack and a final value is returned to the application after possible modification by intervening adaptlet advice.

For C++, we make use of two mechanisms available in the TAO ORB: the TAO smart proxies [98] and the Object Reference Factory [62]. Similar mechanisms do not exist in JacORB, so AspectJ was a natural alternative.

Both the smart proxy and the Object Reference Factory essentially provide the same functionality for our purpose; details vary in the implementation. The smart proxy acts on the client and the Object Reference Factory acts on the server. Both allow stubs or skeletons to be replaced by a PROXY [35]. The proxy maintains a reference to the real stub or skeleton (i.e., the delegate). Equivalent code that was implemented as AspectJ advice in our Java implementation is generated in the methods of the C++ proxy. Proxies are generated for each IDL `interface` that parallel the aspects generated for Java.

Naturally, client- and server-side adaptlets, even if using different languages or different instrumentation mechanisms, are fully inter-operable. The adaptlet programmer remains agnostic with respect to the actual instrumentation mechanism that is used to trigger the adaptlet.

3.3 Advice Operations

Recall that Dado introduces several new service-related roles into the software process: a *service architect*, *service programmer*, and a *service deployer*. When a service architect decides that some additional behavior on the client or server of a

```
(1) service TimeStockServer :  Timing {
(2)    client {
(3)       void timedOperation();
(4)       before call(float StockServer.getQuote(string)) :
(5)              timedOperation();
(6)    };
(7)};
```

Figure 3.9: Timing deploy. Written by Deployment Expert as in item 5, Figure 3.1.

distributed application is desirable, she can add an advice operation to the interface of an adaptlet. Advice operations can be specified to be client-side or server-side advice. The service deployer can then add the behavior specified by the advice interface to a specific application object by writing an appropriate `pointcut`. The service programmer has the obligation to implement each advice.

In the example, to deploy `Timing` adaptlets for a given application object, the server-side would make the `service` available by registering a server `Timing` adaptlet. When clients become aware of those server objects, the Dado run-time will automatically deploy client adaptlets based on the client's pointcuts (Figure 3.9).

In this deployment the client would like invocations to the `getQuote` operation to be intercepted by the advice `timedOperation`. Here, a client deployment expert extends the `Timing` adaptlet interface independently from the server. The server side is unaware of the advice declaration (line 3) for `timedOperation` and the pointcut binding on lines 4-5. The server-side does not need to specify any additional pointcut instructions, as the `request` operation `timeRequest` is invoked dynamically by the client-side adaptlet.

The `Timing` extension `TimeStockServer` generates an `interface` which extends the generated `interface`, `Timing_Client`. In Figure 3.10, the `interface` is used (line 2) to extend the implementation of `Timing_Client_Impl` as presented in Figure 3.4.

```
(1) class TimeStockServer_Client_Impl extends Timing_Client_Impl
(2)                                   implements TimeStockServer_Client {
(3)    TimeStockServer_Server_Stub _that;
(4)    void timedOperation() {
(5)       _that.timingRequest();
(6)    }
(7)    void initialize(TimeStockServer_Server_Stub that) {
(8)       _that = that;
(9)    }
(10)}
```

Figure 3.10: TimeStockServer implementation. Written by Service Programmer as in item 3, Figure 3.1.

A new advice, `timedOperation` has been added[3] by the deployment expert, to trigger the timing `requests` at a particular client joinpoint. This is achieved on line 5 by invoking the `timeRequest` operation on the provided server adaptlet stub.

The separation of advice and request operations in the adaptlet interface represents two levels of adaptation required to implement crosscutting distributed heterogeneous services. In this section, we detailed how advice operations would be used by the different roles in the Dado development process and how they are implemented in our prototype middleware tools. Next, we see how adaptlets communicate by modifying client/server application communication using `request` messages.

3.4 Adaptlet Requests

Some services can be implemented simply by executing advice on the client- or server-side. However, in some cases, additional information may need to be sent along from the client to the server side adaptlet (or vice versa).

In our running example we presented a service where a client-side adaptlet can

[3]Since the implementation of timedOperation is not application specific it could have been included in the base class. However, since this is not necessary for coordinating client and server, we included it in the client specific sub-class.

request that a matching server adaptlet calculate server processing time for specific invocations, and then communicate this information back to the client adaptlet. This additional information conveyed between client and server adaptlets is contextual. It must be associated with some original CORBA invocation. Likewise, the timing behavior by the server adaptlet must occur before and after the processing of the invocation for which the client adaptlet requested statistics.

The service architect can include operations tagged with the `request` modifier keyword to provide an extra communication path between `client` and `server` adaptlets that is associated with the current CORBA invocation. The body of `client` and `server` advice can be programmed to add `request` messages by using a reference which exposes the interface of request operations available to a client adaptlet by the server adaptlet and vice versa.

Adaptlet requests give service developers more ways of programming interactions between client-side and server-side adaptlets. Consider that a client adaptlet may require different types of actions to be taken at the server side. As a very simple example, a per-use payment service adaptlet attached to a server object might accept e-cash payments, or a credit card. Another example is authentication. It could be based on Kerberos-style tokens, or on a simple password. We could include both options as possible parameters, in a single operation signature, along with an extra flag to indicate the active choice; this leads to poorly modularized operations with many arguments. Rather, we take the "distributed object" philosophy of supporting different requests at a single server object; we allow adaptlets on either side to support several different requests.

Adaptlet requests are implemented as one-way, asynchronous "piggy-backed" message that are sent along with an invocation (from client to server) or a response (vice versa). Since multiple services can be present simultaneously, the requests are queued on each client and packaged with the original invocation.

```
(1) class Timing_Client_Stub {
(2)    void timingResults(long before, long after) {
(3)       CDROutputStream out = RequestStack.top();
(4)       out.write_string("Timing_Client");
(5)       out.write_string("TimingResults");
(6)       out.write_long(before);
(7)       out.write_long(after);
(8)    }
(9)}
```

Figure 3.11: Timing_Client_Stub. Generated by DDL Compiler as in item 2, Figure 3.1.

The generated stub for the Timing client adaptlet is presented in Figure 3.11. A reference to this stub is made available to the server side adaptlet for queuing `timingResults requests`.

In line 3, a reference to the top of the `RequestStack` for the current thread is obtained. First, the interface name of the adaptlet for which the request is intended is written to the stream. In our example, this `request` would be dispatched to the `TimeStockServer_Client` implementation because it implements the `Timing_Client` interface.

Next, in line 5, the name of the `request` operation is added. This is so the appropriate operation is invoked on the client-side. Operations are dispatched through a `switch` statement in the corresponding `Timing_Client_Skeleton` (not shown). In this `switch`, a `case` for the `timingResults` operation deserializes the arguments that were serialized in line 6 and 7, to be used as actual arguments in a call to `timingResults`.

Advice and `request` play different roles in adapting the execution of a distributed application. Advice operations are used to add code at points in the program determined by pointcut based deployment. Although the addition and removal of advice can occur dynamically at runtime it is still based on referring to static elements in the IDL interface. Pointcuts create a connection between client programs and client adaptlets or server objects and server adaptlets only. The connection between client adaptlets and server adaptlets is made through request messages and is completely dynamic. The request messages serve both to convey additional information and invoke behavior to process the information. In essence, request messages provide a form of dynamic per-invocation adaptation [94] while supporting type-checked interactions and modular design through IDL declaration.

3.5 Transparent Late Service Binding

In a WAN environment such as the Internet, where servers are discovered at runtime, clients cannot predict the set of services provided by (or required by) a particular server until it is located. Static approaches that install new services based only on type information cannot easily provide this kind of late binding.

When server objects are associated with a Dado service (this happens at deployment time via a configuration file), they are assigned an external object reference that is used by the client side run-time to detect the applicable services.

Essentially, the reference encodes information about the adaptlets associated with this object. This information is used by the Dado interception logic on the client-side to transparently engage the corresponding client-side adaptlets.

The process is graphically illustrated in Figure 3.12. When an application object registers itself with a naming service, the reference encodes all active services (Arrow 1). Subsequently, a retrieved reference (2) is intercepted by the Dado runtime, which decodes the applicable service identifiers from the reference. For each operation in the interface, it injects the appropriate advice into the execution path of invocations originating from the client.

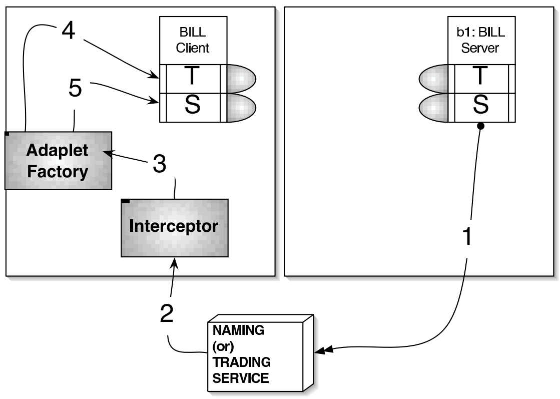

Figure 3.12: *Late-binding service adaptations* (1) Server object, with Security and Transaction adaptlets, named "b1" of type "Bill" is registered with a Naming service. The identifiers "Transaction" and "Security" are tagged to the external object reference. When client looks up object named "b1", the returned object reference (2) is intercepted by Dado component. Dado attempts (3) to find client-side adaptlets for "Transaction" and "Authentication" from client-side factory. Factory creates and binds transactions (T) and security (S) adaptlets to client application object.

3.5.1 Advice Factory

This transparent engagement is achieved through *Advice Factories.* Each adaptlet implementation is packaged with a generated class that provides `AdviceProxy` lists for specific IDL operations.

Figure 3.13 shows the `AdviceFactory` for the `TimeStockServer_Client` adaptlet. It implements the `StockServer_AdviceFactory`, so it can provide advice for the operations declared in the `StockServer` interface. These `AdviceFactory` interfaces are generated for all IDL interfaces in the application. In general, an adaptlet `AdviceFactory` will implement the factory interface for any IDL interfaces that it crosscuts.

When a client stub or server skeleton are created, an *introduced* constructor (Figure 3.14) will iterate through the operations in the `AdviceFactory`s for applicable adaptlets to obtain the advice lists. In Figure 3.13, no advice is returned for the `setQuotes` operation (line 3) and a single advice is returned for the `getQuote` operation (line 7). This return value in line 7 is the implementation of the `AdviceProxy`

```
(1) class TimeStockServer_Client_AdviceFactory
(2)                     implements StockServer_AdviceFactory {
(3)    AdviceProxy StockServer_setQuotes() {
(4)       return null;
(5)    }
(6)    AdviceProxy StockServer_getQuote() {
(7)       return new
(8)       TimeStockServer_Client_timedOperation_StockServer_getQuote();
(9)    }
(10)   Object getInstance() {
(11)      return Class.forName(System.getProperty("TimeStockServer.Impl"))
(12)                              .newInstance();
(13)   }
(14)}
```

Figure 3.13: TimeStockServer Factory. Generated by DDL compiler as in item 6, Figure 3.1.

class that will dispatch the `TimeStockServer_Client` advice, `timedOperation`, whenever `getQuotes` is invoked. A Factory Method [35] is generated on line 8-10 to obtain an instance of the implementation for the `TimeStockServer_Client` adaptlet. The instance is created through reflection of the environment variable "TimeStockServer.Impl", which names the implementation class.

```
(1) StockServer_Client.new() {
(2)    List<StockServer_AdviceFactory> factories = decodeReference(this);
(3)    for(Object factory :  factories) {
(4)       Object adaptlet = factory.getInstance();
(5)       _setQuotes.append(factory.StockServer_setQuotes(), adaptlet);
(6)       _getQuotes.append(factory.StockServer_getQuotes(), adaptlet);
(7)    }
(8)}
```

Figure 3.14: Template for initializing advice lists. Generated by Dado IDL Compiler as in item 2, Figure 3.1.

Figure 3.14 shows the generated code to bind adaptlets to an application at runtime. This is done by introducing an explicit no-argument constructor in place of the default no-argument constructor for a client stub. The constructor obtains the

`AdviceFactory` objects for all the adaptlets found in the server reference (line 2). The code for the function `decodeReference` is not shown. Next, we iterate over all the factories. Each iteration obtains a reference to the adaptlet implementation (line 4) and appends the `AdviceProxy` from each factory to the current lists. This is done for each operation in the IDL interface (line 5 and 6). During the append process (not shown), the `instance` field for each `AdviceProxy` is set to the adaptlet reference obtained from the factory.

The late-binding process described in this section ensures that client adaptlets are deployed to match adaptlets on the server.

3.6 Measurements

The data presented is in the style of micro-benchmarks: we measure the incremental effect of the actual additional marshaling work induced by the new communication code (generated by DDL compiler), as well as for dispatching adaptlet advice and request. For this reason, we use null advice and request methods that do no computation, so that we can focus primarily on the actual overhead of the runtime.

The measurements were taken for a single client server pair. The client machine was a 1.80 GHz Intel Pentium with 1GB main memory running Linux 7.1. The client middleware was JacORB 1.4 on JDK 1.4. The server machine was an 800 Mhz Intel Pentium Laptop with 512MB main memory running Microsoft Windows 2000. Server software used TAO 1.2 compiled in C++ Visual Studio. The DDL interface to the adaptlet used for performance measurement is shown in Figure 3.15.

As can be seen, there is one client-side advice and one server side request. The client-side advice is bound to a method call (with a single argument of type `string`) by the pointcut. In our implementation, the client-side advice simply captures the string argument from the invocation and calls the server side request, passing along

```
service Test{
  client {
    void grabArg(string arg);
    before call(* StockServer.getQuote(arg)) :
           grabArg(arg);
  };

  server {
    request putArg(string arg);
  };
};
```

Figure 3.15: Test Adaptlets

the string argument. So the overhead we are measuring (beyond the normal CORBA invocation overhead) includes the additional cost of: intercepting the invocation on the client-side, dispatching the client-side advice, executing the client-side request stub, marshaling the additional data, transmitting the additional data over the wire, unmarshaling the data on the server side, and dispatching and executing the request implementation on the server side. All measurements given here are for round-trip delays for a simple invocation that sends a “hello world” string. The data is averaged over 1000 invocations, and is given in milliseconds.

Experiment	**100 Base-T RPC (ms)**
1. Plain CORBA	0.65
2. with 1 advice execution, 1 request	1
3. with 10 advice executions, No request	0.68
4. with 10 advice executions, 10 request	1.52
5. Plain CORBA with equivalent raw data Payload for 10 requests	1.38

The first row is the plain unloaded CORBA call, as a baseline for comparison. The second row is a CORBA call with one adaptlet advice, and one additional request. In the third row, we show the effect of applying the advice 10 times. The fourth row shows the effect of executing the advice shown on the second row 10 times, and attaching a request for each advice. The critical *fifth* row shows an interesting comparison: it measures the plain CORBA call, with additional "piggy-backed" data, *exactly equivalent to 10 request messages*, without any adaptlet code whatsoever. This row corresponds to the precise straw-man comparison for sending data without Dado, and corresponds to the way interceptor-based services (such as Transactions and Real-Time, as per [62], page 30 of Chap. 13) are currently programmed.

As can be seen, the advice itself, which does not send any data, does not induce very large overheads (comparing rows 1 and 3, it is about 5% in both cases for 10 advice invocations). We believe the overhead for sending *request*s is largely due to the base cost of marshaling and adding our "piggybacked" messages. Addition of messages is achieved through calls to a middleware-level CORBA API.

3.7 Conclusion

We conclude here with several observations about Dado.

The design space of a convenient framework to implement $\mathcal{D}H$ crosscutting services is quite large, comprising many dimensions such as synchronization mechanisms, scope of data, and the handling of exceptions. The current implementation of DDL has made some reasonable choices, but other choices will need to be explored as other application demands are confronted. Some examples: "service-scoped" state, i.e., state that is implicitly shared between adaptlets; services whose scope transcends a matched stub-skeleton adaptlets; other (e.g., synchronous) interactions between adaptlets.

Feature interactions are a difficult open research issue that Dado services must deal with. The next chapter describes an approach to negotiate selection of services when feature interactions induce preferences over possible configurations. Note that it is also possible to program interactions between two Dado services: one can write a third service that pointcuts adaptlets in each, and responds to the triggering of both by preventing one from running, changing argument values, return values etc. However, we do still not have enough experience with this approach, and it remains an open area.

In conclusion, Dado is an approach to programming crosscutting concerns in distributed heterogeneous systems based on placing "adaptlets" at the points where the application interacts with the middleware. It supports heterogeneous implementation and triggering of adaptlets, allows client- and server- adaptlets to communicate in a type-checked environment using automated marshaling, provides flexibility in communication between adaptlets, allows flexible binding, and late deployment of adaptlets on to application objects.

Chapter 4

Examples

Chapter 3 presented an overview of the syntax and implementation of Dado. This chapter presents two illustrative examples to show the range of possible non-functional features that can be captured by Dado in a distributed heterogeneous setting.

4.1 Client Puzzle-Protocol

As Chapter 2 discussed, CPP defends against DoS (denial of service) attacks. The protocol works by intercepting client requests and refusing service until the client provides a solution to a mathematical problem. The time it takes to solve the problems is predictable; fresh problem instances are created for each request. The need to solve puzzles throttles back the client, preventing it from overloading the server. Typically the puzzle involves finding a collision in a hash function, e.g., finding an input string that hashes to a given n bit value modulo 2^m, for $n > m$. Such puzzles are very easy to generate and require about 2^m times as much effort to solve, given a collision-resistant hash function. Here, m is called the difficulty, size, or length of the puzzle. This explanation is made more concrete through Figure 4.1.

A server can generate a `PuzzleChallenge`, as in line 1. The `PuzzleChallenge` is delivered to a client through the `PuzzleException` (line 7) which encapsulates the

```
(1) struct PuzzleChallenge {
(2)    long puzzleID;
(3)    short puzzleLength;
(4)    Octet64 puzzlePreImage;
(5)    Octet16 puzzleHash;
(6) };

(7) exception PuzzleException {
(8)    PuzzleChallenge challenge;
(9) };

(10) struct PuzzleResponse {
(11)   long puzzleID;
(12)   Octet64 puzzleSolution;
(13) };
```

Figure 4.1: Client-Puzzle Protocol Messages

challenge. The server will not honor the client's invocation until it has solved the puzzle. It can provide a solution through the structure on line 10. Now, we discuss the rationale for this protocol.

By providing a solution to the puzzle a client has *proven* to the server that it has performed a certain computational task. In other words, the client has *expended* some amount of CPU resources. Pragmatically, no client can ever obtain an infinite amount of CPU resources. The CPP is designed so that legitimate clients expend a moderate amount of resources. However, to launch a DoS attack would require more resources than is practical. Three questions commonly arise in the context of this rationale.

1. *Why not simply authenticate clients to avoid interactions with malicious clients?* This is problematic for two reasons. First, the server may wish to operate in an open environment where anonymous clients should be able to obtain service. Second, authentication itself may be an expensive operation and thus subject to DoS.

2. *Why not impose a server-side throttle to prevent the server from being over-*

loaded? We assume client invocations would be placed in a FIFO queue. The server could reduce its CPU consumption by only servicing invocations from the queue at certain intervals. However, since a malicious client would have many more invocations on the queue, legitimate clients would essentially be denied service. This leads to the final question.

3. *Why not process the queue to distribute server processing evenly among the client's requests?* This brings us back to our first question. It may not be possible to determine the identity of clients. Criteria such as the client IP address is problematic because the IP address could be forged or hidden by a Network Address Translator.

By addressing these questions we hope to convince the reader of the suitability of the CPP protocol as a potential usage scenario for our Dado software.

4.1.1 CPP Adaptlets

```
(1) service CPP {
(2)    client {
(3)       around void catchPuzzle();
(4)    };
(5)    server {
(6)       around void CPP() raises PuzzleException;
(7)       context puzzleSolution(PuzzleResponse solution);
(8)    };
(9) };
```

Figure 4.2: Generic Cache Adaptlets

Figure 4.2 shows the DDL for the CPP adaptlets. The client side has one advice used to *bracket* the execution of the application invocation. Clients receive puzzle challenges through the exceptions that are caught by this code.

The server side includes an advice operation `CPP` (line 6) that should be bound to application operations protected by the CPP. The operation `puzzleSolution` allows clients who have been presented with a puzzle challenge to return their solution. This operation is tagged by the keyword `context`. A `context` operation is a special type of `request` message which we introduce now.

4.1.2 Context Operations

In some circumstances a service may require extra data to be shared by the client and server adaptlets. Previously, we described how this was achieved using a `request` message. When a `request` message arrived at a matched adaptlet, some operation was invoked that matched the signature of the message.

In many situations, the data will need to be accessed by advices in the adaptlet. For example, the puzzle solutions need to be accessed by the CPP advice. This is not directly handled by `request` messages. A naive solution would be to have the `request` operation save the value of the data in a field of the adaptlet object. Later, when advice executes it could inspect the data.

This is problematic since the adaptlet field is shared by concurrently executing threads. Rather than force developers to manage the thread/data relationship explicitly we provide the `context` message mechanism. Instead of being dispatched to a method site, context messages are stored in a Java `ThreadLocal`. The actual arguments of the context operation can be retrieved by an advice by invoking an operation of the same name on its matched adaptlet stub. This operation returns either true or false, indicating whether the message is available. The arguments (passed by reference) are set to the data provided in the message. We show how this is useful in the implementation of the CPP server adaptlet.

4.1.3 CPP Implementations

The implementation of the CPP is completely application agnostic. The only interaction with the application is through modification of the client/server control flow. Server responses are preceded by the issuing of a `PuzzleException` and subsequent verification of the `PuzzleResponse`. First, we present the client implementation where `PuzzleExceptions` are caught and used to construct the `PuzzleResponse`.

```
(1) CPP_ClientImpl implements CPP_Client {
(2)    void catchPuzzle(catchPuzzle_Proceed proceed) {
(3)       try {
(4)           proceed.proceed();
(5)       }
(6)       catch(PuzzleException pc) {
(7)           try {
(8)               PuzzleSolution ps;
(9)               ps.puzzleID = pc.challenge.puzzleID;
(10)              recursiveSolver(pc.challenge.length,
(11)                              pc.challenge.puzzlePreImage);
(12)              ps.puzzleSolution = pc.challenge.puzzlePreImage;
(13)              _that.puzzleSolution(ps);
(14)              proceed.proceed();
(15)          }
(16)          catch(PuzzleException pc2) {
(17)              throw new RemoteException("CPP Error");
(18)          }
(19)      }
(20)   }
(21)}
```

Figure 4.3: CPP_ClientImpl : Declaration of `_that` and `initialize` not shown.

The client implementation, Figure 4.3, consists of one operation `catchPuzzle`. This code is placed around invocations that may potentially raise a `PuzzleException`. In Dado, a special argument (shown on line 2) is provided to every `around` advice. It provides the same functionality as the `proceed` keyword in the AspectJ language, which invokes execution of the original application code. However, in Dado, no special language is used for the implementation of adaptlets. The `proceed` object encapsulates a reference to the next `AdviceProxy` on the list. When the function `proceed` is invoked, it indirectly causes the next `run` method to be invoked.

We see this in lines (3-5). An attempt is made to contact the server by `proceeding` with the invocation. In case an exception is raised, it is handled in lines 6-16.

First, the ID of the puzzle is copied (line 9) so that the server may correlate challenges and responses. We note that it is not feasible for a client to guess a solution by forging an ID. Second, a “helper” method, `recursiveSolver`, is called to solve the puzzle (line 10). This code is provided in the appendix. Once the puzzle is solved it is added to the invocation through the `puzzleSolution` context message (line 13). Once again the invocation is attempted on line 14. If failure occurs for the second time, the client adaptlet gives up and throws an exception to the application (line 17).

```
(1) CPP_ServerImpl implements CPP_Server {
(2)    void CPP(CPP_Proceed proceed) throws PuzzleException {
(3)       PuzzleResponseHolder response;
(4)       if(_that.puzzleSolution(response) && checkResponse(response)) {
(5)          proceed.proceed();
(6)       else {
(7)          PuzzleChallenge challenge = createChallenge(16);
(8)          throw new PuzzleException(challenge);
(9)       }
(10)   }
(11)}
```

Figure 4.4: CPP_ServerImpl : Declaration of `_that` and `initialize` not shown.

Now we turn to the CPP_Server implementation in Figure 4.4. Here also, only one operation is required to be implemented. The context operation is only a vehicle for communicating PuzzleResponses to the existing advice. The advice is declared on line 2. It must first check for an incoming response (line 4). If one is available, it must also verify its validity (also line 4) by a call to the method `checkResponse` (see appendix). At this point the invocation is allowed to `proceed`. Otherwise, a new challenge will be created (line 7) by calling `createChallenge` (also in the appendix). The challenge is delivered to the client through an exception. If client and server deployment have been properly coordinated, the advice on the client side will catch the exception. Otherwise, the exception will propagate to the application. One of our motivations for explicitly describing matched client and server adaptlets is to ensure this proper coordination.

4.1.4 RCT Implementation

In order to test the compatibility of our implemented Dado software, we set out to deploy an example on a realistic CORBA application. This was done to see if we had made some unrealistic assumptions in our design choices. Although a single case study cannot serve to completely validate our implementation, it has proven useful in understanding some CORBA use-cases that were not previously anticipated. For the remainder of this section we present our example deployment using the CPP adaptlets and the RCT application.

As was discussed in Section 2.2, the RCT is a client/server application running over the CORBA middleware. Since this software was developed by an outside third-party we felt it made a good candidate for our case-study. An initial problem involved the server-side software implemented in C++. This software utilized the Omni ORB, while our software was written for the TAO ORB. Omni ORB does not support the mechanisms we required for transparent wrapping of client and server side objects (through Smart Proxies and the Object Reference Factory). After an initial review of the RCT implementation we judged that it would be straightforward to "port" the software to the TAO ORB. This involved recompiling the RCT IDL using the TAO compiler and refactoring dependencies on the proprietary Omni API. At this time we became familiar with the implementation details of the RCT server software. On the client side we had very little difficulty. Our AspectJ implementation depends only on the details of the generated CORBA stubs and skeletons. For Java, the interface of this code is standardized by the CORBA Portable Stubs/Server specification.

Previously, we had implemented the generic CPP Java and C++ adaptlets. We hypothesized that some applications would require protection on *some* but not *all* of the operations made available to clients. In these cases pointcuts could be used by a deployment expert to seamlessly apply the CPP protocol across those operations.

After reviewing the server software we noticed that a subset of the operations across the RCT interfaces had code that interacted with the server-local relational database. We identified this as a *crosscutting concern*. Potentially, these operations would be highly CPU intensive. Each query to the database involves communication across processes and execution of the relational query engine. The other operations (those not part of the database concern) simply return some values stored in the CORBA objects.

This database concern that we identified serves as the basis for the example deployment of the CPP adaptlets.

CPP Deployment

```
service CPP_RCT : CPP {
   pointcut databaseConcern() :  !(call(* Server.*(..))  ||
                                   call(* PingServer.*(..)));
   client {
      around databaseConcern() :  catchPuzzle();
   };
   server {
      around databaseConcern() :  CPP();
   };
};
```

Figure 4.5: CPP_RCT

As shown in Figure 4.5, the pointcut `databaseConcern` is declared inside the `service` scope and shared by both the client and server adaptlets. This allows coordination of advice that throws puzzle exceptions and advice that catches puzzle exceptions. The pointcut matches all 76 operations in the interfaces of the RCT application except those declared in either the `Server` or `PingServer` interface.

It is often claimed that a pointcut, such as the one shown here, is more concisely able to capture the *intention* of the developer modifying an application. Here, the databaseConcern is captured and ameliorated by applying the CPP protocol.

4.1.5 CPP Measurements

To measure the end-to-end overhead of our generated code, we compare our adaptlet tool modified RCT application with a hand-modified RCT application. We measure the round-trip latency for a synchronous remote procedure call for eight different application configurations. Measurements were performed in two different environments. The first environment was a single 1.6Ghz Pentium M laptop computer with 1GB of main memory running both client and server processes in Windows XP Professional. The second environment placed the client process on a 2Ghz Pentium IV computer running Linux at UCDavis (smf.cs.ucdavis.edu) and the server process on the aforementioned laptop computer at the University of British Columbia. First we present the details and measurements for the traditional application configurations where no tool support is used. Then we compare these results with the configuration and measurements of the adaptlet modified application.

The baseline "App" configuration measures the round-trip latency for a call to the `get_id_from_user_alias` function on the RCT server. This was chosen as a representative application function for convenience because its execution presumes no particular application state. We hand-modified the original Java RCT 1.0 client application to take timing measurements using the `java.lang.System.currentTimeMillis()` function before and after 10,000 calls in the local environment and 100 calls in the Internet environment. The values presented are the average over the 10,000 (100) calls in milliseconds. We performed ten trials of the 10,000 (100) calls and took the average of each measurement. The standard deviation for the local measurements was quite small; for example, .0014 in measurement 1. In the case of the Internet measurements, the deviation was more significant; for example, 4.19 in measurement 3. These measurements were performed with and without the original server function body present. The server code is the RCT 1.0 application running on the TAO CORBA ORB compiled under Microsoft Visual Studio for C++. Measurements

Experiment	**RPC latency (ms) local**	**RCP latency (ms) Internet**
1. App(null)	.1274	77.40
2. App(null) with CPP(null)	.2467	148.6
3. App with CPP(null)	1.207	151.6
4. App with CPP	132.3	349.4

Figure 4.6: Traditional Application Measurements

where the function body was removed are labeled as "App(null)" and are presented to see just the overhead of communication through the middleware.

In the "App with CPP" configurations we implemented the CPP protocol over the original application by adding a new application function, `get_challenge`, and modifying the signature of the `get_id_from_user_alias` function. A call to `get_challenge` is made before each call to the original functions to obtain a `PuzzleChallenge`. The signature of the application function was widened to include a parameter for the `PuzzleResponse`. We include measurements where the `PuzzleChallenge` generation and `PuzzleResponse` solving code are removed in order to show just the communication costs. These measurements are labeled as "CPP(null)".

By comparing measurements 1 and 2 in Figure 4.6 we see that the overhead for CPP communication is about 93% (local) and 92% (Internet) . Recall that the "App with CPP" configurations include an additional function call so two RPC calls are required instead of one. Comparison of local measurements 2 and 3 shows the cost for the actual `get_id_from_user_alias` function, .9603ms. Measurement 4 includes the cost for the application and CPP protocol functions. The CPP generation and solving functions add two orders of magnitude to the local end-to-end latency (131.3ms) and more than doubles the latency over the Internet.

Four more measurements were made in order to see the overhead from our tool generated code. The first measurement shows the cost of 10,000 (100) RPC calls hand-modified to use the PortableInterceptor API [60] to piggyback a `PuzzleResponse` to

Experiment	**RPC latency (ms) local**	**RPC latency (ms) Internet**
5. App(null) with Interceptors	.3398	77.64
6. App(null) with CPP Adaptlets(null)	.5419	150.9
7. App with CPP Adaptlets(null)	1.641	154.5
8. App with CPP Adaptlets	133.2	349.6

Figure 4.7: CPP Adaptlet Measurements

the client request and a `PuzzleChallenge` to the server response. Our generated code uses these PortableInterceptor API calls to package adaptlet `request` and `context` messages. This measurement was performed to see what percentage of overhead is due to the use of this API. This measurement is labeled as "App(null) with Interceptors".

The final three measurements apply the CPP adaptlets previously presented in this chapter automatically to the RCT application using our tool support. We include measurements where the adaptlets simply pass dummy challenges and responses, "CPP Adaptlets(null)", in order to highlight the cost of the generated code.

A comparison of local measurements 1 from Figure 4.6 and 5 from Figure 4.7 reveals the overhead due to the use of the PortableInterceptor API. As was detailed in Chapter 3 this cost can be high in a local setting, accounting for a 166% overhead or .2124ms. By factoring this cost from measurement 6 we are left with .3295ms which compared to measurement 2 gives us a 33% overhead for our remaining generated code in the local setting. Factoring the PortableInterceptor API out of measurements 7 and 8 gives us an overhead of 18% and .004% respectively when compared to measurements 3 and 4.

In the Internet setting a comparison of measurements 1 through 4 to measurements 5 through 8 shows a smaller overhead. For example, comparing measurement 3 to 7 gives a 2% total overhead. This variation falls within the range of the standard deviation for measurement 3 (4.19ms) and so is fairly insignificant.

As anecdotal evidence to the performance overhead of our tool we compared the perceived responsiveness of the Java client GUI program in a sample user session. First, we compared execution of a session in local configurations 3 and 7. The sample session consisted of logging in, checking the user's class enrollment, and logging out. Throughout this session we could perceive no difference in the GUI responsiveness. Next, we executed the session for local configuration 8. In this case some sluggishness in the response was noticeable. This can be attributed completely to the overhead of the client-puzzle protocol. Therefore, we would recommend protecting only certain operations such as initial authentication or operations requiring anonymous access.

Here we conclude with some remarks about the performance of our current tools. Our dependence on the PortableInterceptor API has an obvious drawback in terms of performance. We made the decision to use this approach because it does not require tightly integrating our tools with a particular CORBA middleware. Tighter integration could probably improve performance at the cost of portability. Even still, there is a significant performance overhead for local measurements 6 and 7. Thus we conclude that our current implementation may not be suitable for many high performance embedded applications where communication and application latency is low.

This concludes the client puzzle-protocol example, which concerns a security non-functional feature. We now present an additional example, this time concerning client-side caching.

4.2 Client-Side Cache

We present a caching example, where a client requests some key indexed values (e.g., stock quotes) from a server. These values are assumed to periodically change, as the server receives updates. This example illustrates the use of parametric polymorphism [9] in Dado, and to our knowledge is a rather unusual use of parametric polymorphism in a modeling language. In the example, type parameters are shown in italic font.

Here, a client-side adaptlet maintains a cache of values. If a request can be answered from the cache, the server is never contacted. To complicate matters slightly, we also assume that the client-side adaptlet can receive from the server-side a time-to-live (TTL) value. The server-side adaptlet maintains TTL values for various keys by intercepting updates and observing update frequency for various keys (i.e., some values may be updated more frequently than others).

Here we illustrate another feature of the DDL language: parametric polymorphism, as in C++ type templates or Java generics. Type parameters can be used in three places: on the service scope, the client interface, or the server interface. The first of these is rather different than the latter two and we discuss it first.

Type parameters can be attached to the definition of a `service` using the syntax for Java 1.5 generics. This ensures that some data of the advice and requests of both the client and server are type compatible. Type parameters on the `service` scope must be instantiated at the DDL level before a `service` can be deployed. In the generated code, a template class is generated that encloses the generated client and server pure virtual base classes (interfaces). These interfaces are *nested* C++ classes.

Parameterization may also occur for the client or server interfaces. This is useful for the specification of generic adaptlets where type instantiation is a purely local phenomenon. Our tools generate a pure virtual base class (i.e., interface) in C++. This class can be implemented using generic types and later extended when the concrete types can be determined by a particular deployment scenario. In the example we highlight both the case of `service` and adaptlet parameterization.

We start with an abstract interface to the client-side and server-side adaptlets (Figure 4.8) necessary to carry out the caching service. The client-side includes a

```
(1)service Cache<KeyT> {
(2) client<ValueT> {
(3)   context set_TTL(long ttl);
(4)   around ValueT get(KeyT key);
(5) };
(6) server{
(7)   request ask_for_TTL(KeyT key);
(8)   void update_TTL(sequence<KeyT> values);
(9) };
(9)};
```

Figure 4.8: Generic Cache Adaplets

context operation (line 3) to receive TTL values that are "piggybacked" on server responses. Additionally, an advice (line 4) is specified to short circuit client requests for which cached data is available. The cache is keyed through application data of type *KeyT*.

The server-side has one request and one advice operation. The request `ask_for_TTL` (line 7) allows the client-side adaptlet to signal the server-side that it is planning to cache the data in the current invocation response. The advice `update_TTL` (line 8) should be triggered whenever application events occur on the server-side that invalidate cache data. This allows the server-side to heuristically adjust its forecasted TTL values. If application data cannot be used directly to key the cache, a deployment specific adapter function can be applied. Both the client-side and server-side adaptlets can have a polymorphic implementation given a suitable target language mapping (currently there is support for C++ templates).

4.2.1 Cache_Client Generated Code and Implementation

Chapter 3 showed how adaptlet interfaces were used to generate interfaces for a programming language and stubs for communication code. In particular, the discussion for Figure 3.4 discussed how these interfaces were generated. Figure 3.11 presented the code for the generated stub.

```
(1) template <class KeyT>
(2) class Cache {
(3)    template <class ValueT>
(4)    class Cache_Client {
(5)       virtual void initialize(Cache_Server_Stub that) = 0;
(6)       virtual ValueT get(KeyT key) = 0;
(7)    };
(8)    class Cache_Server_Stub {
(9)       void ask_for_TTL(KeyT key) {
(10)         //Marshaling Code
(11)      }
(12)   };
(13)};
```

Figure 4.9: Cache_Client Interface and Cache_Server_Stub

Now, we reiterate this discussion for the new case where the client interface and server stub must agree on some type parameters. The generated code is presented in Figure 4.9. Here both the client interface (lines 3-7) and server stub (lines 8-12) share a common type scope. The scope is implemented as an enclosing template class `Cache` (the entire figure) with the parameter *KeyT*. This common scope is used to ensure compatibility between the key type for the client side value cache and the key type for the server side TTL cache.

Figure 4.10 shows the implementation for the generic client-side cache. This class implements the Cache_Client interface from Figure 4.7. In lines 1 and 2, the free type variables are captured by the class template. These type variables are used in four places in the code.

First, in line 3, a server stub reference is declared that is coordinated through the *KeyT* type. This reference would be set through the `initialize` method (not shown). In line 4, the actual "cache" is declared as a Hash_Map with values of type ValueT and keys of type *KeyT*. Recall that the parameter ValueT is local to the client-side adaptlet. In line 5 another map is declared for recording TTL values. This will allow us to determine if a cached value has become "stale". Finally, on line 6 the `get` operation is implemented. Its signature mimics that required for the actual

```
(1) template <class KeyT, class ValueT>
(2) class Cache_ClientImpl :  Cache<KeyT>::Cache_Client<ValueT> {
(3)    Cache<KeyT>::Cache_Server_Stub _that;
(4)    Hash_Map<KeyT, ValueT> cache;
(5)    Hash_Map<KeyT, long> ttls;
(6)    ValueT get(KeyT key, get_Proceed proceed) {
(7)           long ttl;
(8)           ttls.find(key,&ttl);
(9)           if(ttl < currentTimeMillis()) {
(10)             ValueT cached;
(11)             cache.find(key,&cached);
(12)             return cached;
(13)          }
(14)          else {
(15)             _that.ask_for_TTL(key);
(16)             ValueT returned = proceed.proceed(key);
(17)             long ttlReturned;
(18)             if(_that.set_TTL(&ttlReturned)) {
(19)                ttls.bind(key,ttlReturned);
(20)                cache.bind(key,returned);
(21)             }
(22)          return returned;
(23)          }
(24)   }
(25)};
```

Figure 4.10: Cache_ClientImpl

cache (line 4). In addition, a `proceed` object is made available for short-circuiting the client invocation.

The logic for the `get` method consists of two cases. Either the value for the key has not expired, lines 9-13, or it has, lines 14-23. In the first case, the value is retrieved from the cache and returned (i.e., the server is not contacted). Otherwise, an `ask_for_TTL` request is queued (line 15) and the invocation is allowed to proceed (line 16). After a value has been returned, the `set_TTL` operation is queried for a new TTL value (line 18). Finally, the cache and TTL values are updated (lines 19-20).

```
(1) template <class KeyT>
(2) class Cache_ServerImpl :  Cache<KeyT>::Cache_Server {
(3)    Cache<KeyT>::Cache_Client_Stub _that;
(4)    Hash_Map<KeyT, long> last;
(5)    Hash_Map<KeyT, long> ttls;
(6)    void ask_for_TTL(in KeyT key) {
(7)       long ttl;
(8)       ttls.bind(key,&ttl);
(9)       _that.set_TTL(key,&ttl);
(10)   }
(11)   void update_TTL(Sequence<KeyT> values) {
(12)      for(int i=0;i<values.length();i++) {
(13)         long lastUpdate;
(14)         last.bind(values[i],&lastUpdate);
(15)         ttls.bind(values[i],currentTimeMillis()-lastUpdate);
(16)      }
(17)   }
(18)}
```

Figure 4.11: Cache_ServerImpl

4.2.2 Cache_Server Implementation

On the server-side, Figure 4.11, TTL values for each key are being tracked and reported back to the client. This is achieved through the `ask_for_TTL` request and the `update_TTL` advice.

The code for the `request ask_for_TTL` is shown on lines 6-10. This method is dispatched whenever an invocation arrives at the server with an `ask_for_TTL` message attached to it. It simply retrieves the current TTL value for the key (line 8) and responds by attaching a `set_TTL` context to the invocation response.

The code on lines 11-17 tracks updates and modifies the TTL values. An update consists of a sequence of keys that are to be refreshed. TTL values are adjusted heuristically by calculating the time difference from the last update to the current time (line 15). This provides an estimated TTL (by the adaptlet) in situations where TTL values have not been explicitly guaranteed by the application itself.

Now, deployment can proceed separately on the client-side and server-side. First, we present client-side deployment.

4.2.3 Client-Side Deployment

```
service StockCache : Cache<string> {
  client : client<float> {
    around call(float StockServer.getQuote(string x)):
      get(x);
  };
};
```

Figure 4.12: Deploying the cache for the StockServer client

Figure 4.12 shows the instantiation of the Cache for the StockServer client. To cache the result of the `getQuote` operation the client-side cache type parameter *ValueT* is instantiated as float. The pointcut matches the invocation to `getQuote` and captures the argument to key the cache. Finally this pointcut is applied to the advice, `get`, that we want to intercept these invocations.

```
class StockCache_ClientImpl : StockCache_Client,
                              Cache_ClientImpl<string,float>
{ };
```

Figure 4.13: Instantiation of concrete class for client Cache adaptlet

This final snippet of C++ code, Figure 4.13, is needed simply to instantiate the type parameters for the generic Cache_ClientImpl and ensure compatibility with the generated interface for the StockCache_Client sub-interface.

4.2.4 Server-Side Deployment

The server-side deployment, Figure 4.14, consists of dispatching the advice `update_TTL` for operations where TTL calculation can be updated. The pointcut matches the operation `StockServer.setQuotes` and is applied so `update_TTL` will be dispatched before those invocations. Here, the application data-structure QuoteList

```
service StockCache : Cache<string> {
  server{
    StringSeq hashCodes(in QuoteList quotes);
    before call(* StockServer.setQuotes(ql)):
      update_TTL(hashCodes(ql));
  };
};
```

Figure 4.14: Deploying the cache for the Stock server

does not match the expected structure of the instantiated sequence of string. The function hashCodes can be used as an adapter to convert the QuoteList, `ql`, captured by the pointcut into the correct format.

```
class StockCache_ServerImpl : StockCache_Server,
                              Cache_ServerImpl {
   StringSeq hashCodes(in QuoteList quotes) {
      //Copy key string from each Quote into a new list of strings
   }
};
```

Figure 4.15: Instantiation of concrete class for server Cache adaptlet

Similarly to the client deployment scenario a new concrete class, Figure 4.15, must be instantiated. In this case the deployment expert has more work than simply applying some pointcuts. Since it is desirable to reuse the existing adaptlet implementation, a simple type adapter function is implemented. The code for the adapter is straightforward.

4.2.5 Cache Example Conclusion

In this chapter we demonstrated the usefulness of matched client-server adaptlet through the presentation of caching with TTL values. This was done to highlight an additional contribution of our Dado methodology and as a study into the usefulness of aspect-oriented programming.

Throughout the thesis we have claimed that many features in client/server application require modifications to both the client and server. Thus, we proposed the Dado framework for matched client/server adaptlets. Here we presented the usefulness of the matching when some features can be implemented generically. The DDL language extension allowed the coordination of type parameters between the adaptlets.

Also in this chapter we identified the crosscutting concern of TTL caching. This not only spanned the client and server but also crosscut the structure of the server interface. Modifications were required at both the “get” and “set” operations in order to provide and update the TTL. This shows how Dado expands the notion of aspect-oriented development into the specific domain of client/server applications.

Chapter 5

GlueQoS

Extending component interfaces directly with information about non-functional concerns limits the reusability of an interface. Each component implementing the interface must be prepared to handle these concerns appropriately. Furthermore, it also limits customizability, for example, the ability of local security officers to tailor the policies to suit their settings. The previous chapter presented a new approach to the development of software that adds support for such non-functional concerns.

With this approach, components only implement a functional interface; QoS features such as security are left unresolved until deployment time. A specification, written by a deployment expert, can be used to transform the original components through the use of aspect-oriented programming. As presented, the approach is server-centric, and does not consider the issue of matching client-side QoS features to the deployment policy on the server.

This inflexibility and "server-centricity" limits the use of the approach in new, emerging application areas such as Service-Oriented [99] computing and Peer-to-Peer (P2P) [93] computing. Here, we need a highly dynamic, and symmetric (*not* server-centric) way of managing end-to-end QoS requirements. In these settings, components, deployed as autonomous software processes, create and manage relationships with other processes dynamically; processes can play either client or server role. Processes can exist in different administrative domains, with different deployment contexts (which may also change dynamically), and thus have different QoS requirements. This chapter presents an approach to provide *dynamic and symmetric reconciliation* between the (potentially different) QoS features of two communicating processes. However, different QoS features can interact in various ways, and this complicates reconciliation.

We use the term *feature interaction* [102] to reflect how feature combinations affect each feature's ability to function as it would separately. Feature interactions can be complex, subtle, and very difficult to identify. Finding such interactions is outside the scope of this thesis. In addition, feature preferences are a matter of deployment policy and can vary.

Our work assumes each feature is universally named and potentially configured through *feature parameters*. Realistically this may only be possible through standardization of QoS features such as in the web services setting. Our contribution is a resolution and deployment middleware. The middleware supports the dynamic management of QoS features between two components, in a WAN setting, that encounter each other for the first time.

We provide a declarative language, GlueQoS, for specifying the QoS feature *policies*. The middleware-based resolution mechanism uses these specifications to dynamically find a satisfying set of QoS features that allow a pair of components to inter-operate. The basic language semantics are based on *propositional logic* [37]. However, in order to reason over feature parameters with numerical values, we base our implementation on *mixed-integer programming*.

Note on formal presentation In our formal presentation, concepts are defined in terms of mathematical *sets* and *tuples*. Sets contain objects of a single *type* (i.e., they are homogeneous). This is denoted $Set\langle TypeName\rangle$. The operator += adds a single element to a set. An iterator over sets is defined by the `foreach` construct.

A tuple is defined as a heterogeneous set of fixed size. Each object in a tuple is named and has a corresponding type; this is analogous to a `struct`. The "dot" notation is used as a projection function to access the named objects.

The remainder of this chapter is organized as follows: Section 5.1 presents the desired elements of policies, Section 5.2 presents the concrete language syntax, Section 5.3 describes how policy matching is implemented, Section 5.4 provides a description of policy exchange between client and server, and finally Section 5.5 ends with some conclusions.

5.1 Motivation and Informal Semantics

The propositional logic provides a formal basis for reasoning about binary relationships between abstract objects. An object can either be an *atomic variable occurrence* or a *sentence*. Sentences are are composed of atomic variable occurrences and other sentences. Composition is achieved through the familiar logical operators (propositional connectives) ($\wedge$,$\vee$,( )). The famous satisfiability problem, 3-SAT, is based on propositional logic. We use the term "variable occurrence" to distinguish between an instance of a variable (which may appear *negated*) and the object itself. Variables are represented by the type Var.

In GlueQoS, policies are expressed as propositional sentences. The atomic variables stand for two types of statements called *atomic statements* (see below). Atomic statements contain no propositional connectives.

Atomic Statements (A):

- Feature Status, a statement that a particular feature is enabled.

- Parameter Constraint, a restriction on the values that the parameters of some features may take.

In propositional logic, the truth of a sentence is considered in the context of some choice of truth (true or false) for all the variables, called a valuation $\mathcal{V}$. A valuation that satisfies a sentence is called a *model*, $\mathcal{M}$. Likewise, a GlueQoS policy is considered to be satisfied or violated based on choices of feature status and feature parameters. We want to find a model for both the client and server policies.

A model for a sentence in propositional logic can be found through the process of *resolution* [22]. The algorithm requires that a sentence is expressed in *conjunctive normal form* (CNF). A sentence in CNF requires that *clauses* are joined by *conjunction* (i.e., "and"). Then, each clause consists of one or more variable occurrences joined by *disjunction* (i.e., "or"). Another important form is the *disjunctive normal form* (DNF). The DNF is built of conjunctive clauses joined by disjunction.

The interpretation of policies as propositional sentences is straightforward for feature status but parameter constraints complicate matters considerably. First, we consider this simple case where feature status is mapped to boolean variables.

5.1.1 Propositional Interpretation

Instances (as in object-oriented programming) of features are managed by the GlueQoS middleware. Each client-server *session* is associated with a set of feature instances. If a feature is in this set, we say the feature's *status* is *on* (true) for that session; otherwise, the status is *off* (false).

Clients and servers must have implementation code available with which they can create an instance. An implementation could, for example, take the form of an adaptlet.

For a given GlueQoS policy, the models of that policy are called the *acceptable feature configurations*. A model tells us which features should be on or off. Rather than be forced to enumerate models explicitly, a user can express a policy as the conjunction of *configuration requirements*. Here we list some possible configuration requirements for the special case of two features (in general requirements may contain any number of features):

- Dependency: The deployment of a feature, A, depends on the deployment of another feature, B. This can be expressed as an implication $(\bar{A} \vee B)$.
- Conflict: Two features conflict if their combination has a negative effect on the behavior of the entire application. The deployment of one feature should exclude the deployment of the other. The decision that an effect is negative is application dependent but may include effects such as introducing deadlock, putting data in inconsistent states, or degrading performance. This conflict is represented by, $(\bar{A} \vee \bar{B})$.
- Choice: Either feature or both can be chosen to meet some requirement. This is expressed as $(A \vee B)$.
- Exclusive Choice: Two features are equivalent for the satisfaction of some requirement. An exclusive choice must be made as to which feature should be instantiated. This is expressed by combining choice and conflict: $(A \vee B) \wedge (\bar{A} \vee \bar{B})$.

We stated that a model for propositional sentences could be found by resolution. However, given the policy of both a client and a server, our policy resolution[1] problem is to find a model which satisfies both.

Our client/server scenario must account for the fact that client and server policies are written in isolation. Therefore, the sets of variables mentioned in each policy might not be the same. For example, given the single requirement policies of a client, $Policy_C$, and a server, $Policy_S$,

$$Policy_C = (A \vee B)$$

[1]We note that our use of "resolution" used for the remainder of this section is in analogy to propositional resolution but not technically related.

$$Policy_S = (A \wedge C)$$

is there a model which satisfies both?

Before the question can be answered, it must be agreed on what variables can be included in a model. We call this agreement the client/server *feature set*. Policy developers should not be expected to declare their requirements over any possible feature. For example, in the above case, it would not be safe to assume that the client deployment environment included an implementation of feature C. We define the function *vars*[2] that returns the set of feature status variables mentioned in a policy.

Definition 5.1. The client/server feature set, S, is defined as $vars(Policy_C) \cap vars(Policy_S)$.

Definition 5.2. The basic client/server policy resolution problem is to find a model for $Policy_C \wedge Policy_S$ such that $\forall v \notin S = false$.

So we see, any feature not in the feature set cannot be turned on for the session. In the example, the server cannot meet the requirement $(A \wedge C)$ because C is not in the feature set. This is because $vars(Policy_C) = \{A, B\}$ and $vars(Policy_S) = \{A, C\}$. While the intersection is just $\{A\}$, the server policy requires both A and C. This leads us to our definition of client/server policy resolution. This straightforward interpretation allows us to continue to view the problem as propositional resolution.

5.1.2 Linear Constraints

Now that we have addressed the motivation and interpretation for policies regarding acceptable feature configurations, we turn to the matter of feature parameter constraints.

[2]We omit the definition of *vars*, which can be defined by a structural induction on the policy.

Every instance of a feature may be configured according to a set of parameters. This is analogous to Component Oriented Programming [88]. For example, in the Java Beans component model every bean may expose a set of attributes for deployment time configuration. However, in our scenario we must allow for joint agreement, between client and server, of the session-time feature parameters.

For this purpose, we allow the representation of *linear constraints* [78, 20] over feature parameters. In this thesis we limit our discussion of feature parameters to values in $\Re$ (the reals).

Definition 5.3. A *linear constraint*, L, is a triple $(Components, Op, Rh)$.

- $Components$ is a set of $Component \equiv (Var, Coefficient)$, where $Coefficient \in \Re$.
- $Op \in (\geq, =, \leq)$[3] (i.e., the *Op*erator)
- $Rh \in \Re$ (i.e., the *R*ight-*h*and side)

Example The linear constraint,
$(\{(PayFeature.price, -2.0), (QoSMonitor.guarantee, 1.0)\}, =, -100.0)$
is interpreted as the mathematical equality,

$$-2.0 * PayFeature.price + 1.0 * QoSMonitor.guarantee = -100.0$$

Graphically, this allows clients and servers to negotiate a choice of the two feature parameters anywhere along the line defined by the equation. We will often refer to the expression involving the variables as the "left-hand side"; the constant (-100.0 in the example) will be called the "right-hand side".

A *linear program* consists of a set of linear constraints and a linear expression (the utility) to be maximized. The utility assigns a weight to each variable. Thus, a solution to a linear program consists of a model that not only satisfies the constraints

[3]Due to limitations in our implementation technology the operators $<$ and $>$ are not supported.

but also maximizes the utility. Modern solvers are usually based on the *Simplex* [20] algorithm due to Dantzig.

In our policy language we would like to support the combination of propositional sentences and linear constraints, using propositional connectives. Intuitively, this combination allows one to express: linear arithmetic relationships between feature parameters (constraints), logical relationships between constraints, and logical relationships between constraints and feature status.

Now, we must look beyond solution strategies based on the Davis, Putnam, Lovelace algorithm in order to account for arithmetic relationships. Section 5.3 shows our implementation based on leveraging an extension of Linear Programming, called *Mixed Integer Programming* [20].

5.1.3 Run-time Policy Adaptation

Recall that clients and servers execute in an environment that is continuously changing; they might need to be configured according to a dynamic deployment context. Rather than force deployment experts to constantly update policies manually, our policy language includes constructs to reflect these environmental changes. The constructs are of two types: user-defined value functions and user-defined predicate functions.

The values of coefficients or constants in linear constraints can be input through user-defined value functions. Evaluation of these functions occurs periodically throughout the execution of client and server applications. Before policy resolution occurs, a "snapshot" of the client and server policies is taken to reflect their current states. The implementation details are discussed further in Section 5.4. For example, we can update the example given as,

$$-2.0 * PayFeature.price + 1.0 * QoSMonitor.guarantee =$$
$$\text{``}CPULoad() * 100.0 - 100.0\text{''}$$

Graphically, this allows the expression of a line which is shifted vertically based on the current value of the user-defined function `CPULoad`.

Likewise, requirement of a particular feature in an acceptable configuration set may also depend on the state of the execution environment. A security feature, for example, may only be required for certain types of network connections e.g.,

Password and (Encryption when "linkType(mobile)").

Here, the required configuration varies between using the Password feature alone and using both the Password and Encryption feature. This variation is based on evaluation of the `linkType` user-defined predicate.

We have shown that the acceptable feature configurations may vary dynamically. The actual policies expressed depend on the moment when resolution occurs. We have provided two constructs in the our language to express this variation.

5.2 Policy Language

Policies are specified in the GlueQoS policy language. The design goals of this language are to provide a declarative, expressive means of describing QoS features, their interactions, and their sensitivity to operating conditions. The language provides a set of built in operators to specify acceptable feature configurations, as well as the ability to extend the system with functions to measure operating conditions (such as load, available energy, and bandwidth).

As described in the previous section, feature status variables and parameter constraints could be combined freely according to the propositional logic. However in the GlueQoS policy language, policies are restricted to a particular form to accommodate the underlying resolution algorithm. This form encompasses both CNF and DNF; it is based on the conjunction of DNF sentences. Here we provide a rationale for this form, in terms of a possible usage pattern.

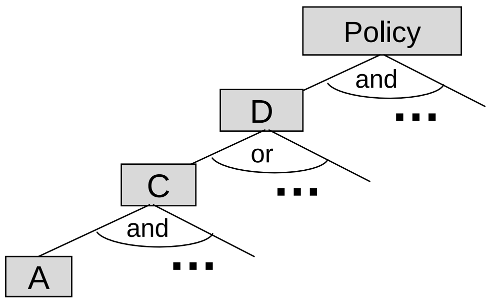

Figure 5.1: *Policy Hierarchy* : Policies consists of a set of **D**isjunctions representing the policy requirements. Each requirement is built from the **C**onjunction of some **A**tomic statements.

Policies are constructed as a hierarchy of four levels as shown in Figure 5.1. At the root of the hierarchy is the policy itself, followed by requirements (**D**isjunctions), configurations (**C**onjunctions), and ending in the atomic statements(**A**tomic). Considering the EBNF in Figure 5.2, we describe these levels in more detail.

A GlueQoS policy (line 1) consists of the conjunction of several requirements. Each requirement (line 2) is either a disjunctive clause, a *preference* (described below), or a *supports* clause (described below). A disjunction allows a requirement to consist of dependencies, conflicts, or choices between *configurations.* A configuration (line 5) is a conjunction of atomic statements. All the atomic statements in a configuration must be satisfied in order to satisfy the configuration. An atomic statement (line 6) is either a feature status variable (line 7) or a parameter constraint (line 8).

Definition 5.4. A *Policy* is a four-tuple $(Vars, \mathit{Parms}, \mathit{Req}, Prefs)$.

- $Vars$ is a set of variables considered as feature status variables
- Parms is a set of variables considered as feature parameters
- Req is a set of requirements (disjunctions)

- *Prefs* is an optional ordered set of variables (Preferences, described in Section 5.2.1)

Atomic statements can be qualified through the use of user-defined functions. A feature status variable can be appended by a "when" clause (in line 6). The when clause consists of a quoted Java expression of type `boolean`. Parameter constraints can also be decorated with user-defined functions. This allows the coefficients (line 12) or right-hand sides (line 8) of the linear constraints to vary over time.

```
(1) PolicyClause::  Requirement ("and" Requirement)*
(2) Requirement::   "(" Config ("or" Config)* ")" | Preference | Supports
(3) Preference::    "Preference" "(" Feature* ")"
(4) Supports::      "Supports" "(" Feature* ")"
(5) Config::        "(" AtomicStmt ("and" AtomicStmt) ")"
(6) AtomicStmt::    Feature ("when" JavaExpr)?  | Constraint
(7) Feature::       FeatureName | "not(" FeatureName ")"
(8) Constraint::    Operand ("+" Operand)* Operator ValueExpr
(9) FeatureName::   uppercase_id
(10)ParamName::     lowercase_id
(11)Operator::      "=" | ">=" | "<="
(12)Operand::       ValueExpr "*" FeatureName "." ParamName
(13)ValueExpr::     float | JavaExpr
(14)JavaExpr::      string
```

Figure 5.2: GlueQoS Language EBNF

5.2.1 Special Requirements

In addition to the language elements we have laid out so far, two special (non-disjunction) types of requirements are supported. These are the `Supports` and `Preference` clause.

Section 5.1.1 described the necessity for clients and servers to agree on the client/server feature set. This was motivated by the fact that policies cannot be expected to mention the acceptable status for any possible feature. We chose the semantics that any feature not mentioned would be assumed to have its status forced to off. So in order to add a feature to the feature set, a policy should mention a feature in one of its requirements. The `Supports` clause allows a policy to express that some feature can be included in the feature set without making any restriction on its status. For example, the clause `Supports(CPP)` can be interpreted as adding the tautology $(CPP \vee \overline{CPP})$ to the set of policy requirements.

Assuming the client is given some choices (as in Section 5.1.1) between features to meet a particular server requirement, the `Preference` clause provides a way to instruct the resolution algorithm which feature to choose. Optimization methods based on linear programming allow for resolution of a model which maximizes some utility function over the constraint variables. Leveraging this utility function we can support preferences over the configuration of features from any possible acceptable feature configuration.

In the following section we demonstrate the possible usage of the language elements laid out in this section.

5.2.2 Security Revisited

Figure 5.3 is a realization of the security policies from Chapter 2 as expressed in GlueQoS. The first policy is shown for the server.

Server:
```
(1) (not(Authentication) or (CPP.size = "GQ.CPUUsage()*8")) and
(2) (CPP when "GQ.CPUUsage() > .5") and
(3) (Authentication or (CPP.size = "GQ.CPUUsage()*16"))
```

Client1:
```
(4) Authentication
```

Client2:
```
(5) Supports(CPP,Authentication) and
(6) Preference(not(Authentication),Authentication) and
(7) (CPP.size <= 4)
```

Figure 5.3: Security Example

Each line (1, 2, and 3) represents a different server requirement. The first is an implication between the status of the Authentication feature and a constraint on the size parameter of the CPP feature. It states that with Authentication, the size of puzzles varies linearly from 0 to 8 depending on CPU load. Another requirement (line 2) uses a predicate `(GQ.CPUUsage() > .5)` to determine whether CPP is required. When CPU load is less than .5, the server allows Authentication to be used without the CPP; otherwise just CPP, with the largest puzzle size, can be used. This shows how runtime conditions can dynamically adapt the acceptable feature combinations expressed by hosts.

The first client policy is shown on line 4. This client will only use the Authentication feature (perhaps because of software availability, or because it is too performance-limited for CPP). Therefore, this client can only create a session with the server when the server's load is less than 0.5.

The second client policy (lines 5-7) uses parameter constraints to choose between two feature combinations. Recall that the `Preference` semantics in our language denotes a preference for the first alternative. Consider a situation where this client wishes to maintain its anonymity by not using the Authentication feature. However, it also has a performance requirement that takes precedence. Perhaps the client is on a mobile device with low computing power. Line 6 expresses the client's preference to maintain anonymity. However, in order to keep performance at a certain threshold the client will also use Authentication if it will keep the puzzle size low. By comparing to the sample server's policy (lines 1 and 3 in particular): if this client contacts the server when the server's CPU load is 25 percent or lower the client can maintain its anonymity by using CPP only (from line 3 and 7, $16 * .25 <= 4$). However, if it contacts the server and the server's CPU load is between 25 percent and 50 percent it will agree to reveal its identity to maintain higher performance (from line 1 and 7, $8 * .5 <= 4$). When the server's load passes 50 percent the client will be unable to mediate a satisfactory feature composition with the server.

5.3 Mixed Integer Programming Policy Matching

In the previous section, we described our policy language for expressing acceptable feature configurations. Prior to that description, we laid out the problem of finding a configuration that satisfies both the client and server policies. We have made some informed design decisions and arrived at an implementation based on mixed integer programming. Pragmatically, the best choice for these decisions would be based on best practices observed over a number of years. This is beyond the scope of our thesis.

Mixed Integer Programming has been used widely in the area of Operations Research [20] for decades. Here we apply this technique for automating the configuration management[4] of aspect-oriented software in a client/server setting.

Mixed Integer Programming extends the theory of linear programming. In a mixed integer program (MIP) a subset of variables can be constrained to integer values. Hence, the "mixed" denotation refers to a mix of real and integer variables. A

[4]A more thorough treatment would include management for versioning of features. In this thesis we focus only on management of interactions between features induced by non-functional requirements.

popular strategy for solving a MIP is based on the Branch-and-Bound [78] algorithm. Implementations of these algorithms are widely available as commercial [49] or open source [50] packages. These algorithms are *complete*, a solution can be found if one exists.

In this thesis we view the MIP algorithm as a black-box that is utilized for the purpose of resolving policies. This can be approached by modeling boolean variables as 0/1 integers and feature parameters as real variables. Our task in this section is to describe how a GlueQoS Policy is reduced to a MIP.

Definition 5.5. A *Mixed Integer Program*, MIP, is a pair, $(VarEntries, Constraints)$.

- $VarEntries$ is a set of $VarEntry \equiv (Var, Type, Weight)$, where $Type \in \{\Re, Binary\}$, $Weight \in \Re$
- $Constraints$ is a set of linear constraints (i.e., $Set\langle L\rangle$)

Now we describe the transformation of Policies to a MIP. Our description is bottom-up (from the Figure 5.1); we start by transforming the atomic statements, conjunctions, disjunctions, and finally the entire policy.

5.3.1 Transformation of Variables

```
(1) MIP genVariables(Policy policy)
(2)        MIP mip
(3)        foreach V ∈ policy.Vars
(4)           mip.VarEntries += VarEntry(V,Binary,1/policy.Prefs.indexOf(V))
(5)           mip.VarEntries +=
(6)               VarEntry(V',Binary,1/policy.Prefs.indexOf(not(V)))
(7)           mip.Constraints += L({(V,1),(V',1)},'=',1)
(8)        foreach P ∈ policy.Parms
(9)           mip.VarEntries += VarEntry(P,ℜ,0)
(10)       return mip
```

The transformation starts by generating a *consistency constraint* for every feature status variable in the policy. Lines 3-7 add two binary variables named, V and V', for a boolean variable V, to the MIP. The weight is chosen to be the inverse of the position in the `Preference` clause. A new constraint,

$$V + V' = 1$$

is then added. It prevents the two binary variables representing the boolean variable and its negation from taking on the same truth value simultaneously. Lines 8-9 map each feature parameter directly to a variable of type $\Re$.

5.3.2 Transformation of Atomic Statements

```
(1) $Set\langle L\rangle$ genConstraints(C conj)
(2)    $Set\langle L\rangle$ ls
(3)    Components bools
(4)    foreach V in C.Pos
(5)       bools += Component(V,1.0)
(6)    foreach V in C.Neg
(7)       bools += Component(V',1.0)
(8)    ls += L(bools,'=',$|bools|$)
(9)    ls = ls $\cup$ conj.Constraints
(10)   return ls
```

Figure 5.4: Transformation of Atomic Statements

All feature status variables in a single conjunctive clause will be transformed into an equivalent linear constraint (lines 3-8) (i.e., n variables to 1 constraint). First, we create the constraint variables by including one for each feature status variable occurrence. The occurrence could be negated or not and is taken into consideration in lines 5 and 7 through mapping to V or V'.

Coefficients of 1.0 are generated for each constraint variable. This is so that the sum of the left-hand side matches the number of "true" (value 1) constraint variables.

A conjunction requires that all of its feature status variables are true. So, the right-hand side must be equal to exactly the number of constraint variables in the clause. This is represented by the new constraint in line 8 and formalized in the following lemma.

Lemma 5.6. $t_1 \wedge t_2 \wedge .. \wedge t_n \leftrightarrow (b_1 + b_2 + ... + b_n) = n$

Given that feature status variables in a conjunction are represented as t_i and constraint variables represented as b_i. Here we see the correspondence between the conjunction of feature status variables and a single linear constraint.

Finally, all the parameter constraints are directly mapped by adding them to the return set of constraints (line 9). Then we return the final set of constraints corresponding to the input conjunction. We call this set of constraints the *original* constraints; they *originated* from a conjunction in the policy; later they will be augmented to account for disjunctions in the policy.

Theorem 5.7. *A conjunction is satisfiable if and only if all the original constraints it is mapped to by the transformation procedure are satisfiable.*

A conjunction is satisfied if all its atomic statements are satisfied. Lemma 5.6 shows the correspondence for feature status variables and one original constraint. The other atomic statements in a conjunction, the parameter constraints, are mapped directly into linear constraints (the correspondence is direct). This shows the correspondence for the entire conjunction. Now we make the steps described so far more concrete with an example.

Example The conjunction,

$$(CPP \wedge Authentication \wedge (CPP.size \geq 4))$$

is mapped into the system of equations,

$$CPP + Authentication = 2$$

$$CPP.size \geq 4.0$$

$$CPP + CPP' = 1$$

$$Authentication + Authentication' = 1$$

$$s.t.\ CPP, CPP', Authentication, Authentication' \in 0, 1$$

$$CPP.size \in \Re$$

5.3.3 Transformation of Disjunction of Conjunctions

All status variables, features parameters, and individual conjunctions have been transformed, but we must still take into account the disjunction of conjunctions.

This is done by augmenting each original constraint by an expression in a new constraint variable rx_i [78]. All L originating from C_i are related because the choice of value in rx_i has an effect on each. Given rx_i, the constraints augmented by rx_i are identified by the notation L/rx_i.

Here is the effect we desire: if $rx_i = 1$ then all L/rx_i are satisfied regardless of the original constraints. Hooker et al. [39] identify this effect as *relaxation* of constraints by the variable rx_i. We limit our discussion of relaxation to constraints with a $\geq$ operator; the case for $\leq$ is symmetric. Equality constraints are not handled by relaxation.

To achieve the desired effect we must choose a coefficient, m (referred to in the literature as "Big-M") for rx such that when $rx = 1$ the left-hand side is greater than or equal to any possible right-hand side (m acts like infinity). Our choice of m is made on the basis of the maximum real value representable in the policy language.

This augmentation step is carried out in Figure 5.5. Line 2 sets the value of m according to the value we desire. Lines 4-5 handle the case where the constraint operator is a $\geq$. We return the augmented constraint by adding the new variable, rx, with the coefficient m. In lines 6-7 the case for $\leq$ constraints is handled by reversing the sign of m. In lines 8-10 equalities are dealt with. Since the BigM formulation only applies to inequalities [39] we simply transform the equality into two inequalities: one $\geq$ and one $\leq$.

```
(1) $Set\langle L\rangle$ BigM($Set\langle L\rangle$ conj, Var rx)
(2)     m = $(MaxValue^2 * MaxVars) + MaxValue$
(3)     $Set\langle L\rangle$ ls
(3)     foreach l $\in$ conj
(4)        if(l.Op == $\geq$)
(5)          ls += L(l.Component $\cup$ Component(rx,m),$\geq$,l.Rh)
(6)        else if(l.Op == $\leq$)
(7)          ls += L(l.Component $\cup$ Component(rx,-m),$\leq$,l.Rh)
(8)        else if(l.Op == '=')
(9)          ls += L(l.Component $\cup$ Component(rx,m),$\geq$,l.Rh)
(10)         ls += L(l.Component $\cup$ Component(rx,-m),$\leq$,l.Rh)
(11)    return ls
```

Figure 5.5: "Big-M" Transformation

```
(1) L relaxation($Set\langle Var\rangle$ rxs)
(2)    $Set\langle\Re\rangle$ coefficients
(3)    foreach rx in rxs
(4)       coefficients = 1.0
(5)    return L(rxs,coefficient,$\leq$,|rxs|-1)
```

Figure 5.6: Linear Relaxation

Now we arrive at the last step in the transformation. There were two important steps so far: the transformation of atomic statements into equivalent linear constraints and the augmentation of constraints with BigM. The augmentation was formed in such a way that constraints originating from C_i could selectively be relaxed by choosing the value 1 for rx_i. Here, we see how this is used to represent disjunctions of conjunctions.

Let $rx_1 ... rx_n$ represent the variables generated for all the conjunctions in a single disjunction. The requirement is that one rx_i is equal to 0, reducing L/rx_i to the original constraints. Now, $L/rx_i \leftrightarrow C_i$ by Theorem 5.7. This requirement is enforced by adding the following constraint to the MIP (lines 3-5 in Figure 5.6),

$$\sum_{i=1}^{n} rx_1 ... rx_n \leq n - 1$$

Without this constraint, a solution could always be constructed by choosing all rx_i to be 1. We want to make sure that a solution is only found when at least one of the conjunctions is true. So, at least one of the rx_i must be 0.

5.3.4 Transformation of Policies

```
(1) $MIP$ genMIP(Policy policy)
(2)       $MIP$ mip = genVariables(policy)
(3)       foreach D $\in$ policy.Clauses
(4)          $Set\langle Var\rangle$ cs
(5)          foreach C $\in$ D
(6)             Var rx = freshVariable()
(7)             rxs = rxs $\cup$ rx
(8)             mip.Constraints += BigM(genConstraints(C),rx)
(9)       mip.Constraints += relaxation(rxs)
(10)      return mip
```

Figure 5.7: Overall Policy $\rightarrow$ MIP translation

The transformation steps we have covered are summarized in the overall Policy to MIP conversion shown in Figure 5.7. First, we transformed all the variables in the policy (line 2). Then, we transformed all the conjunctions in a disjunction. Each conjunction was associated with a new variable (as in line 6) to selectively relax the constraints originating from it. The constraints are generated in line 8 and augmented by the BigM procedure. All the relaxation variables are collected (in the *Set* declared on line 4) to create a relaxation constraint in line 9 for each disjunction. Now the MIP is completed. Here, we present a proof that the transformation is correct.

5.3.5 Proof of Transformation Correctness

Proof. We must show that the Policy is satisfied $\leftrightarrow$ the MIP is satisfiable. A model for a policy determines a model for a MIP by mapping any true valued boolean variable, V, into a 1.0 value for V in MIP and a 0 value for V' in MIP. The opposite occurs for false valued boolean variables. Feature parameter values in a Policy model determine the corresponding real valued constraint variables in MIP. The values of rx variables in constraints are free to take any value; they are not determined by the mapping from a Policy model. The relaxation constraint simply prevents a MIP from being satisfiable when the corresponding Policy is not satisfiable.

$\rightarrow$ Suppose *Policy* P is satisfied, but the MIP is not. We will show by construction that a model for P can be transformed into a model for MIP. Since P is satisfied, all disjuncts are satisfied. Without loss of generality, pick one disjunct D_i. Some C_s in this disjunct must be satisfied. We set that corresponding rx_s in the MIP to be 0, and this satisfies the relaxation constraint associated with the disjunct; all the other rx's arising from D_i are set to 1, thus satisfying the associated linear constraints. Now, by Theorem 5.7, we can construct a model for all the other linear constraints in MIP arising from C_s. This means that the all constraints arising from C_s are satisfied, and all others in D_i are relaxed. Similarly, we can argue that all MIP constraints from the other disjuncts in the entire policy P can be satisfied.

$\leftarrow$ Suppose MIP is satisfiable but the *Policy*, P, that it transforms into is not satisfiable. This means that some disjunctive clause, D_u in P is unsatisfiable, but every constraint $L \in MIP$ is satisfiable. We know every $C \in D_u$ is unsatisfiable since D_u is a disjunction.

Now, consider the relaxation clause generated by D_u. Since it is satisfied, there is at least one rx_j with a value of 0, such that $C_j \in D_u$. So all L originating from C_j are reduced to the original constraints. By Theorem 5.7, we have a contradiction, since C_j must be satisfiable. $\square$

5.4 Implementation

Our prototype implementation builds on the existing Dado middleware and the Lindo API for mixed integer programming. This involves attaching policies to applications, maintaining a run-time representation of policies, and finally deploying the properly parameterized resolved features.

A deployment expert considers local requirements and feature interactions to design a QoS Policy. The policies are associated with CORBA interface types, before an application is executed. Our implementation currently does not support policies on a per-method basis; a single policy can be assigned to each interface type. At application load-time the GlueQoS middleware builds a data-structure representing these policies. Now we describe the overall set-up as in Figure 5.8.

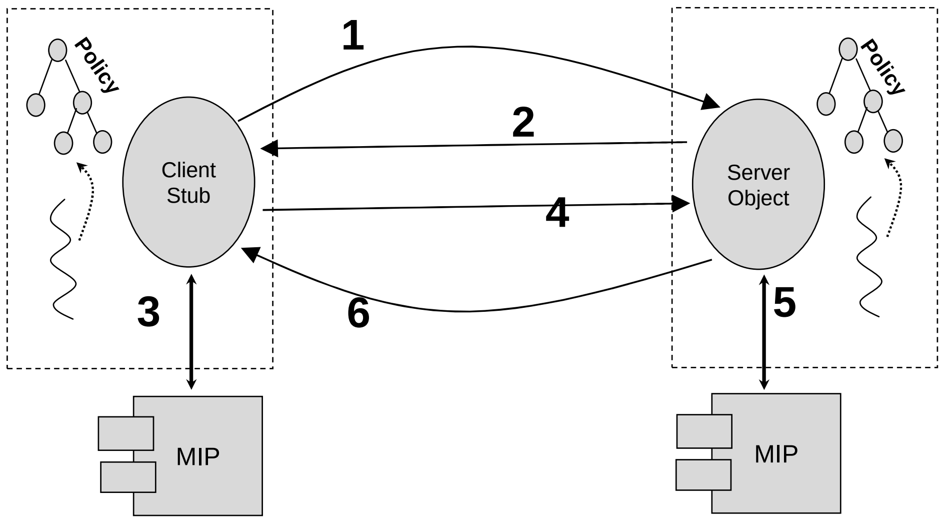

Figure 5.8: The overall flow of the GlueQoS run-time, including client stub, server stub, and the Mixed Integer Programming (MIP) run-time component

The figure represents the client and server run-time using our GlueQoS middleware, separated on the left and right sides respectively. Each side is symmetric in terms of its structure but not in terms of its responsibilities. The dotted-line boxes represent the boundary between middleware related functionality and the black-box MIP component.

Inside the dotted lines are three pieces. First, the large circles represent the client stub and server object to which the session based feature resolution applies. Second, the tree of nodes represents the policy data-structure. Third, a separate thread, shown as the curved line, is responsible for updating this data-structure based on the values retrieved from the user-defined functions. Now we focus on the interaction defined by the numbered flow of the diagram.

5.4.1 Client/Server Interaction

The GlueQoS middleware at each end of an interaction determines an acceptable feature combination for each application session. These features and their operating parameters remain fixed for the lifetime of the session.

When a client locates a server, it sends a *policy request* (1) to the server object to initiate a session. Policy requests are implemented as a CORBA operation that is transparently added (introduced) to all IDL interfaces. This is performed by the DDL compiler when it processes application IDL, as described in Chapter 3. The implementation of the policy request operation is introduced into the server skeleton using AspectJ.

The server creates a session for the client in the form of a *cookie*. The cookie is a generated number that will be used by the server to correlate client requests with their session. Now, the server serializes the policy data-structure, associates it with the newly created session, and returns the serialization to the client.

The client must match its own policy with the server and choose a feature combination acceptable to both. First, client and server data-structures are merged. Now, a client matches policies by carrying out the mixed integer program resolution. The merged data-structure and a vector representing the client's preferences are passed to the Lindo API. It will return a satisfying assignment for all variables or signal unsatisfiability (3). These results are used to control the execution of features. In the case of unsatisfiability, an exception is thrown to the application to signal incompatible policies. In our Dado middleware, features are implemented as adaptlets.

The model chosen by Lindo is used in the creation of adaptlets. The adaptlets whose status variables are enabled in the satisfying assignment are instantiated using the Java Reflection API. The parameter values chosen are passed to the adaptlets constructor. We assume that the signature of the constructor is standardized as are the parameters for each adaptlet. The values can then be used by the advice to configure adaptlet execution. In this way the resolved features are activated and configured according to the policies of both client and server.

The model chosen by the client is then sent to the server (4). This message is piggybacked on a subsequent application request to the server. The server must verify that the model chosen by the client actually satisfies its own policy. This requires only a simple linear time check of constraint satisfiability (5). The values for the variables are plugged into the policy which was associated with the client's session. If verification is successful, the server can discard the associated policy and create adaptlets in the manner described for the client side. On subsequent requests, the cookie from the client is used to execute adaptlets and advice on a per-client basis. If verification is unsuccessful an exception is thrown back to the client (6).

5.4.2 GlueQoS Prototype

Our GlueQoS implementation has been tested on the example presented in this chapter and an example in a related paper [89]. In the future we hope to further validate the approach by applying it in a more realistic setting.

To understand some of the performance impact induced by the GlueQoS software we measured the cost of the GlueQoS negotiation phase (Figure 5.8, steps 1 - 5) on the example of Figure 5.3 with the second client policy. Measurement was performed on a single 1.6Ghz Pentium M laptop computer with 1GB of main memory running both client and server processes. The negotiation phase is initiated when a client binds to a remote server object. We took timing measurements using the `java.lang.System.currentTimeMillis()` function before and after 1,000 calls to the function that performs binding. We performed ten trials of the 1,000 calls and took the average of the measurements. The average setup cost was 2.282ms. This is an order of magnitude smaller than the end-to-end communication latency in many current Internet application settings. For example, at the time of writing, the latency from the University of British Columbia to the University of California, Davis was about 66ms as measured by the `ping` UNIX application. So, for this particular example the overhead of the protocol would be dominated by communication costs of steps 1, 2, and 4 in Figure 5.8.

An important detail missing from this experiment is the fact that only a single example policy was used. Since the policy solver of step 3 grows exponentially with the number of integer variables required in the policy encoding, it will be important to repeat the experiments for a range of policy sizes. We could draw from the approach described in [56]. This work shows how to generate random 3-SAT instances of a desired size and difficulty (i.e., time required to solve the instance). In the future it may be possible to extend this work for generating random policies of varying difficulty that can be used for further experiments.

5.5 Conclusion

GlueQoS is middleware software to support dynamic adjustment of QoS features between clients and servers. QoS feature preferences are specified in the GlueQoS policy language. These policies are exchanged at binding time between systems interacting in an ad-hoc setting. The polices are then matched up, and resolved by the middleware. The resolved features are then deployed and executed. This chapter described GlueQoS and provided an illustrative example. GlueQoS has been implemented in the context of adaptlets.

Chapter 6

Related Work

6.1 Middleware Based Adaptation

6.1.1 Foundations

The motivation for work on middleware based adaptation can be traced back to the theoretical study of adaptation in distributed systems. Early work on termination algorithms recognized the utility of separation of the core algorithm from the termination criteria [11]. Chandy and Misra presented a termination algorithm that could be reused by interleaving the actions of a chosen algorithm with the termination algorithm. This model of stepwise refinement has come to be known as Superposition or Superimposition. The integration of first-class superimposition constructs in a programming language was presented by Katz [42]. Other work focused on the verification and validation of superimposed algorithms [7]. Minsky [55] identified the usefulness of these ideas in the separation of mandatory security controls from the core implementation of a distributed system. In his work the "law" of a system is comprised of the implementation of global security constraints only. A preprocessor is used to interleave these controls with the implementation in a Prolog based distributed language.

6.1.2 Customizable Middleware

As the environments in which applications based on middleware were deployed in became increasingly heterogeneous, it became clear that one size fits all solutions would not work. By exposing certain components in the middleware implementation, customizable middleware platforms provided a means for programmers to plug-in middleware components customized for their environment. Projects such as Flick [29] and COMERA [100] focused mainly on the customization of the stub layers. Flick takes the approach of allowing developers to customize the IDL compiler directly. This allows for a greater range of optimization possibilities.

Work on design patterns and object-oriented frameworks influenced the design of The ACE ORB (TAO) [74]. By virtualizing the ORB components and relying on pluggable factories it became easier for developers to plug in their own concrete ORB implementation classes.

Quarterware [79] is a software architecture for basic middleware functionality. Its authors demonstrate how this architecture can be customized to provide an implementation of CORBA, RMI, or MPI (Message Passing Interface) middleware.

More and more, middleware platforms were being relied upon for critical system infrastructure in areas such as telecommunications and military. Bringing these systems offline in order to program reconfiguration became too costly.

6.1.3 Dynamic Customization

Motivated by work on dynamically reconfigurable distributed systems [48], middleware research looked toward dynamic customization of middleware.

DynamicTAO [47] extends the TAO CORBA middleware with support for runtime changes of ORB properties. New security, threading, and monitoring behaviors can be added or removed dynamically. Control of dynamic changes can be made remotely through a distributed configuration management tool.

Lasagne [94] is a framework for dynamic and selective combination of extensions in component based applications. Each component can be combined with a set of wrappers to refine the interaction behavior with other components. Every wrapper layer is tagged with an extension identifier. Related wrapper layers are tagged with the same identifier in order to group them into a cohesive set. At run-time client components attach a set of extension identifiers to the context of a remote procedure call. This communicates to server components which wrapper layers should be invoked for this call allowing selective per-invocation adaptation. Since related layers on disparate components share the same identifier this allows clients to turn on and off adaptive behavior that crosscuts component implementations.

The Cactus [12] framework provides customizations in distributed systems and has been applied in the area of distributed object computing. Software implementing customizations is known as an adaptive component (AC). ACs are injected into a system using the PROXY pattern. These ACs encapsulate alternative implementations of a specific service using the STRATEGY pattern. When important state changes in the system occur ACs can respond by swapping in and out different strategies. ACs may communicate in order to reach consensus on strategy changes however their model dictates no one particular consensus approach.

The reconfiguration in these systems is typically driven by outside forces such as configuration management tools. To provide a framework where middleware components could reason about their own configuration, middleware designs began to provide more support for reflective software extensions.

6.1.4 Reflection

Reflection refers to the ability of a computational system to reason about its own structure or behavior. Mature middleware platforms such as CORBA already provided many reflective capabilities. The Interface Repository (IR) supports read-only run-time data structures describing an object's functionality. CORBA also offers the Dynamic Invocation Interface (DII) and Dynamic Server Interface (DSI) facilities for constructing invocations to or handling of invocations to interface types not know statically. Borrowing from the idea of meta-object protocols [44], interceptors were developed as a way adding behavior before and after the events of applications hosted inside the middleware. Interceptors can make use of information about the events such as function call names, parameter values, and target types. Portable Interceptors are now part of the CORBA 3.0 standard and are used for introducing security, transaction propagation, and real-time scheduling. Narasimhan et al. [60] show how to use interceptors for introducing fault-tolerance into a CORBA based application.

Similar to the approach taken by Lasange, *communication reflection* reifies the channels between client and server to address adaptation on a per-message level [10]. Meta-Classes were exploited in the FRIENDS [30] project using OpenC++ in order to add fault tolerance and security to distributed programs. OpenCOM [16] takes the idea of reflection a step further by reifying the communication between an ORB's internal components. This exposes more than just the application events for applying reflective components such as interceptors. A more detailed comparison of reflective APIs in the OpenCOM and DynamicTAO middleware is presented in [46].

Developers were recognizing that certain adaptations crosscut the implementation of both the application and the middleware, making these adaptations especially difficult to program.

6.1.5 Containers

The Enterprise Java Bean component model introduced a new middleware model called Container Based Deployment in order to simplify the deployment of some adaptations. Container deployment tools make all necessary configuration changes to the application and middleware for the container supported adaptations. The idea of having a “kit” of adaptations that developers could easily deploy led to the success of the model in industry. However, the EJB model still lacked the flexibility of customization needed by many developers.

6.1.6 Aspect-Oriented Middleware

Aspect-Oriented Middleware is motivated by the need to provide flexible customization with a simplified deployment process, combining the benefits of reflective middleware with container based deployment.

Recently, the open-source JBoss [41] application server announced aspect-oriented deployment of container services using the Javassist [15] byte code editing toolkit. A similar approach is used in the Java Aspect Components (JAC) framework [67] that also utilizes load-time byte code weaving (using BCEL [19]) in Java. New services can be constructed by implementing aspect-specific interceptors. Deployment takes place using the notion of pointcuts from the AspectJ language. In JAC, aspects can be un-deployed/re-deployed dynamically using a standardized API.

The Quality of Objects (QuO) [51] project aims to provide consistent availability and performance guarantees for distributed objects in the face of limited or unreliable computation and network resources. QuO defines an abstraction known as the operating region for processes (client or servers) cooperating in a distributed object environment. Changes in perceived runtime conditions move a process into different operating regions. Application code that is bound to a particular operating region or region transition is the main vehicle by which adaptation is achieved. Objects known as System Conditions are used by the QuO system to monitor the runtime conditions of the operating environment. Programmers using QuO write code for System Condition objects to affect the procurement of operating condition variables. The QuO language is used to define predicates over the values of System Condition objects to determine the current operating region. QuO extends the notion of AOP by introducing transition callbacks. This functionality is executed when a transition is perceived from one operating region to another. An example of this is for migrating objects when a system condition object denoting a danger status reaches a certain level.

The Desault Systemes Component Virtual Machine (DS CVM) [28] provides aspect-oriented adaptations using the CORBA Component Model (CCM). The CCM model makes explicit the connection between specific component implementations and interfaces using deployment descriptors. Because of this the DS CVM aspect language can discriminate between different implementation entities. This aspect language also provides hooks for controlling the tools used to generate components and services. Another advantage of DS CVM is the use of customizable communication proxies to handle lower level adaptation. These can be used for example to provide custom marshaling routines, this feature was present in Flick and COMERA but lacking in the newer aspect-oriented approaches.

The aspect-oriented middleware presented in this section achieve both flexible customization and simplified deployment. This is made possible by a clear separation between adaptation programming and deployment. Deployment is facilitated by pointcut based descriptions which map adaptation behavior to application events.

6.2 Other Areas

6.2.1 Wrapper-Based Techniques

Monitoring the information flow at the application to operating system interface has often been used for the transparent addition of security behavior in a system. This idea was popularized in a paper by Stephanie Forrest et al. [32]. These ideas sparked an entire sub-field of computer security known as *hardening.* The goal of hardening is to provide language and tool support for the transformation of insecure applications to that of a secure one. Generic Software Wrappers [33] were proposed as a means for simplifying the process of hardening for Commercial Off-The-Shelf (COTS) applications. Loadable Kernel Modules are used as a hook by which security wrappers gain access to system calls. A language called Wrapper Definition Language is introduced to specify monitoring behavior.

6.2.2 Binary Editing

Recent research has exploited the fact that Java programs are typically deployed in a format known as byte-code. Because byte-code is relatively high level it is easy to reconstruct and edit programs statically or even at load time. BCA [43], Javassist [15], and BCEL [19] all provide programmed support for byte code editing . The Distributed Virtual Machine (DVM) [80] uses a network firewall to intercept and edit Java code downloaded by client virtual machines. The DVM can enable insertion of new security policies into Java programs. The Evolvable Systems Project provides similar functionality through dynamically adaptive security [66]. J-Orchestra [92] uses byte code editing to transparently introduce distribution into standard programs.

6.2.3 Mobile and Pervasive Adaptation

Several projects are focusing on adapting network communication primarily for supporting different QoS. This can be useful for mobile applications where bandwidth is limited.

Puppeteer [23] performs adaptations of content serving distributed applications (such as web servers). The interception mechanism performs transformations of content based on users preferences. Puppeteer has been used to add content pre-fetching, caching, and compression.

CANS [34] is a Composable Adaptive Network Services Infrastructure. This technology provides a pipelined architecture for the transformation of network data streams. Events such as decreased bandwidth can trigger the reconfiguration of adaptation components.

The techniques described in this section provide a valuable feature that is not directly addressed in our work. Low-level control over network stream representation is required for some services such as encryption/compression. However, if an appropriate API is supported by middleware, programmers can make use of Dado's crosscutting description language and cross-host coordination to invoke the API for making these types of low-level changes.

6.2.4 Software Architecture

In software architecture, connectors [54, 68, 3] have proven to be a powerful and useful modeling device. Connectors reify the concern of interaction between components, and are a natural foci for some crosscutting concerns. Implementations of architectural connectors have also been proposed [27, 76, 21, 83]. Some of these provide specific services [76, 21, 83] over $\mathcal{D}H$ middleware, such as security. Our work can be viewed as providing a convenient implementation vehicle for different connector-like services in a heterogeneous environment. The DDL language and compiler allow service builders to write client and server adaptlets that provide many kinds of "connector-style" functionality, while the DDL "plumbing" handles the communication details. Furthermore, the pointcut language allows a flexible way of binding this functionality to components, using pattern matching to bind events to adaptlets. The question as to whether connector specifications (e.g., in an ADL) can be translated to DDL specifications and pointcuts is interesting, and we hope to address it in future research.

Cheng et al. [13] propose a software architecture based adaptation for pervasive systems. Their framework consist of three layers: the task layer, the model layer, and the runtime layer. The task layer is responsible for scheduling users tasks based on negotiated QoS. The model layer is programmed with information regarding how to adapt when QoS requirements can not be met. The runtime layer provides a low level API that is used by the model layer to make required changes. This approach is similar to the Quality of Objects *QuO* project.

6.2.5 Programming Language Approaches

Several programming languages provide built in support for static and run-time program transformations.

Many programming languages use the concept of reflection to manipulate program structure. Three types of reflection have been identified. Introspection provides a read-only view of the program (e.g., java.lang.reflect and C++ RTTI). Behavioral reflection allows only to wrap functions with "before" and "after" behavior. Structural reflection provides complete program rewriting. Smalltalk [70] and CLOS [44] use meta-objects and dynamic inheritance to provide introspection and behavioral adaptation. Because they are dynamically typed they enjoy none of the advantages of static-type checking. Open-C++ [14] provides structural reflection only at compile time and was extended to load-time with JavaAssist [15], no changes can be made at run-time. Czarnecki [18] exploited the C++ templates system for similar types of program transformation. Mixin-Layers [81] are an approach for encapsulating object-oriented collaborations as composable software layers. Collaborations cut across several objects so mixin-layers provide support for incremental addition of new features across entire layers. The Dynamic Virtual Machine [53] allows changes to be made at run-time in Java and has been extended for distributed settings.

A wide variety of language based techniques support transparent and semi-transparent adaptation of program source. None of them can support type-checked interactions between adaptations across heterogeneous network hosts. However, we hope to promote the usage of programming language specific features as implementation techniques for easing Dado into existing middleware for these languages.

6.2.6 Advanced Separation of Concerns

Advanced Separation of Concerns (ASOC) seeks to provide modularity and reuse for many aspects of program behavior. Certain types of application requirements such as security, concurrency, or monitoring are known to crosscut the program structure. ASOC techniques attempt to factor out the implementation of all concerns into well defined modules. The techniques differ in the way actual programs can be decomposed at design time and reconstructed at compile time. AspectJ [45] uses Aspect-Oriented Programming as described in Chapter 2. HyperJ [90] and Subject-Oriented Programming [36] decompose programs into multi-dimensional inheritance hierarchies. The hierarchies are composed using a language similar to pointcuts. Composition Filters [2] extends the object-oriented model by providing a model where messages between objects must pass through any number of user defined filters. Filters have been used to separate synchronization and concurrency concerns from object-oriented programs without introducing inheritance anomalies.

EEK [97] occurs whenever a needed datum is passed through a component to another object further down on the invocation chain. Other forms of EEK occur when components become dependent on the identity of other components where the actual dependency is one of functionality and not identity. Programs can be designed without EEK and later modified with boundary maps to install the required control flow and data flow for a functional system; this is called *Contextual Dispatch*. It may be possible to leverage Dado for solving similar problems in a distributed heterogeneous context.

We have already used AspectJ as one tool for introducing Dado into the JacORB middleware. By relying on mature program transformation techniques such as these instead of platform dependent API's, we feel the Dado framework can ensure its role as a useful mechanism for distributed transformations.

6.2.7 Electronic Contracts and Service-Level Agreements

Our work on QoS feature composition relates to work on electronic contracts [38] and (formal representations of) service-level agreements (SLAs), in particular those that address nonfunctional requirements such as WSLA (Web SLA) [52]. Such contracts define agreed-upon, non-functional (and possibly other) characteristics of (Web) services and a model for measuring, evaluating, and managing the compliance of these characteristics. Their representation involves assertions comparable to our policies, and algorithms for assertion match-making and negotiation have been developed in the context of self-managing systems (systems management) and dynamic e-business.

However, SLAs such as those described using WSLA focus on performance characteristics only, and on locally (nondistributed) measurable phenomena as seen by the client. The service provider's performance offerings are matched with a client's specified expectations to determine a binding contract. Adherence to this contract can be monitored. GlueQoS feature policies address a wider range of end-to-end quality-of-service requirements, such as transactions, security, etc. In addition, GlueQoS has been designed to support policy-driven configuration of client and server middleware to ensure interoperability. SLAs can be used to model contracts that can still be violated. With GlueQoS, an interaction is only executed if a compatible feature composition has been determined, and where no violation should be possible.

6.2.8 Requirements Engineering

As described in Figure 3.1 we envision our middleware to be part of a larger software process. Work on managing conflicts between requirements or features is especially relevant.

The KAOS [95] methodology uses formal specification to detect conflicting requirements using goals which can be identified using a temporal logic. Feather [31] extends the approach for monitoring of actual runtime behavior. Their work is concerned with functional requirements as opposed to mediation of conflicting QoS requirements between software in a distributed setting.

Automatic detection of interacting features in communication systems is an area of active research. Much of the work focuses on reconciling customer features in a telephony setting such as call-waiting and voice mail. Zave [102] views features as modular components connected in a pipeline architecture similar to our layered feature architecture. She identifies an ontology of feature interactions and provides techniques for automatic detection of interactions.

We have not developed any formal techniques for identifying requirements level phenomenon. Our work involves a middleware to be used for mediating policies between QoS features once possible feature interactions have been elucidated.

Chapter 7

Conclusion

7.1 Discussion

Our framework for aspect-oriented development of distributed object applications consists of two main contributions: the Dado language and tools for developing and deploying adaptlets; and the GlueQoS language and software for resolving feature policies. These contributions are summarized in this conclusion.

Modeling, Type-Checking, and Marshaling Adaptlets employ an enhanced IDL and code-generation to support the following:

- Explicit IDL-level modeling of adaptlets and their interaction with application components.
- Safer interaction (via static type-checking) between adaptlets, with automated generation of marshaling code.

Pointcut-based Binding The enhanced IDL can describe the interfaces supported by adaptlets separately from a deployment description, which specifies the precise deployment context of a service. This allows a deployment expert to tune the connection between adaptlets and different application components. The binding language is agnostic with respect to the implementation; adaptlets could be incorporated into the existing application using static transformations (binary or source) or dynamic wrapping, depending on available tools or performance issues.

Multiple Contextual Invocations Middleware extensions allow adaptlets on the client and server side to communicate via messages. However, rather than inducing additional middleware invocations, multiple messages are piggy-backed within the single pre-existing application invocation.

Transparent Late binding Client software transparently (without additional programming) discover the services associated with a server, and deploy additional adaptlets as needed. We provide a declarative language for specifying the feature preferences and conflicts, and a middleware-based resolution mechanism, GlueQoS. The middleware reasons using these specifications to dynamically find a satisfying set of features that allow a pair of components to inter-operate.

We hope to foster a new design methodology for transparent addition of features to distributed applications that promotes reuse of software (adaptlets) between developers. The goal is to reduce development costs associated with middleware application maintenance and deployment. Static type-checking of adaptlet interactions should increase the dependability of software making use of cross-host features. We believe transparent deployment of adaptlets onto existing applications will make it easier to evolve components to new usage scenarios. Although some performance overhead was observable in performance measurements, we showed this to be fairly small (less than 2%) when used a part of a typical application.

7.2 Dissemination

Currently our system has not been used by an outside third-party. Consequently, it is still difficult to assess the long-term success of these ideas. Recently, this work was presented to the Object Management Group (OMG) at an official technical meeting. The OMG is a consortium of companies responsible for standardizing and promoting the CORBA technology. We have been invited back to an upcoming meeting to further discuss how the work in this thesis can be used to improve upcoming OMG standards. Through these discussions we hope to lay the groundwork for dissemination of our research. This may provide us the opportunity for further validation through outside use of our framework.

An initial hurdle that must be overcome in order to see adoption of this research is in the training of developers who may benefit from its use. Technically, adoption can take place independently of other aspect-oriented approaches such as AspectJ. Realistically, it may be that the long-term success of our approach is tied to the success of Aspect-Oriented technologies in general. This can be understood in analogy to the success of distributed object middleware. For example, it can be argued that CORBA is useful in a non-object-oriented environment. However, we believe that much of its success hinged upon the adoption of the Java and C++ programming languages to which it was closely related. So, if the aspect-oriented way of programming becomes familiar to developers then we feel that the learning curve required to benefit from this research will be low.

7.3 Future Plans

In the future we plan to extend this research to the area of *cross-layer* adaptation. The key insight of cross-layer adaptation is that decisions made by code in a particular system layer can often be made better by knowing the state of other system or application layers. Dean et al. [24] show how to improve defense against denial of service attacks using a combination of application (web server) and transport layer information. The GRACE project [101] combines an adaptive multimedia application with soft-real time information from the operating system to improve playback of streams on mobile devices. While these projects provide motivation they do not address the challenges of software development in these settings.

Programming cross-layer adaptive software appears challenging because support must exist in the original software to allow run-time reconfiguration to take place. This support code may become tangled with the original functions inhibiting program understanding and maintenance. The key research question is to determine whether adaptation can take place across components at multiple layers (e.g., application, middleware, and operating system) without sacrificing the benefits of the original layered composition. Current approaches (e.g., [24, 101]) require tight coupling between components at different layers which may inhibit evolution and introduce significant administrative burdens.

The tools developed as part of this research may not directly support an environment for cross-layer adaptation. Our software is targeted at CORBA programming which is not a popular model for developing many important applications (such as web servers) or operating system modules. However, application and system layers are often built from a variety of different languages and platforms. Therefore we believe that this thesis may provide some useful insights and techniques that will be applicable in the cross-layer setting.

Appendix A

Additional Source Code

Java CPP Client Adaptlet Helper Function

```
boolean recursiveSolver(int n, PuzzleChallenge challenge)
{
  if(n == 0)
  {
      MessageDigest md5 = MessageDigest.getInstance("MD5");
      md5.update(challenge.puzzlePreImage);
      byte[] digest = md5.digest();
      if(!byteArrayEqual(challenge.puzzleHash,digest))
      {
        return true;
      }
      else
      {
        return false;
      }
  }
  else
  {
    for(int i=-127;i<127;i++)
    {
      challenge.puzzlePreimage[challenge.puzzleLength-n] = (byte) i;
      if(recursiveSolver(n-1,challenge))
      {
        return true;
      }
    }
  }
  return false;
}
```

C++ Server Adaptlet Helper Functions

```
bool checkResponse(PuzzleResponse * response)
{
  PuzzleChallenge challenge;
  table.find(response->puzzleID,challenge);
  if(!strncmp((const char *) challenge.puzzlePreImage.get_buffer(),
      (const char *) response->puzzleSolution.get_buffer(),64))
  {
    return true;
  }
  else
  {
    return false;
  }
}

PuzzleChallenge createChallenge(int size)
{
  char digest[16];
  PuzzleChallenge challenge;
  challenge.puzzleID = id;
  challenge.puzzleLength = size;
  challenge.puzzlePreImage = preImage();
  MD5((unsigned char *) digest,
      challenge.puzzlePreImage.get_buffer(),64);
  challenge.puzzleHash.replace(16,16,(unsigned char *) digest);
  table.bind(id,challenge);
  for(int i=0;i<size;i++)
  {
    challenge.puzzlePreImage[i] = 0;
  }
  id++;
  return challenge;
}

Octet64 preImage()
{
  Octet64 returnVal;
  returnVal.length(64);
  for(int i=0;i<64;i++)
  {
    returnVal[i] = (CORBA::Octet) rand()%256;
  }
  return returnVal;
}
```

Bibliography

[1] ACE and TAO Success Stories. <http://www.cs.wustl.edu/˜schmidt/ACE-users.html>. 12 pages.

[2] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In *Proc. of the Workshop on Object-Based Distributed Programming*, pages 152–184, 1993.

[3] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[4] T. Amsler and R. Walters. Open RCT home. <http://davinci.cs.ucdavis.edu/>.

[5] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. In *Proc. of the International Conference on Software Reuse*, pages 117–136, 2000.

[6] G. Blair and R. Campbell, editors. *Reflective Middleware*, 2000.

[7] L. Bouge and N. Francez. A compositional approach to superimposition. In *Proc. of the Symposium on Principles of Programming Languages*, pages 240–249, 1988.

[8] G. Brose. JacORB: Implementation and design of a Java ORB. In *Proc. of the International Working Conference on Distributed Applications and Interoperable Systems*, pages 143–154, 1997.

[9] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[10] W. Cazzola and M. Ancona. mChaRM: A reflective middleware for communication-based reflection. Technical Report DISI-TR-00-09, Universit degli Studi di Genova, 29 pages, 2000.

[11] M. Chandy and J. Misra. An example of stepwise refinement of distributed programs: Quiescence detection. *ACM Transactions on Programming Languages and Systems*, 8(3):326–343, 1986.

[12] W.-K. Chen, M. Hiltunen, and R. Schlichting. Constructing adaptive software in distributed systems. In *Proc. of International Conference on Distributed Computing Systems*, pages 635–643, 2001.

[13] S.-W. Cheng, D. Garlan, B. Schmerl, J.P. Sousa, B. Spitznagel, P. Steenkiste, and N. Hu. Software architecture-based adaptation for pervasive systems. In *Proc. of the International Conference on Architecture of Computing Systems*, pages 67–82, 2002.

[14] S. Chiba. A metaobject protocol for C++. In *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 285–299, 1995.

[15] S. Chiba. Load-time structural reflection in Java. In *Proc. of the European Conference on Object-Oriented Programming*, pages 313–336, 2000.

[16] M. Clarke, G. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Proc. of the International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, pages 160–178, 2001.

[17] Y. Coady, A. Brodsky, D. Brodsky, J. Pomkoski, S. Gudmundson, J.S. Ong, and G. Kiczales. Can AOP support extensibility in client-server architectures? In *Proc. of the ECOOP Workshop on Aspect-Oriented Programming,* 4 pages, 2001.

[18] K. Czarnecki, U. Eisenecker, and P. Steyaert. *Generative Programming : Methods, Tools, and Applications.* Addison-Wesley, 2000.

[19] M. Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universit at Berlin, Institut fur Informatik, 31 pages, 2001.

[20] G.B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, N.J., 1962.

[21] E.M. Dashofy, N. Medvidovic, and R. Taylor. Using off-the-shelf middleware to implement connectors in distributed architectures. In *Proc. of the International Conference on Software Engineering*, pages 3–12, 1999.

[22] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–205, 1960.

[23] E. de Lara, D. Wallach, and W. Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *Proc. of the USENIX Symposium on Internet Technologies and Systems*, pages 159–170, 2001.

[24] D. Dean and A. Stubblefield. Using client puzzles to protect TLS. In *Proc. of the USENIX Security Symposium,* 9 pages, 2001.

[25] J. des Rivieres and B. Smith. The implementation of procedurally reflective languages. In *Proc. of the Symposium on LISP and functional programming*, pages 331–347, 1984.

[26] T. Dierks and C. Allen. Internet RFC 2246: Transport layer security, 1999.

[27] S. Ducasse and T. Richner. Executable connectors: Towards reusable design elements. In *Proc. of the Symposium on Foundations of Software Engineering*, pages 483–499, 1997.

[28] F. Duclos, J. Estublier, and P. Morat. Describing and using non-functional aspects in component based applications. In *Proc. of the International Conference on Aspect-Oriented Software Development*, pages 65–75, 2002.

[29] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: A flexible, optimizing IDL compiler. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 44–56, 1997.

[30] J.-C. Fabre and T. Perennou. A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers*, 47(1), 1998.

[31] M.S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *Proc. of the International Workshop on Software Specifications and Design*, pages 50–60, 1998.

[32] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for Unix processes. In *Proc. of the Symposium on Security and Privacy*, pages 120–128, 1996.

[33] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *Proc. of the Symposium on Security and Privacy*, pages 2–16, 1999.

[34] X. Fu and V. Karamcheti. CANS: Composable, adaptive network services infrastructure. In *Proc. of the USENIX Symposium on Internet Technologies and Systems,* 12 pages, 2001.

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1994.

[36] W. Harrison and H. Ossher. Subject-Oriented programming: A critique of pure objects. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 411–428, 1993.

[37] D. Hilbert and W. Ackermann. *Principles of Theoretical Logic.* Springer-Verlag, 1928.

[38] Y. Hoffner, S. Field, P. Grefen, and H. Ludwig. Contract-driven creation and operation of virtual enterprises. *Computer Networks*, 37(2):111–136, 2001.

[39] J.N. Hooker and M.A. Osorio. Mixed logical/linear programming. *Discrete Applied Mathematics*, 96(97):395–442, 1999.

[40] R. Housley, W. Ford, W. Polk, and D. Solo. Internet RFC 2459: Internet x.509 public key infrastructure certificate and CRL profile, 1999.

[41] JBoss. <http://www.jboss.org>. 4.0 edition.

[42] S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, 1993.

[43] R. Keller and U. Hölzle. Binary component adaptation. In *Proc. of the European Conference on Object-Oriented Programming*, pages 307–324, 1998.

[44] G. Kiczales and J. des Rivieres. *The art of the metaobject protocol.* MIT Press, Cambridge, MA., 1991.

[45] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proc. of the European Conference on Object-Oriented Programming*, pages 327–355, 2001.

[46] F. Kon, F. Costa, G. Blair, and R. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, 2002.

[47] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L.C. Magalhães, and R. Campbell. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In *Proc. of the International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, pages 121–143, 2000.

[48] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[49] Lindo API. <http://www.lindo.com/>. 2.0 edition.

[50] R. Lougee-Heimer. The Common Optimization INterface for operations research: Promoting open-source software in the operations research community. *IBM Journal of Research and Development*, 47(1):57–66, 2003.

[51] J. Loyall, D. Bakken, R. Schantz, J. Zinky, D. Karr, R. Vanegas, and K. Anderson. QuO Aspect languages and their runtime integration. In *Proc. of the Workshop on Languages, Compilers and Runtime Systems for Scalable Components,* 16 pages, 1998.

[52] H. Ludwig, A. Keller, A. Dan, R.P. King, and R. Franck. A service level agreement language for dynamic electronic services. *Electronic Commerce Research*, 3(1-2):43–59, 2003.

[53] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J.F. Barnes. Runtime support for type-safe dyanmic Java classes. In *Proc. of the European Conference on Object-Oriented Programming*, pages 337–361, 2000.

[54] N. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proc. of the International Conference on Software Engineering*, pages 178–187, 2000.

[55] Naftaly H. Minsky. The imposition of protocols over open distributed systems. *IEEE Transactions on Software Engineering*, 17(2):183–195, 1991.

[56] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Prof. of the Conference on Artificial Intelligence*, pages 459–465, 1992.

[57] G. Murphy, A. Lai, R. Walker, and M. Robillard. Separating features in source code: An exploratory study. In *Proc. of the International Conference on Software Engineering*, pages 275–284, 2001.

[58] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *Software Engineering*, 18(6):483–497, 1992.

[59] P. Narasimhan, K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Providing support for survivable CORBA applications with the Immune system. In *Proc. of the International Conference on Distributed Computing Systems*, pages 507–516, 1999.

[60] P. Narasimhan, L. Moser, and P. Mellior-Smith. Using interceptors to enhance CORBA. *IEEE Computer*, 32(7):62–68, 1999.

[61] NSF Middleware Initiative. *National Science Foundation.* <http://www.nsf-middleware.org/middleware/>.

[62] Object Management Group. *CORBA 3.0 Specification*, 3.0 edition.

[63] Object Management Group. *CORBA Transaction Service*, 1.4 edition.

[64] Open Systems Architecture Puts Six Bombs on Target. *<http://www.cs.wustl.edu/˜schmidt/TAO-boeing.html>. 2 pages.*

[65] P. Pal, J. Loyall, R. Schantz, J. Zinky, R. Shapiro, and J. Megquier. Using QDL to specify QoS aware distributed (QuO) application configuration. In *Proc. of the International Symposium on Object-Oriented Real-time Distributed Computing,* 12 pages, 2000.

[66] R. Pandey and B. Hashii. Providing fine-grained access control for Java programs via binary editing. *Concurrency: Practice and Experience*, 12(14):1405–1430, 2000.

[67] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible framework for AOP in Java. In *Proc. of the International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection),* 24 pages, 2001.

[68] D. Perry and A. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[69] H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders. Quantifying the cost of providing intrusion tolerance in group communication systems. In *International Conference on Dependable Systems and Networks*, pages 229–238, 2002.

[70] F. Rivard. Smalltalk: A reflective language. In *Proc. of the International Conference on Metalevel Architectures (Reflection),* 18 pages, 1996.

[71] M. Robillard and G. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proc. of the International Conference on Software Engineering*, pages 406–416, 2002.

[72] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and J.B. Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proc. of the USENIX Windows NT Workshop*, pages 1–8, 1997.

[73] D. Sames, B. Matt, B. Niebuhr, G. Tally, B. Whitmore, and D. Bakken. Developing a heterogeneous intrusion tolerant CORBA system. In *International Conference on Dependable Systems and Networks*, pages 387–397, 2002.

[74] D. Schmidt. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, 38(10):65–74, 1995.

[75] D. Schmidt, D. Levine, and T. Harrison. The design and performance of a real-time CORBA object event service. In *Proc. of the Conference on Object-Oriented Programming, Languages, Systems and Applications,* 19 pages, 1997.

[76] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *Software Engineering*, 21(4):314–335, 1995.

[77] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[78] M. Simonnard. *Linear Programming.* Prentice Hall, 1966.

[79] A. Singhai, A. Sane, and R. Campbell. Quarterware for middleware. In *Proc. of the International Conference on Distributed Computing Systems*, pages 192–201, 1998.

[80] E.G. Sirer, R. Grimm, A. Gregory, and B. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proc. of the Symposium on Operating Systems Principles*, pages 202–216, 1999.

[81] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. of the European Conference on Object-Oriented Programming*, pages 550–570, 1998.

[82] Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *Software Engineering and Methodology*, 11(2):215–255, 2002.

[83] B. Spitznagel and D. Garlan. A compositional approach to constructing connectors. In *Proc. of the Working IEEE/IFIP Conference on Software Architecture*, pages 148–158, 2001.

[84] W. Stallings. *Data and Computer Communications.* Prentice Hall, 1997.

[85] D. Sterne, G. Tally, C.D. McDonell, D. Sherman, D.L. Sames, P.X. Pasturel, and E.J. Sebes. Scalable access control for distributed object systems. In *Proc. of the USENIX Security Symposium,* 14 pages, 1999.

[86] M. Stillerman, C. Marceau, and M. Stillman. Intrusion detection for distributed applications. *Communications of the ACM*, 42(7):62–69, 1999.

[87] Java 2 Platform Enterprise Edition. *Sun Microsystems.* <http://java.sun.com/j2ee/>.

[88] C. Szyperski. *Component Software – Beyond Object Oriented Programming.* Addison Wesley, 1997.

[89] Stefan Tai, Thomas Mikalsen, Eric Wohlstadter, Nirmit Desai, and Isabelle Rouvellou. Transaction policies for service-oriented computing. *Data and Knowledge Engineering Journal: Special Issue on Contract-based Coordination and Collaboration*, 51:59–79, 2004.

[90] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. of the International Conference on Software Engineering*, pages 107–119, 1999.

[91] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. OpenJava: A class-based macro system for Java. In *Proc. of the Workshop on Object-Oriented Reflection and Software Engineering*, pages 117–133, 1999.

[92] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 178–204, 2002.

[93] Peer to Peer. In *Wikipedia.* <http://en.wikipedia.org/wiki/Peer-to-peer>.

[94] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B.N. Jorgensen. Dynamic and selective combination of extensions in component-based applications. In *Proc. of the International Conference on Software Engineering*, pages 233–242, 2001.

[95] A. van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering*, 24(11):908–926, 1998.

[96] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 359–369, 1996.

[97] R. Walker and G. Murphy. Implicit context: Easing software evolution and reuse. In *Proc. of the Symposium on Foundations of Software Engineering*, pages 69–78, 2000.

[98] N. Wang, K. Parameswaran, and D. Schmidt. The design and performance of meta-programming mechanisms for object request broker middleware. In *Proc. of the Conference on Object-Oriented Technologies and Systems*, pages 677–694, 2000.

[99] Web Services Activitiy. *W3C.* <http://www.w3.org/2002/ws/>.

[100] W. Yi-Min and L. Woei-Jyh. COMERA: COM extensible remoting architecture. In *Proc. of the Conference on Object-Oriented Technologies and Systems,* 10 pages, 1998.

[101] W. Yuan, K. Nahrstedt, S. Adve, D. Jones, and R. Kravets. Design and evaluation of a cross-layer adaptation framework for mobile multimedia systems. In *Proc. of the Conference on Multimedia Computing and Networking,* 13 pages, 2003.

[102] P. Zave. An experiment in feature engineering. *Programming Methodology*, pages 353–377, 2003.