DADO: Enhancing middleware to support cross-cutting features in distributed, heterogeneous systems

DRAFT–PLEASE KEEP CHECKING BACK FOR NEW VERSION

Eric Wohlstadter, Stoney Jackson and Premkumar Devanbu
Center for Software Systems Research,
Department of Computer Science,
University of California, Davis, CA 95616
wohlstad,devanbu@cs.ucdavis.edu

Abstract

Some "non-' or "extra-functional" features, such as reliability, security, and tracing, defy modularization mechanisms in programming languages. This makes such features hard to design, implement, and maintain. Implementing such features within a single platform, using a single language, is hard enough. With distributed, heterogeneous ($\mathcal{DH}$) systems, these features induce complex implementations which cross-cut different languages, OSs, and hardware platforms, while still needing to share data and events. Worse still, the precise requirements for such features are often locality-dependent and discovered late (e.g., security policies). The DADO[1] approach helps program cross-cutting features by improving $\mathcal{DH}$ middleware. A DADO service *comprises pairs of* adaplets *which are explicitly modeled in IDL. Adaplets may be implemented in any language compatible with the target application, and attached to stubs and skeletons of application objects in a variety of ways. DADO supports flexible and type-checked interactions (using generated stubs and skeletons) between adaplets and between adaplets and objects. Adaplets can be attached at run-time to an application object. We describe the approach and illustrate its use for several cross-cutting features, including performance monitoring, caching, and security. We also discuss software engineering process, as well as run-time performance implications.*

1 Introduction

This paper is concerned with an approach to supporting the development of late-bound, cross-cutting features in distributed heterogeneous systems.

Cross-cutting features 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, [31, 14, 19, 7] among others; we present a sample security policy in the next section which provides an illustration). 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*: locality dependent, discovered late, and change often—security policies again being a prime example. Programmers are thus confronted with the difficult challenge of making a scattered set of changes to a broad set of modules, often late in the game.

Distributed Heterogeneous systems (abbreviated $\mathcal{DH}$) are becoming part of the IT infra-structure in many organizations: many needed software functions are provided by systems assembled from pieces running on different platforms and programmed in different languages. Distribution arises from pressures such as globalization and mobility. Heterogeneity arises from considerations such as performance, legacy systems, weight, size, vendor specialization, and energy consumption. Cross-cutting features in $\mathcal{DH}$ systems present special challenges. Feature implementations are scattered across different languages, operating systems and hardware platforms. Feature implementation elements in one platform need to correctly exchange information with existing application code, and with such elements on other platforms. Any cross-platform (remote) interactions between feature implementation elements may negatively impact application performance. In a WAN context, the presence of different, incompatible features (*e.g.* different security policies) may even cause the application to fail. In addition, the operator of a service may wish to change security policies at run-time. Some platforms may be too resource-limited or performance-constrained to support some types of software evolution techniques (*e.g.*, reflection). In some cases, source code may not be available for modification, so binary editing techniques (or middleware-based wrapping) might have to be used. However, since feature implementations may cross-cut platforms, all these different techniques of software evolution should be allowed to co-exist, and inter-operate. Finally, since cross-cutting feature implementations might be widely applicable, we would like to reuse them (in either source or binary form,

[1] DADO: Distributed Adaplets for Distributed Objects. We also note that a "dado" is a carpenter's tool for making cuts across the grain.

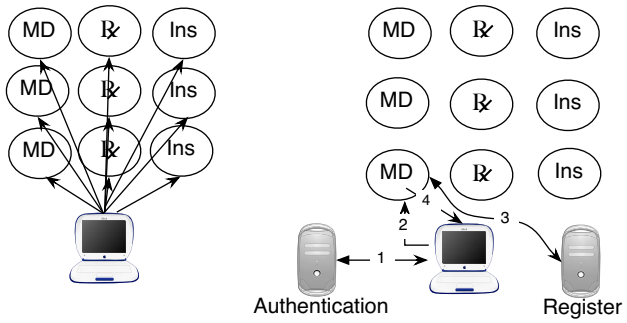

Figure 1. A distributed health-care system, with many service providers, without (*left*) and with (*right*) security. The right one enforces this security policy: the client must first get get an authentication token (1) from an authentication server, and then present this token and his request (2) to a service-provider, who then checks with the registration server (3) (to prevent multiple fraudulent requests) before servicing the request (4). Such a policy requires implementations that cross-cut system and language boundaries.

as applicable) by changing the way they are "bound" to application implementations.

In this paper, we describe 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 remote interaction. The paper begins with a motivating example in Section 2. We then survey the surrounding area in Section 3. Section 4 presents our research goals in more detail. Section 5 describes the current status of our experimental implementation of DADO (including the run-time, code-generation and deployment tools), which is based on the OMG CORBA standard. Section 6 presents some sample applications of DADO. Section 7 presents some micro-benchmarks evaluating the performance impact of DADO. In section 8 we describe closely related projects. Finally we conclude with a overall view of the work, the current limitations, and our future plans.

Note An earlier short position paper (4 pages) published in IWPSE [39] outlined the goals of our research and some initial designs. This paper presents similar motivations, but the designs, examples, and results presented here are new and different.

2 An Example

For expository reasons, we review the example used in an earlier position paper [39]. Consider a $\mathcal{DH}$ medical application (Fig. 1), with a set of clients making use of three groups of servers (shown as groups of circles with indicative labels): clinics, pharmacies, and insurers. The servers in a group could be running on different platforms (each doctor's office might use a different type of computer), but each provides the same service (e.g., through the same CORBA IDL interface). The components in this architecture communicate using $\mathcal{DH}$ middleware. In Fig. 1 (left), the original services are shown. The multiple arrows suggest drug fraud, with an unauthorized impostor client contacting multiple doctors and getting many prescriptions for the same drug, possibly getting each prescription dispensed many times, by different pharmacies, and then issuing multiple fraudulent insurance claims.

Consider injecting a *security* policy into this system, consisting of two critical elements. First, each client must be authenticated by an authentication server (*e.g.,* by a password scheme). Next each client must deal *with only one* server from each category. Thus, each client must use just one doctor (*except* for second opinions!!), one pharmacy, and one insurer within a given time interval. Fig 1 (right side) schematically indicates the new high-level architecture. A authentication server has been added to validate users, and a registration server to register client-service provider relationships.

This policy requires changes to *every component* and to *every interaction* between components. The client now has to authenticate itself to the authentication server, which provides an identity token. This token must now be added to all client-service requests. All members of each group of services must now coordinate among themselves to make sure that a client with a particular identity does not interact with more than one specific member of a group. Since malicious clients may try to induce race conditions among members of a group, they must synchronize to "commit" to serving a client.

The changes are clearly "cross-cutting". 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 function on the server side for doctors, pharmacists and insurers are similar, it would be desirable to reuse the same policy implementation, *even if their IDL interfaces are different*, should the platforms be compatible.

Next, we survey current approaches to $\mathcal{DH}$ evolution, considering how they address programming challenges such as this one.

3 Current Approaches

There are a variety of approaches to dealing with cross-cutting features. Our survey here is limited by space to be representative rather than exhaustive; no judgment of omitted or included work is implied. A more complete survey can be found in [38].

Several *language-based techniques* have been proposed. Classical syntactic program transforms [2] were perhaps among the earliest to provide the capability of broad changes to programs. Reflection [23] provided means of introducing cross-cutting changes at run-time in languages such as Smalltalk. Compile-time [5, 32] reflection in C++ and Java has been developed and extended to load-time in Java using byte code editing [6]. Mixin-layers [28] also provide a way of adding features to methods in several different classes simultaneously. Implicit Context [36] is a method for separating extraneous embedded knowledge (EEK) (or cross-cutting knowledge) from the design of a program, and re-weaving it back in later. Monads and monad transformers [16] have been used in lazy, pure functional

languages to capture cross-cutting features such as states and side-effects. They work by encapsulating the basic notion of a *computation*, and then allowing fundamental evaluation mechanisms such as value propagation to be overridden. Recently, approaches such as HyperJ [31] and Aspectj [14] provide differing approaches to implementing cross-cutting features in Java. A detailed comparison (but see [13] for a comparison of compositional *vs.* aspectual views of program evolution mechanisms) of these differing approaches is beyond the scope of this paper; suffice to say we are interested in a $\mathcal{DH}$ setting, thus transcending language boundaries. While details vary, most of these languages provide two features: a *hook* or pattern, for describing where to insert cross-cutting changes, and then a way to program the changes themselves. Since our approach uses the "hook" mechanism from AspectJ, we discuss it in more detail here.

AspectJ provides a pattern mechanism, called *pointcuts* for capturing groups of events, called *joinpoints* that may occur during a program's operation (such as method calls/receptions, constructor calls, field accesses, and exception events). The pattern-matching mechanism includes regular expression matching, with wild-carding over fragments of method names, argument names, types etc. Extra code, called *advice* can be associated with point-cuts, and is inserted by the AspectJ compiler into the join-points. Advice can inspect and modify data that are available at join-point events (*e.g.* method-call arguments and return values), and can create new data dynamically that is only shared with other advice. Our work uses these ideas for *modeling* cross-cutting changes to distributed systems at the IDL level. However, the distribution, heterogeneity, and versioning problems that arise in our context, require new and different *implementations*.

Middleware-based approaches are certainly relevant. Some works exploit language-based reflection in the middleware [37] and other approaches use specially constructed reflective ORBs [15]. SOM [8] was an early approach to support reflection directly in the middleware. Interceptors [37, 20] and filters [27] provide a way of inserting extra functionality into *every* method that originates or arrives at a request broker; middleware-specific APIs provide means for interceptor code to reflect upon the details of the intercepted invocations. While these reflective methods are suitable for implementing cross-cutting services [4], (and for some very idiosyncratic, and highly dynamic services may be the only way to do it) the use of the low-level reflection APIs, along with the need for frequent use of type-casting makes programming difficult and error-prone; thus it would be preferable to use more statically checkable methods when possible. Proxies and wrappers [12, 29] are another approach. However, they are typically tailored for a specific application object interface; so thus, it would not be possible to reuse a wrapper to implement the same security policy on such entirely different components as doctors and insurers. *Container models* [24, 33] address this problem through code generation. They provide a fixed set of services (depending on the container vendor) to application components. Via configuration files and code-generation, services selected from a given set can be added to any component. However, some services cannot be completely located within the container. Consider that a client may not be willing to reveal his password to just any old application container, and so the initial step of authentication (password based or public-key signature based) might need to occur at a separate location that the client trusts. So the authentication exchange must be custom-programmed using an approach similar to interceptors. Programming here can be thus sometimes as hard as programming interceptor-based services.

Recently, Duclos, Estublier, and Marat [11] have proposed the model of a *Component Virtual Machine*, which captures important events in a component's lifecycle. These events can be viewed as *joinpoints*. An enhanced container implementation allows extra Advice to bound to specific pointcut patterns over these joinpoints. This approach allows much easier implementation of custom services on the container side. We discuss this work in more detail later in § 8; we merely note here that our work focuses more on heterogeneous systems rather than container-based systems. Section 8 also surveys several other closely related works, that are easier to relate to ours after DADO details have been presented.

4. DADO Overview

As illustrated in Section 2, late-bound, cross-cutting functions such as security require extra functional elements (which in DADO we call *adaplets*) to be located together with (potentially distributed) application software components. A client-server pair of adaplets would constitute a distributed DADO service. We begin with a discussion of the main goals of our project. Then we describe the features of DADO that address these challenges.

4.1 Desiderata

Heterogeneity and Communication Adaplets may need to exchange information and co-ordinate with each other, and/or with the application components. While this is strongly analogous to AspectJ, adaplets must communicate and co-ordinate in a *distributed heterogeneous* context. The adaptation mechanisms (source/binary transformation, runtime wrapping) may depend on the platform; even so, heterogeneous adaplets should co-exist and inter-operate correctly.

Binding and Deployment It would be desirable to support *late binding* and *flexible deployment* of DADO services. Consider that container standards such as J2EE allow independent container developers to develop services that are customized for specific applications at deployment time. Likewise, we would like to allow vendors to build services consisting of DADO services, independently of application builders, and then allow deployment experts to combine services and applications to suit their needs.

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

Flexible Communication and Co-ordination The interaction between a matched pair of client and server adaplets may not be simple and monolithic. Under different circumstances, the client adaplet may require and request different functions (with different parameters) that are supported by a server adaplet (just as a distributed object can support several distinct methods). Likewise, the server adaplet may request different post-processing functions on the client side. A client adaplet can refer to it's server "mate" via the reserved name "`that`" (and vice versa). However, for efficiency, it would be better to have only a single invocation event through the middleware (e.g., a single CORBA synchronous call).

4.2 DADO Features

Modeling, Type-Checking, and Marshalling DADO employs an enhanced IDL and code-generation to support the following:

- Explicit IDL-level modeling of adaplets and their interaction with application components.
- Ability to implement adaplets in different languages, while supporting:
- safer interaction (via static type-checking) between adaplets, with automated generation of marshaling code.

Point-cut based Binding DADO separates services (which describe the interfaces supported by adaplets) from a deployment description, which specifies the precise deployment context of a service (using a pointcut language similar to AspectJ). This allows a deployment expert to tune the connection between DADO services and different application components. The binding language is agnostic with respect to the implementation; DADO adaplets could be incorporated into the existing application using static transformations (binary or source) or dynamic wrapping, depending on available tools, performance issues, etc.

Multiple Contextual Invocations DADO allows adaplets 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 DADO clients transparently (without additional programming) discover the services associated with a server, and deploy[2] additional adaplets as needed.

4.3 Process implications of DADO

Currently, the process of building $\mathcal{DH}$ systems using middleware such as CORBA includes modeling the high-level design using IDL. IDL specs are then implemented by developers, be they COTS vendors, or application builders, 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, along with any COTS software and ORB-provided services (naming, lifecycle, events etc.).

DADO brings three new roles into this process (see appendix 4): a *service architect*, *service programmer*, and a *service deployment specialist*. This service architect can design a $\mathcal{DH}$ service that implements a cross-cutting feature, such as the ones illustrated in Section 6. This process begins with a description of a cross-cutting DADO *service* as in an enhanced IDL (known as DAIDL, for DADO IDL). A service is a collection of DADO *adaplet interface* descriptions, which consist of several methods, just like a CORBA IDL interface. These interfaces are then compiled using DAIDL compilers for different target implementation languages (currently we support C++ and Java), producing marshaling routines and typing environments. The implementation then proceeds by service programmers just as with conventional middleware.

The deployment specialist binds an implemented service to a given application by specifying bindings using an AspectJ like pointcut language. The deployment specialist will need to understand both the application and the service, and select the bindings based on the specific installation. Currently, these bindings must be specified ahead of time and pre-compiled; one can then choose from different pre-compiled bindings (each of which bind a service to a set of application objects in a particular way) dynamically[3]. Duclos, Estublier and Marat's DS-CVM [11] also includes similar roles, but their implementation strategy is different, utilizing a sophisticated container architecture (we come back to this later, in § 8).

5 DADO implementation

We now present more details on the DADO features outlined in the above section. The current DADO experimental implementation is based on the OMG CORBA standard. It includes IDL language extensions for services, DADO IDL (DAIDL) compilers for C++ and Java, run-time software extensions for two different ORBs (JacORB and the TAO ORB), and tool support for the deployment of services (*i.e.*, for dynamically inserting DADO services into existing CORBA applications).

5.1 IDLs, Type-checking, and Marshalling

DADO adopts the philosophy (as does DS-CVM [11]) 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 "plumbing" code and typing environments for static type-checking. DADO IDL introduces the notion of a *service* that refers to a cross-cutting feature. A service comprises client and/or server *adaplets*. Each *adaplet* supports several methods, which may be of 2 different kinds. *Advice* methods, identified in DAIDL by the **advice** keyword, may be bound, via pointcut patterns (like AspectJ advice, as explained later, in Section 5.3) to application objects. Advice methods basically provide additional functionality that is run *every time* certain methods defined in an IDL interface are invoked. Advice can be on the client or the server-side.

[2] In Java, with a suitable `classloader`, adaplets could be even dynamically downloaded over the internet.

[3] This suggests another role, perhaps an *operator*, who selects services based on operating conditions.

In addition to advice methods, DAIDL services can also include *request* methods (identified in DAIDL by the **request** keyword). These are a form of queued asynchronous methods (see section 5.4 on RMCI) that may be invoked by any adaplet methods. Advice and request are explained in more detail in Section 5.2.

DAIDL compilers can currently generate heterogeneous typing environments (*i.e.*, C++ header files, or Java imports), as well as stub and skeleton routines; adaplets can currently be implemented in either C++ or Java (but must be written in the same language as the application object[4]). We also note that advice adaplet methods have direct typed access to any argument in the application invocation; the actual bindings are specified in the pointcut. Programming within the context of typed stubs and skeletons, and leveraging generated marshaling and other "plumbing" code offers a distinct software engineering advantage over the current practice of "type-less" programming of late-bound services that use untyped string data in an invocation context object for data exchange.

5.2 Advice and Request

The separation of advice and request operations in the adaplet interfaces represents two levels of adaptation required to implement cross-cutting distributed heterogeneous services. In this section we detail the relationship of advice and requests to the development and runtime execution of standard CORBA components and to each other.

We recall (from Section 4.3) 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 distributed application is desirable, she can add an advice operation to the interface of an adaplet. 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.

Some services can be implemented simply by executing advice on the client- or server-side, along with application method invocations. However, in some cases, additional information may be need to be sent along from the client to the server side adaplet (or vice versa). For example, in section 6.1 we present a service where a client side adaplet can request that a matching server adaplet calculate server processing time for specific invocations, and then communicate this information back to the client adaplet. This additional information conveyed between client and server adaplets is contextual. It must be associated with some original CORBA invocation. Likewise, the timing behavior by the server adaplet must occur before and after the processing of the invocation for which the client adaplet requested statistics. This type of adaptation is handled by the RMCI mechanism described below, in Section 5.4.

[4]This is primarily for performance reasons; if adaplets are in a different language, it would be necessary to go through middleware to get from an application object to an adaplet. With a "polyglot" middleware like .NET's common language runtime, this problem can be finessed to some extent.

The service architect can include operations tagged with the `request` modifier keyword to provide an extra communication path between client and server adaplets that is associated with the current CORBA invocation. The body of client and server advice can be programmed to add request messages by using the "`that`" reference which exposes the interface of request operations available to a client adaplet by the server adaplet and vice versa. In object-oriented languages the service programmer will derive adaplet implementations from a generated abstract base class which includes an appropriately typed member variable named "`that`". "`that`" is automatically bound to a generated stub that implements RMCI semantics for each request operation.

Advice and Request play different roles in adapting the dynamic execution of a distributed application. Advice operations are used to add behavior 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 adaplets or server objects and server adaplets only. The connection between client adaplets and server adaplets 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.

The exact mechanism by which the original client and server programs are modified can be platform-dependent; heterogeneity is allowed. Several options are possible, including source-code weaving, generating customized stub components, or modifying the middleware. The transmission format of request messages, however, is standardized because it must be understood by the DADO runtime on heterogeneous hosts. Our experimental implementation relies on packing request messages into the `ServiceContext` of a CORBA invocation. The `ServiceContext` is part of the CORBA protocol format for communicating invocation specific information between ORBs. Naturally, the generated request and advice skeletons, and typing environments are standardized, using the usual OMG IDL language mappings.

5.3 Binding and Deployment

Once built, a service can be integrated with applications by specifying a *binding*, which is done using a pointcut language. This process involves one platform-*independent* tool, which matches the pointcuts against a known set of component interfaces, and produces a digested match-table in XML format; and a separate platform-*dependent* means for actually ensuring that the adaplets get triggered when the pointcuts get activated.

The pointcut language extends the AspectJ pointcut language to specify client or server side pointcuts, extending the AspectJ regular expression syntax for the declaration of generic or cross-cutting behavior. Matching of pointcuts with invocations could be done off-line or on-line. The current DADO tool (the pointcuts pre-processor) matches pointcuts (against the IDLs of the application objects) at compile-time. This tool identifies all the IDL level events requiring adaplet intervention, and also the in-

formation in the events that should be made available to each adaplet. The output of the preprocessor is a representation of all the event/action matches as an AST (represented as XML). We call this Intermediate Joinpoint Representation (IJR). (*Note* Although this matching happens at compile-time, services with pointcuts that are already compiled into IJR can be added or removed at runtime). Of course, future tool (and associated runtime) support could allow new point cuts to be created and inserted at run-time.

In order to trigger adaplet 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 [5, 32, 14, 25]. Middleware, also, can support highly dynamic reflective mechanisms [3]; Duclos, Estublier and Morat [11] have build a "component virtual machine" that allows great flexibility in instrumentation.

In keeping with the $\mathcal{DH}$ philosophy, *we allow 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. Currently, several instrumentation mechanisms are supported for translating DADO pointcuts (in IJR form) into actual trigger mechanisms. For Java, we use AspectJ [14] to insert the necessary trigger code into generated stubs and skeletons (thus avoiding the need for application implementation source code. For C++, we make use of a range of mechanisms, including TAO's smart proxies [37], and the Portable Interceptor standard in CORBA; this approach is also compatible with binary-only application components, and can work in any language, even one that does not support source-code or binary instrumentation mechanisms. However, both approaches require that adaplets be written in the same language as the application objects. Removing this restriction is certainly possible, but would require adaplets to engage a large segment of the middleware stack for cross-language interoperability with application components. Naturally, client- and server-side adaplets, *even if using different languages, different instrumentation mechanisms*, are *fully inter-operable, and portable*. The adaplet programmer remains agnostic with respect to the actual instrumentation mechanism that is used to trigger the adaplet. An adaplet can communicate with other adaplets (the matched one, or any others that it has a handle to) using the DAIDL interface description. In our implementation, this is accomplished through appropriate code generation; the generated code pumps data around by packaging it into the untyped *service context* object (See [21], Chapter 21) API in CORBA.

5.4 Remote Multiple Contextual Invocation

Remote multiple contextual invocation (RMCI) in DADO gives service developers more ways of programming interactions between client-side and server-side adaplets. Consider that a client adaplet may require different types of actions to be taken at the server side. As a very simple example, a per-use payment service adaplet 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 pass-

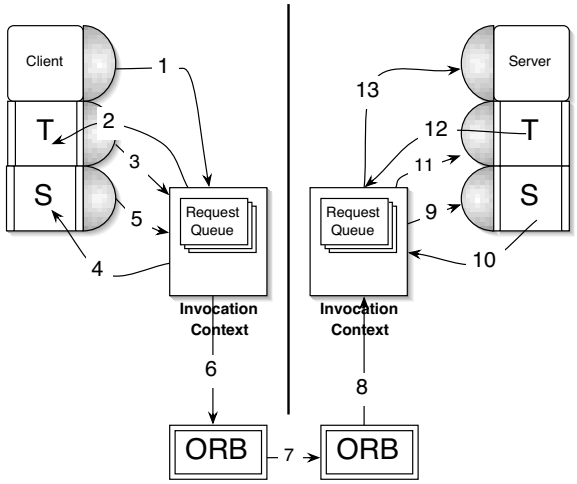

Figure 2. *Remote Multiple Contextual Invocation* Client-server application object interactions are mediated by "T" (transaction) and "S" security adaplets. Gray semi-circles denote generated marshaling code. Initial client invocation (1) is diverted by the interceptor in turn to each adaplet (2,4) until finally arriving (6) at the ORB. Adaplets use marshaling code for their invocations (3,5). Each adaplet may enqueue several one-way messages (for the server-side adaplets) which are piggy-backed as a request queue through the normal middleware invocation (7) over the WAN to the server side. The process occurs in reverse on the server side, with the requests in the queue being delivered to the corresponding adaplets. Likewise, server-side adaplets may enqueue messages to the client side adaplets which are piggy-backed on the invocation response.

word. We could include both options as possible parameters, in a single method signature, along with an extra flag to indicate the active choice; this leads to poorly modularized methods with many arguments. Rather, we take the "distributed object" philosophy of supporting different requests at a single server object; we allow adaplets on either side to support several different requests. As another illustration of the use of requests, consider a generic caching service, (implemented using DADO adaplets) which can for example be attached to a stock quotation server(this example is discussed in more detail in Section 6). Client-side advice can cache values and return them instead of going to the server for each request. However, the server may want to communicate a "time-out" interval back to the client, so that it can adjust the time-out period for cached quotes based on market volatility. So it would be useful to have a special client-side request method that the server can invoke when it needs to adjust the time-out value.

DADO adaplets support a special type of 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 for dispatch at a server side adaplet. This also works in reverse for requests going from the server-side adaplet to the client-side adaplet. The keyword "`request`" in the DAIDL adaplet inter-

face can be used to designate operations as having RMCI semantics.

In figure 2 we show application objects using both a security and a transaction service. Note the presence of corresponding adaplets for each service on both the client and server side. Adaplets might include both advice and request methods; the figure illustrates how the client side advice gets executed in turn. Each client adaplet may enqueue multiple requests to be executed by the server side adaplets. The requests are collected into a queue that gets piggy-backed onto the regular middleware invocation and passed through to the server side. The RMCI designation thus arises from **M**ultiple **R**emote requests contained within the *single* **I***nvocation* **C***ontext*. The implementations of these requests (regardless of adaplet's location) have full reflective access to the current active invocation, via provided APIs. Of course, if the information needed by the adaplet is known statically, there is no need to use reflection.

On the server side, the designated advice adaplet methods for each adaplet get executed, as are the enqueued requests. The server side adaplets may also enqueue requests to be executed by the client side. This feature can be used to pass information back to client-side adaplets; we illustrate with a performance monitoring example where server-side time-stamps are passed back to the client via a request adaplet method.

In essence, RMCI provides a form of dynamic per-invocation adaptation as in Lasange[34] while supporting type-checked interactions and modular design through IDL declaration.

5.5 Transparent Late Service Binding

In a WAN environment such as the internet, where servers are discovered at run-time, 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 on only on type information cannot easily provide this kind of late binding.

When server objects are associated with a DADO service (this can happen dynamically, from the command-line or at deployment time via configuration files) they are assigned an external object reference that is used by the client side run-time to detect the applicable services[5]. Essentially, the references encode information about the adaplets associated with this object[6]. This information is used by the Dado interception logic on the client-side to transparently engage the corresponding client-side adaplets. Our implementations use different triggering mechanisms, depending on the platform, to achieve this.This process is illustrated in figure 3. When an application object implementation 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. It then instructs the local factory to create instances of the corresponding client-side adaplets, and injects them into the execution path of invocations originating from the client.

[5]We assume that these external object references uniquely identify a server object.

[6]Using CORBA this is possible with Tagged IOR Components. Other middleware such as SOAP could add information to a URL.

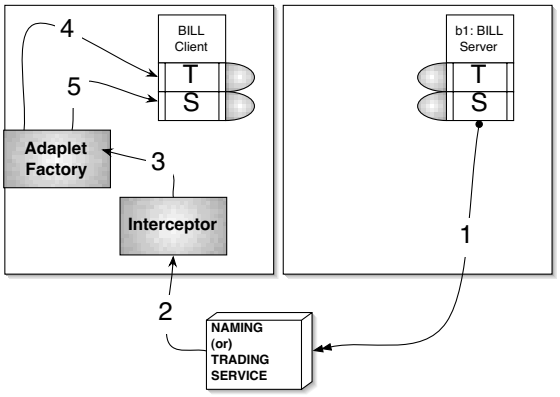

Figure 3. *Late-binding service adaptations* (1) Server object, with Security and Transaction adaplets, named "`b1`" of type "`Bill`" is registered with a Naming service. The identifiers "Transaction" and "Security" are tagged to the external object reference. When client looks up object named "`b1`",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

If the service deployment at a server object changes dynamically, it re-registers with the naming service to alert future clients. If service deployment changes at a server while current clients are still active, the server can throw a DADO-specific exception upon their next invocation; the client-side DADO runtime transparently responds, reconstructing the set of client-side adaplets so that interactions may continue.

5.6 Adaplet Instance Considerations

A service developer may need to control the granularity of how server objects and clients are affected by adaplets. This may be necessary to conserve resources, by not creating too many adaplet instances, or for associating state in adaplets with particular application object instances. Currently, we provide a mechanism for associating adaplets on a one-per-ORB basis (*i.e.,* a singleton) or on a one-per-POA[7] basis. We plan to add support for per-object adaplets in the future; currently, per-object adaplet instantiation is only possible by placing objects in separate POA's.

6 Examples

We now present some sample applications of DADO. All of these examples have been implemented with a Java client on JacORB and a server on TAO. For space reasons only the DAIDL interface descriptions are presented.

6.1 Round-Trip Performance

Consider a simple performance monitor in a particular client software. One can easily write code (*e.g.,* using interceptors, see [20], or [21], Chap. 21) to attach to the client that will record

[7]The Portable Object Adapter is a container abstraction available in CORBA for associating policies (such as thread policies) with a number of server objects.

the time each invocation leaves and response arrives. But 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 cohesion between interacting client and server interceptors. This service requires three critical elements: clients must be able to ask the server for timing statistics for *some*, not all, invocations. Servers must return data through a type-checked interface. Clients need some way to modify existing software to add logic for requesting timing statistics; different means should be allowable. Finally, client- and server- adaptations should be coordinated; clients will not request timing statistics from servers unable to provide them.

```
adaplet Timing {
  client {
    advice void timedOperation();
    request timeResult(in long long received,
                       in long long sent);
  };

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

```
crosscut Timing {
  client {
    before call(*) :
          void timedOperation();
  };
};
```

DADO service developers first write the DAIDL interfaces (above the line) of the client and server adaplets, and implement them for target languages (utilizing DAIDL tools and run-time libraries). The client adaplet has two methods. One, `timedOperation` is an `advice` method that can be bound to an application method. The other, `timeResult` is a `request` method that is used by the server-side adaplet to report back the timing results. This timeResult `request` message can be included with the original response by the server and will be dispatched to the client side adaplet before the client application receives the response. The implementation of `timedOperation` can dynamically decide whether to invoke `that.timeRequest` or not; we note again that the special variable `that`, denoting the (other) matched adaplet is implicitly made available to both client/server adaplets. When `that.timeRequest` is invoked by the client side advice a request message is added to the original invocation and dispatched to the server side adaplet before the server object receives the original invocation. The server adaplet can respond to a `timeRequest` by taking two timing measurements to determine the actual execution time for that application method invocation; it can then report the result back to the client, using the `that.timeResult` client-side request. We note that *the implementation of the advice is responsible for invoking the request*; there is thus no explicit modeling of this detail at the IDL level. For instance, consider a client that would like to time one out of every ten invocations. This logic could be programmed into the `timedOperation` advice by an adaplet programmer. It would be inappropriate to introduce this type of implementation detail at the IDL level. Thus the service programmer decides when `request` messages are triggered; however, the IDL model does allow `request` messages to be marshalled and triggered in a heterogeneous, yet type-checked manner.

To deploy `Timing` adaplets for a given application object, the server-side would make the service available by registering a server `Timing` adaplet component with the servers' object adapter. When clients become aware of those server objects, the DADO run-time will automatically deploy client adaplets based on the clients deployment preferences (see the `crosscut` declaration above). In this deployment the client would like all invocations to be intercepted by `timedOperation` as indicated by the wildcard. The server side doesn't need to specify any additional pointcut instructions, as the operation `timeRequest` is invoked by the client-side adaplet.

6.2 Client-Side Cache

Systems are often built without performance optimizations such as caching in mind. However, it would be nice to leverage some off-the-shelf caching behavior, without requiring extensive modifications to client and server code. We consider a feature whereby clients can cache data associated with a particular server. Consider a stock-quote server, which provides accessor and mutator methods. The accessor methods are called by clients, and mutators would be called by a data provider to "pump" data into the quote server. We would like to cache the returned quote value at the client side. When the server returns data it associates a time-to-live (TTL) value with the data, for use by the client. An invocation will be serviced using cached data from the client (without contacting the server) as long as the TTL has not expired. The server will adjust its TTL value heuristically depending on the frequency of calls (from its data provider) to its mutator method.

```
adaplet Cache {
   server {
      request requestTTL();
      advice void trackWrite(in string key);
   };
   client {
      advice any_type readcache(in string key);
      request putTTL(in long ttl);
   };};
```

The DAIDL interface for the Cache adaplet specifies two operations, `trackWrite` and `requestTTL`. The client-side cache adaplet issues `requestTTL` along with accessor operations for which it has no cached data. The server adaplet sends back the TTL value associated with the invocation, back along with the data response; it does this by issuing a `putTTL request` to the client. The server adaplet estimates the TTL values heuristically by timing the mutator operations that it receives from its stock quote providers. The `trackWrite` is the `advice` that is triggered to calculate TTL based on mutate operation frequency. The client-side advice `readcache` performs the caching operation. The keyword `any_type` gives `readcache` access to the return value, as a generic CORBA `any`, of the operations on which it is deployed. This design requires that the operation to be

cached, the accessor operation, uses a string "unique key" argument to determine the returned data. This fits our scenerio where clients access stock quote prices based on stock market symbols but may require a different interface for other applications. Consider a simple `StockQuotes` server with operations `setQ` and `getQ`. We could apply the adaplet `advice` hooks to introduce caching using the following deployment file ,

```
crosscut Cache {

   client {
      around call(float StockQuotes::getQ(key)) :
               any_type readcache(in string key);
   };
   server {
      before call(void StockQuotes::setQ(key,in float)) :
              void trackWrites(in string key);
   };

};
```

The *key* arguments serve to match parameters in the application operations with parameters for use by the advice.

6.3 Security Policy

Now consider our security policy example (Section 2). Here, servers must restrict access to some operations, based both on clients' identity, and their previous history of use. Clients must be registered with a particular server for some duration. After this time has expired, clients can register with another server. This prevents clients from contacting and obtaining the same services fraudulently from multiple servers. Clients authenticate themselves using a cryptographic token. It is the clients responsibility to obtain an `Authentic::Token` which is a cryptographic object representing the verification of the clients identity. The implementation of the `contactAuthentic` advice in the client adaplet is responsible for this; this advice can be bound to application methods that must be mediated by this security policy.

Since the authentication token is specific to this service, we must include server-side request operations in the adaplet to transmit this information to be server side. The server adaplet has two request operations available for receiving the authentication information. The first time a client contacts a server, she must register (commit) to that server for a specified time interval. The `request` operation `register` is for use by first time clients and includes a parameter for the duration of registration. The implementation of this registration will validate the authentication token and check with a centralized reservation server (not shown here) to make sure that the client isn't fraudulently registered with a different service provider.

For subsequent application object invocations within the registered duration, the client uses the second `request` operation `authenticInfo`, to transmit the authentication token. Now, every server operation that needs to be mediated by this security policy must trigger the policy enforcement mechanism, to check that the client is authentic, registered and still within the registration interval. The server operations that need to be restricted should be mapped to the `Wall` server adaplet operation `check` with a deployment description(see below). An implementation of this advice can implement the policy.

```
adaplet Wall {
  server {
    request authenticInfo(in Authentic::Token tok);
    request register(in Authentic::Token tok,
                     in long duration);
    advice void check()
                     raises(NotRegistered,NoAccess);
  };
  client {
    advice contactAuthentic();
  };
};
```

The `check` implementation (in some host programming language) will contact a registration server (IDL not shown) to ensure that a client can be served. If not, (if authentication token, or registration is invalid) the operation should raise the `NoAccess` exception[8]. However, if this client is not registered with any of the servers in the group, or if the required registration interval with the current server has passed, `check` can throw a `Not Registered` exception. This gives the implementation of `contactAuthentic` a chance to catch this exception and retry the operation by sending along a request for a new registration interval (which will succeed as long as the client hasn't fraudulently registered elsewhere). Catching exceptions and retrying operations is made possible by structuring advice *around* original invocations in a fashion similiar to the Decorator pattern.

The server side advice, `check` , and the client-side advice , `contactAuthentic`, can be bound to any operation that must be policy-mediated. Implementations can choose to cache security information as appropriate. However, we note that once the server has implemented and deployed the `check` advice correctly it does not have to trust the client-side advice at all; also, if the authentication service uses public-key authentication, the client-side advice does not have to leak any authentication secrets (e.g., private key) outside the client's machine.

All examples above have been implemented with the client in Java, and the server in C++. Further details of our current implementation are described in the following section. Full source code is available on `http://rickshaw.cs.ucdavis.edu`.

7. Performance Study

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 "plumbing" code (generated by DADO compilers), as well as other DADO runtime machinery for dispatching adaplet `advice` and `request`. For this reason, we use dummy `advice` and `request` methods that don't do any computation, so that we can focus primarily on the actual overhead of the DADO runtime machinery.

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

[8]These are implemented as a run-time `CORBA::SystemExceptions` because they occur inside of the original target operation.

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. Client-side advice is invoked using modified stubs; a portable interceptor dispatches requests on the server side. The DAIDL interface to the adaplet used for performance measurement is shown below. The actual IDL interface that is bound to is not important, since we are actually just measuring the additional overhead of the adaplet run-time infrastructure; in this case, we use a simple interface with a single, synchronous method that takes a string and doesn't return anything (not shown here).

```
adaplet Test{
  client {
    advice void grabArg(in string arg);
  };

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

```
crosscut Test {
  client {
    before call(* *::*(arg)) :
          void grabArg(in string arg);
  };
};
```

As can be seen above, there is one client-side advice and one server side request. The client-side advice is bound to every method call (with a single argument of type `string`) on every object 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 the string argument. The server side advice receives the string argument, and simply just passes control to the server application object. So the overhead we are measuring (beyond the normal CORBA invocation overhead) includes the additional cost of 1) intercepting the invocation on the client-side, 2) dispatching the client-side advice, 3) executing the client-side request stub, 4) marshaling the additional data transmitted by the request into the `ServiceContext object`, 5) transmitting the additional data over the wire 6) unmarshaling the data on the server side 7) dispatching and executing the request implementation on the server side. All measurements given above 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	Wireless
1. Vanilla CORBA	0.65	3.49
2. with 1 advice, 1 request	1	4.17
3. with 10 advice, No request	0.68	3.65
4. with 10 advice, 10 request	1.52	7.45
5. Vanilla CORBA with equivalent raw data Payload for 10 requests	1.38	7.27

The first row is the plain unloaded CORBA call, as a baseline for comparison. The second row is a CORBA call with one adaplet advice, and one additional request. In the third row, we show the effect of "artificially" forcing a dummy advice (that doesn't transmit any requests) to execute 10 times. The fourth row shows the effect of executing the advice shown on the second row 10 times, thus forcing 10 request messages. The critical *fifth* row shows an interesting comparison: it measures the plain CORBA call, with additional data loaded into the service context object, *exactly equivalent to 10 request messages*, without any adaplet code whatsoever. This row corresponds to the precise straw-man comparison for sending data *sans* DADO, and corresponds to the way interceptor-based services (such as Transactions and Real-Time, as per [21], 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's about 5% in both cases for 10 advice invocations) The overhead for sending requests is largely due to the base cost of transmitting data over the service context object. By comparing the 100-Base-T and Wireless measurements one can see the diminishing cost of marshaling as the benefits of reduced latency from piggybacked requests increases. The motivation of RMCI is to provide type-checkable interactions and modularization of service features, we feel these measurements show the feasibility of this approach.

8. Closely Related Work

In this section, we discuss closely related work and compare them in greater detail with DADO.

Dassault Systèmes CVM The Dassault Systèmes Component Virtual Machine (DS CVM) [11] is targeted at container-based systems, and allows the implementation of custom container services. The DSCVM comprises an efficient, flexible CVM that essentially supports a meta-object protocol which can be used for instrumentation of middleware-mediated events. This allows this CVM to support the triggering of advice when the CVM executes specific events such component method invocations. Pointcut "trigger" specifications are implemented using the DSCVM events. Advice can be bound to patterns of these events, and thus be used to implement services.

DADO is complementary to DS CVM in that DADO allows elements of cross-cutting services to be placed on the client site in a coordinated manner, for reasons argued earlier. DADO also operates outside of a component/container model in "barebones" CORBA; thus it must (and does) allow heterogeneity in the implementation of triggering mechanisms such as source transformations, binary editing etc. (See Section 5.3). The heterogeneity assumption also influences our design of type-checked information exchange between client and server adaplets, using generated stubs and skeletons (Section 5.1).

QuO The Quality of Objects (QuO) [17, 35] project aims to provide consistent availability and performance guarantees for

distributed objects in the face of limited or unreliable computation and network resources. QuO introduces the notion of a "system condition", which is a user-definable measure of the system, such as load, network delay etc. System conditions can transition between "operating region"s which are monitored by the run-time environments. The novelty in QuO is that adaptations can be conditionally run to respond not only to normal middleware events, but also to region transitions. This is useful for services that deal with performance.

QuO's version of adaplets are confined to a single system. Unlike DADO, Quo provides no special support for communicating information from a client-side adaplet to a server-side adaplet.

Lasagne Lasagne[34] is a framework for dynamic and selective combination of extensions in component based applications. Each component can be wrapped with a set of decorators to refine the interaction behavior with other components. Every decorator layer is tagged with an extension identifier. Clients can dynamically request servers to use different sets of decorators at run-time. An innovative aspect of Lasagne is the usage of extension identifiers to consistently turn on and off adaptive behavior. However, the use of the decorator-style constrains all extensions to have the same interface. Any additional extension-specific information must be communicated using a "context" object, without the benefits of typechecking or automated marshaling.

Software Architecture In software architecture, connectors [18, 22, 1] 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 cross-cutting concerns. Implementations of architectural connectors have also been proposed [10, 26, 9, 30]. Some of these provide specific services [26, 9, 30] over $\mathcal{DH}$ 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 DAIDL language and compiler allow service builders to write client and server adaplets that provide many kinds of "connector-style" functionality, while the DAIDL "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 adaplets. The question as to whether connector specifications (*e.g.,* in an ADL) can be translated to DAIDL specifications and pointcuts is interesting, and we hope to address it in future research.

9. Conclusion

We conclude here with several observations about DADO, it's limitations, and our plans for future work.

"Client-Server." First, we note that when we repeatedly discuss *"client-"* and *"server-"* adaplets, we are speaking of *client-server roles in a synchronous RPC-style connector!* Thus DADO is not specific to a client-server architectural style. In fact DADO adaplets may be bound to CORBA `oneway` calls, which are essentially asynchronous messages.

Design Choices: The design space of a convenient framework to implement $\mathcal{DH}$ cross-cutting services is quite large, comprising many dimensions such as synchronization mechanisms, scope of data, and the handling of exceptions. The current implementation of DAIDL 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 adaplets; services whose scope transcends a matched stub-skeleton adaplets; other (*e.g.*, synchronous) interactions between adaplets. We would like to implement an adaplet-per-object instance policy as well. Currently , only run-time exceptions are supported for adaplets—in the Java mapping, better static checking would be desirable.

Implementation Limitations. Currently our implementation has some limits. As outlined earlier, the marshaling needs further optimization. The Portable Interceptor approach to trigger advice prevents the modification of invocation arguments or return values; thus non-orthogonal [14] services that do affect these values must be programmed with source or binary transforms. We need to broaden our base to more languages. .NET is currently not supported, but it could be interesting. CLR [33] would allow us to write adaplets for CLR applications in any CLR compliant language.

Service interactions. Feature interactions are a difficult open research issue that DADO services must deal with eventually. We note here that it is currently possible to program interactions between two DADO services: one can write a third service that pointcuts adaplets 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 future work.

In conclusion, DADO is an approach to programming cross-cutting concerns in distributed heterogeneous systems based on placing "adaplets" at the points where the application interacts with the middleware. It supports heterogeneous implementation and triggering of adaplets, allows client- and server- adaplets to communicate in a type-checked environment using automated marshaling, provides flexibility in communication between adaplets, allows flexible binding, and late deployment of adaplets on to application objects. While much work remains to be done, we believe that the current version of DADO provides many features of interest to the software engineering research community. Source code for Java using JacORB and C++ using TAO with MSVC++ is available on-line at http://rickshaw.cs.ucdavis.edu.

Acknowledgments: We would like to thanks the anonymous reviewers for their detailed comments as well as support from NSF 0204348.

References

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

[2] B. Balzer. Transformational implementation: An example. *IEEE Transactions on Software Engineering*, 7(1):3–14, January 1981.

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

[4] L. Capra, W. Emmerich, and C. Mascolo. Reflective middleware solutions for context-aware applications. *Lecture Notes in Computer Science*, 2192:126–??, 2001.

[5] S. Chiba. A metaobject protocol for C++. In *Proceedings, ACM OOPSLA*, pages 285–299, 1995.

[6] S. Chiba. Load-time structural reflection in Java. *Lecture Notes in Computer Science*, 1850:313–??, 2000.

[7] 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 *Proceedings, ECOOP Aspect-Oriented Programming Workshop*, 2001.

[8] N. Coskun and R. Sessions. Class objects in som. *IBM Personal Systems Developer*, Summer 1992.

[9] E. M. Dashofy, N. Medvidovic, and R. N. Taylor. Using off-the-shelf middleware to implement connectors in distributed architectures. In *Proceedings, ICSE-24*, 1999.

[10] S. Ducasse and T. Richner. Executable connectors: towards reusable design elements. In *Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT symposium on Software engineering*, pages 483–499. Springer-Verlag New York, Inc., 1997.

[11] F. Duclos, J. Estublier, and P. Morat. Describing and using non functional aspects in component based applications. In *International Conference on Aspect-Oriented Software Development*, 2002.

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

[13] W. Harrison, H. Ossher, and P. Tarr. Symmetrically and assymmetrically organized paradigms of program transformation, 2202.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[15] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, and R. H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143, New York, April 2000. Springer-Verlag.

[16] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press.

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

[18] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering*, pages 178–187. ACM Press, 2000.

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

[20] P. Narasimhan, L. Moser, and P. Mellior-Smith. Using interceptors to enhance CORBA. *IEEE Computer*, July 1999.

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

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

[23] F. Rivard. Smalltalk: a reflective language. In *Proceedings, Reflection*, 96.

[24] E. Roman, S. Ambler, and T. Jewell. *Mastering Enterprise JavaBeans*. Wiley, 2001.

[25] T. Romer, G. V. D. Lee, A. Wolman, W. Wong, H. Levy, B. N. Bershad, and J. B. Chen. Instrumentation and optimization of Win32/Intel executables using etch. In *Usenix Windows NT Workshop*, pages 1–8, 1997.

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

[27] J. Siegel. *CORBA 3 Fundamentals and Programming*. Wiley Press, 2000.

[28] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. *Lecture Notes in Computer Science*, 1445:550–??, 1998.

[29] T. S. Souder and S. Mancoridis. A tool for securely integrating legacy systems into a distributed environment. In *Working Conference on Reverse Engineering*, pages 47–55, 1999.

[30] B. Spitznagel and D. Garlan. A compositional approach to constructing connectors. In *Proceedings, IFIP/IEEE WICSA*, 2001.

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

[32] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. Openjava: A class-based macro system for java. In *OORaSE*, pages 117–133, 1999.

[33] A. Troelsen. *C# and the .NET Platform*. Apress, 2001.

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

[35] R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, and D. Bakken. Quo's runtime support for quality of service in distributed objects. In *International Conference on Distributed Systems Platforms and Open Distributed Processing*, 1998.

[36] R. J. Walker and G. C. Murphy. Implicit context: easing software evolution and reuse. In *Foundations of Software Engineering*, pages 69–78, 2000.

[37] N. Wang, K. Parameswaran, and D. Schmidt. The design and performance of meta-programming mechanisms for object request broker middleware, 2000.

[38] E. Wohlstadter. Managing evolution in distributed heterogenous systems.

[39] E. Wohlstadter, B. Toone, and P. Devanbu. A framework for flexible evolution in distributed heterogeneous systems. In *International Workshop on Principles of Software Evolution (4 pages)*, 2002.

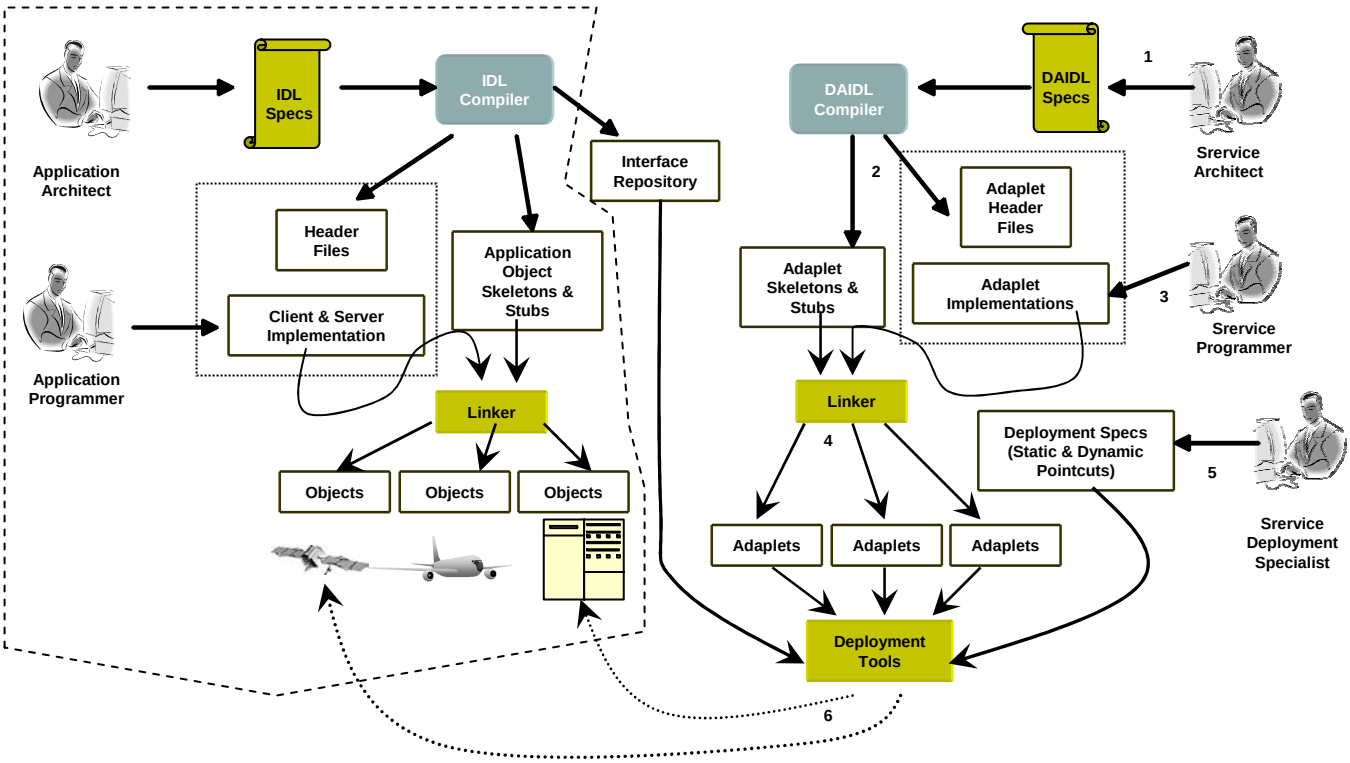

Figure 4. *DADO Development Process.* The left hand side (within the dotted lines) indicates the conventional CORBA process. On the right, the DADO service development begins (1) with modeling the interfaces to DADO adaplets using DAIDL; from this the DAIDL compiler generates (2) plumbing code, and typing contexts for adaplet implementations. The programmer writes (3) the adaplet implementations and links to get (4) the adaplets. Now, the development specialist produces (5) deployment specs, and these are used by deployment tools to install (6) the adaplets at the proper application object locations. Deployment can occur at compile time, link time, or run-time, depending on the instrumentation technology used (only run-time insertion is illustated in the figure).