Generic Middleware Substrate through Modelware*

Charles Zhang, Dapeng Gao and Hans-Arno Jacobsen

University of Toronto
{czhang,gilbert,jacobsen}@eecg.toronto.edu

Abstract. Conventional middleware architectures suffer from insufficient module-level reusability and the ability to adapt in face of functionality evolution and diversification. To overcome these deficiencies, we propose the Modelware methodology adopting the Model Driven Architecture (MDA) approach and aspect oriented programming (AOP). We advocate the use of models and views to separate intrinsic functionalities of middleware from extrinsic ones. This separation effectively lowers the concern density per component and fosters the coherence and the reuse of the components of middleware architectures. Comparing to the conventionally designed version, Modelware improves benchmark performances by as much as 40% through architectural optimizations. Our evaluation also shows that Modelware considerably reduces coding efforts in supporting the functional evolution of middleware and dramatically different application domains.

1 Introduction

The construction of system software such as middleware is complex. A contributing factor to this complexity, as we have observed first hand, is that the code-level design reusability in conventional middleware architectures is incapable of adequately dealing with "change" in two dimensions: *time* (functional evolution) and *space* (functional diversification).

The reusability in conventionally developed software components is insufficient due to the lack of explicit means to effectively distinguish *intrinsic* and *extrinsic* architectural elements. Borrowing terms from subject-oriented programming [10], we use the term "intrinsic" to characterize middleware architectural elements that are essential, invariant, and repeatedly used despite the variations of the application domains. These "common abstractions" are typically pattern-based designs, such as proxy, forwarder-receiver [7], and acceptor [16]. Contrarily, we use the term "extrinsic" to denote elements that are vulnerable to refinements or can become optional when the application domains change. A simple example of an "extrinsic" property is "thread-level concurrency," including patterns

* In Middleware 2005: ACM/IFIP/USENIX 6th International Middleware Conference,November 28th - December 2nd, 2005 Grenoble, France. This research has been supported in part by an NSERC grant and in part by an IBM CAS fellowship for the first author. The authors are very grateful for this support.

such as leader/follower [16], which can become redundant when threading policies are controlled by user applications or if the underlying platform, such as Java Card[1], does not support threads at all. As we have reported in our previous work [23], "intrinsic" and "extrinsic" properties interact non-modularly in conventional middleware architectures. Consequently, middleware architects are faced with immense architectural complexities because the concern density per-module is high. The code-level reusability of the "common abstractions"is also drastically reduced because the generality of intrinsic components is restricted by the "extrinsic" properties in face of domain variations.

Conventional middleware architectures also lack effective means to reuse "extrinsic" properties, especially ones that are crosscutting [13] in nature, i.e., not localized within modular boundaries. We illustrate this problem through the example of *data marshalling*: a major CORBA feature converting the "typed" application data to an array of bytes. We study three popular implementations of CORBA, namely ORBacus [2], a commercial ORB, JacORB [3], an open source ORB, and Sun's ORB, shipped with every Java2.0 SDK. Figure 1 lists the implementations of the marshalling of the data type `long`. These three independent implementations are nearly identical in terms of structure and algorithm. Two design concepts are reused by all of the implementors: the "*buffer*", holding a byte array representing the raw data, and the "*shifting and masking*" algorithm for decomposing four bytes of a `long` value into four byte values. The desired approach is to package this marshalling functionality for type `long`, along with the about 20 other data types in CORBA, as part of a marshalling library, so that it becomes a reusable development artifact. Conventional architectures have fallen short of doing so because they are incapable of componentizing and reusing crosscutting concerns as analyzed in our previous work [24]. Our investigation has revealed similar problems with many other major CORBA functionalities. Being able to componentize and to reuse these functionalities tremendously facilitates the construction of middleware systems.

To tackle the afore-mentioned problems, we propose a new architectural paradigm, Modelware, which embodies the "multi-viewpoints" [14] approach. We capture "intrinsic properties", or common abstractions, in a *base view* consisting of a set of coherent components free of crosscutting concerns. We use *role-based aspect views* and aspect libraries to capture "extrinsic properties", i.e., domain variations. we adopt the Model Driven Architecture (MDA) [4] in both types of views as the vehicle for the mapping abstractions to implementations. Concrete middleware instances can be produced by the *realization* process: selecting the implementations of the abstractions in both kinds of views, and the *projection* process: creating ontological relationships between the elements in aspect views and those in the base view.

---

[1] Java Card. `http://java.sun.com/products/javacard/index.jsp`
[2] ORBacus. `http://www.iona.com/orbacus`
[3] JacORB. `http://www.jacorb.org`
[4] MDA. `http://www.omg.org/mda`

JacORB:

```
private final static void _write4int
   (final byte[] buf, final int _pos, final int value) {
     buf[_pos]   = (byte)((value >> 24) & 0xFF);
     buf[_pos+1] = (byte)((value >> 16) & 0xFF);
     buf[_pos+2] = (byte)((value >>  8) & 0xFF);
     buf[_pos+3] = (byte) (value        & 0xFF);
}
```

Sun CORBA:

```
private final void writeBigEndianLong(int x) {
     bbwi.buf[bbwi.index++] = (byte)((x >>> 24) & 0xFF);
     bbwi.buf[bbwi.index++] = (byte)((x >>> 16) & 0xFF);
     bbwi.buf[bbwi.index++] = (byte)((x >>> 8) & 0xFF);
     bbwi.buf[bbwi.index++] = (byte)(x & 0xFF);
}
```

ORBacus:

```
public void write_long( int value) {
     buf_.data_.[buf_.pos_++] = (byte) (value >>> 24);
     buf_.data_.[buf_.pos_++] = (byte) (value >>> 16);
     buf_.data_.[buf_.pos_++] = (byte) (value >>> 8);
     buf_.data_.[buf_.pos_++] = (byte) value;
}
```

Fig. 1. Data marshaling of type `long` in A:JacORB, B:Sun ORB, and C:ORBacus

In describing our experience of the Modelware paradigm, we make the following contributions in this paper:

1. We present Modelware, a model-driven approach, to separate middleware architectural concerns into multiple "viewpoints": an "intrinsic view" implementing common middleware functionalities through simple and coherent modules, and "aspect views" providing abstractions for crosscutting concerns.
2. We present the implementation details of the views in Modelware. More specifically, we describe the "realization" process for both the base view and aspect views and the "projection" process for mapping aspect view onto the base view.
3. We present a thorough evaluation of the Modelware paradigm to illustrate both the performance benefit and the high-level code reuse in supporting functional variations in both space and time.

The rest of the paper is organized as follows: we first introduce generic models including both the intrinsic models and aspect models of Modelware in Section 3; we then describe in Section 4 how transformations can be used to concretize generic models and to integrate aspect models to support flexible compositions of middleware functionalities; evaluations of Modelware are presented in Section 5.

2 Background and Related Work

Background and related work can be classified into two categories: aspect-oriented programming approaches and model-driven approaches. We will present these categories in turn and discuss similarities and difference to our approach.

Aspect-oriented Programming – *Aspects* modularize crosscutting concerns, coding concerns that are not localized within modular boundaries. Aspect-oriented programming (AOP) allows the developer to cleanly encapsulated crosscutting concerns in separate modules [13]. Aspect-oriented languages, such as AspectJ[5], defines a set of new language constructs to support two kinds of crosscutting: *dynamic crosscutting* and *static crosscutting*. Dynamic crosscutting is defined by means of *join points* that denote well-defined points in the execution of a program. A *pointcut* refers to a collection of join points and parameters associated with these join points. A method-like construct, referred to as an *advice*, is used to define aspect code executed *before*, *after* or *in place* of a join point. Static crosscutting affects the static structure of a program, such as classes, interfaces, and the type hierarchy. *Inter-type declarations* are used to *introduce* new fields and methods into classes or interfaces, as well as new entities into existing type hierarchies through the *declare parents* construct. An aspect module includes pointcuts, the associated advices, inter-type declarations, and declare parents constructs.

In the context of middleware, we refer to *aspect-oriented programming approaches* as existing software platforms that expose hooks for applications using these platforms to adapt, alter, modify, or extend the normal execution flow of a service requested. In that sense, the CORBA interceptor mechanisms, although not explicitly positioned as an aspect-oriented approach, belongs to this category. Other recent examples, explicitly positioning themselves as aspect-oriented approaches, are the JBoss AOP approach [3] and the Spring AOP approach [1]. The key difference to our work is that these approaches expose a number of hooks for enabling the use of the middleware in an aspect-oriented style. However, our main objective is to build aspect-oriented middleware through the use of aspect-oriented programming techniques, with the goal of increasing the modularity of the resulting middleware, to improve the concern separation in the middleware implementation, and to ultimately enable an automated model-driven approach.

AspectJ2EE [5] is a new aspect-oriented language, specifically targeted at the generalized implementation of J2EE application servers and applications. It is a programming language that could form the basis for an approach like ours.

Other approaches have used aspects for the development of middleware, for example, Facet [11] illustrates the use of aspects for the development of an event channel. We have shown how middleware implementations can be successfully refactored with aspects, increasing modularity and configurability [23, 22]. None of these approaches investigates reusability of aspects and effects of aspects on the evolution, as is our objective with Modelware.

Some work has been done on designing reusable aspects. Clarke and Walker[4] suggest the use of compositional patterns to better decouple the implementation of crosscutting concerns from the base classes of a system. Soares *et al.* [19] show how the use of *abstract aspects* effects the re-usability of aspects refactored from a health-care management system. Both approaches are very different from

[5] AspectJ, `http://www.eclipse.org/~aspectj`

the role-based approach of designing aspect-oriented libraries presented in this paper.

Model-driven Development – Generally speaking, model-driven development refers to a software development process that based on models of the software synthesized code. The Model Driven Architecture process (MDA) is one prominent examples of a model-driven development approach. MDA advocates developing complex systems through multiple and hierarchical viewpoints. The "Platform Independent Viewpoint" and the associated "Platform Independent Model" does not specify the details necessary for running the system on a particular platform, which makes it suitable for abstracting the essential functionalities of a system across a number of middleware platforms. By combining the specifications of the PIM with the details of how to use a particular type of platform, a "Platform Specific Model" is established. A set of mapping rules relate a PIM to its PSM that lays out the details with respect to a given middleware platform. How mappings can be effectively realized is still in question. The approach suggested in this paper is one possible realization for automating the mapping between different views and models.

Other approaches aiming at realizing a model-driven approach are [17, 2]. CoSMIC [17] defines a set of domain-specific tools for composing and deploying distributed real-time and embedded middleware-based applications. Bonnet *et al.* [2] describe a model-driven software process for the automated configuration and personalization of smart card software. Both approaches do not employ aspect-oriented techniques, which is central to our approach.

3 Generic Models in Modelware

The orthogonal natures among middleware functionalities allow Modelware to enable multiple viewpoints at the architectural level: a *base view* containing common middleware functionalities through a conventional layered hierarchy of modules, and a collection of *aspectual views*, each containing an "extrinsic" functionality. We raise the levels of abstractions in both kinds of views through models. Benefited from this design, the components of base view modules become much simpler and more coherent, thus, more tolerant to variations of application contexts. Leveraging traditional object-oriented design principles, both base and aspectual functionalities can be flexibly supported with different concrete implementations. The models in the base view carry many invariant properties which foster the creation of middleware-specific aspect libraries. Modelware can be thought as methodology for attacking the problem of commonality and variability [6] through the combination of conventional modules and aspects.

Before details of the models are discussed, we want to rephrase a few MDA nomenclatures in the context of Modelware. Our definition of the *Platform Independent Model* (PIM) refers to abstract concepts in both the base view and aspect views. We define the *Platform Specific Model* (PSM) as the refined models of these concepts for specific functional requirements or deployment platforms. For aspect views, in addition to PIM and PSM models, we introduce *role mod-*

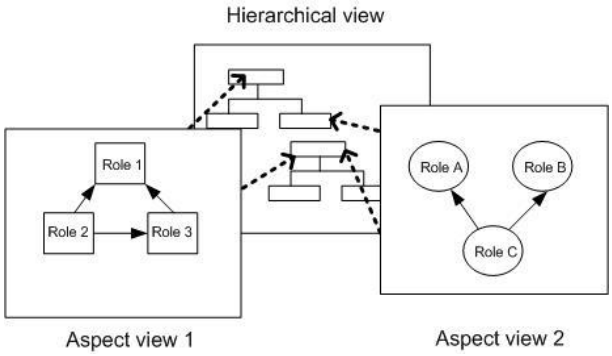

Fig. 2. Base view and aspectual views

els as abstractions for the behavior of aspects. As illustrated in Figure 2, each aspect view contains its own set of role models. An aspect view interacts with base view models via roles in a non-localized manner.

3.1 Invariant concepts in base view

It has long been recognized in literatures [7, 16, 21] that design patterns play essential roles in middleware architectures. In their specific problem contexts, design patterns exhibit invariance in both space and time. The Modelware base view is composed of a collection of "*invariant concepts*" including patterns as well as a number of design choices which we believe to represent common and essential functionalities of middleware.

3.1.1 Models of invariant concepts

The primary responsibility of the "invariant concepts" in the base view is to support the transparent interpretation and transportation of RPC operations. We enumerate a few essential elements and describe their semantics with respect to how they interpret the application requests made through RPC:

1. *Proxies (stub and skeleton)*: Stubs and skeletons are entities masking the middleware substrate as native programming facilities of the user application. Proxies see the application requests as regular method invocations.
2. *Connection facilities (acceptor and connector)*: Acceptors and connectors "*decouple the connection and initialization of peer services ... from the processing these peer services perform after they are connected and initialized*"[16]. Connection facilities see the application requests as a sequence of bytes sent to or received from network hosts.
3. *Protocols (initiator and responder)*: Protocol initiators and responders (also called forwarder-receiver [7]) leverage *connection facilities* and implement a particular sequence of message exchange between clients and servers. Protocols see the application requests as a set of generic messages subject to a specific temporal order and a specific spatial structure.

4. *Request sessions and service sessions*: A request session and a service session represents an instance of interaction among elements of *proxies* and *protocols* in the client and the server side, respectively. Sessions see application requests as instances of collaborations between *proxies* and *protocols*.
5. *Buffer*: Buffer is a commonly used data structure for encapsulating the application data. Buffer represents the application requests as a bounded array of bytes and provides interfaces to manipulate this array.
6. *Messages (outgoing and incoming)*: Messages, including both outgoing and incoming messages, represent the encoding and decoding of byte-oriented data in `Buffer` with respect to type-oriented data in user applications. Messages see application requests as typed and directional data traversing the middleware stack.
7. *Servant:* Servant is the internal representation in Modelware of the hosted servers. It serves as a level of indirection between `Protocols` and `Skeletons` to facilitate management tasks. It sees application requests as invocation requests to be dispatched to the destination services.

3.1.2 Simplicity and invariance

There are two important goals driving our design of the base view models: simplicity and invariance. In Modelware, models of these concepts are kept simple and minimal. On average, there are only around two operations associated with each entity, and most of these operations accept a single input parameter. This kind of simplicity is not arbitrary but derived from a small middleware core refactored out of its complex original version. In other words, this base view is intended to capture the smallest common denominator of middleware architectural variations. In fact, an implementation of this base view is capable of supporting CORBA-style RPC on platforms as small as Java Card, discussed in detail in Section 5.

More importantly, the base view concepts are stable designs surviving evolutions and variations in many middleware implementations. In addition to design patterns, some concepts are specified as standards, such as *request* (specified as streams in CORBA) and *servant* (specified as the object adaptor in CORBA). Others are widely adopted practices, such as *buffer* and *session*[6]. Being resilient to evolution is crucial to the base view in Modelware as it provides the foundation, i.e., architectural invariance, for establishing and integrating aspect views. As summarized by Grady Booch[7], we adhere to the "*simplicity via common abstractions and mechanisms*" principle to manage the complexity of change in middleware architectures.

---

[6] These design elements are present in all of the three major open source Java CORBA implementations, namely JacORB, ORBacus, and Sun ORB.

[7] Grady Booch. The Complexity of Programming Models. Keynote speech at AOSD 2005.`http://www.booch.com/architecture/blog/artifacts/Complexity.ppt`

3.2 Aspect views

Aspect models and views re-distribute the complexity of middleware implementation from a single flat module hierarchy to multiple separated and independent implementations of specialized middleware concerns. In Modelware, each view is oriented upon one or many *role*s specifying a specific interpretation of the Modelware base view. These interpretations are encapsulated within the aspect view in the form of additional program states (*role attributes*), interactions among roles within the view (*role relationships*), and interfaces for transferring control between aspectual views and the base view (*contracts*). Each aspect view interacts with the base view through "*projection*": a process of establishing an ontological relationship by mapping aspect roles to base view entities and fulfilling the aspect contracts on them. There are two types of contracts: abstract interception points (or `pointcut` in AOP terms) and abstract operations enforced by roles. Abstract operations link the behavior of a role to an base-view entity. Abstract pointcuts define points of execution and associated computation contexts of the base view for aspect views to intervene. Each aspect view is modularized as one or many reusable aspect components.

Different from generic roles in design patterns as well as conventional aspect oriented treatments of patterns [12, 9], we make heavy use of domain-specific roles that know about the base view abstractions such as *buffer* or *transport*. This dependency is necessary for making a large number of middleware functionalities reusable such as the synchronous communication model, the marshalling/unmarshalling of data types, and many others. We believe this dependency does not restrict the flexibility of the architecture for two reasons: 1. due to the strong invariance of the base view, the pointcut mapping is stable because the modular structures and the dynamic behaviours of the base view models are unlikely to change rapidly; 2. the dependency is made upon abstract models, therefore, stay unaffected by the platform specific implementations. To further illustrate aspect views, we present two concrete implementations: the thread-level concurrency library and the data type marshalling library. The projection process of aspect views is presented in Section 4.

3.2.1 Thread-level concurrency view

Description: Threads are common concurrency primitives popular in middleware implementations for achieving efficient request handling. From the perspective of the thread-level concurrency view (TC view for short), entities in the base view are of three kinds: non-concurrent, thread owners, and objects carrying the logic for the concurrent task. Currently, the TC view supports two well-studied middleware concurrency models, thread-per-connection and thread pool[8]. The *thread-per-connection* model detaches a new thread for a new network client. The *thread-pool* concurrency model initializes a fixed number of threads

[8] A third concurrency model, Reactive, as used in TAO [18], is also implemented as a separate view. Due to the length limit, we defer the discussion to an extended version of the paper

to execute tasks simultaneously. Threads in the *thread-pool* model are reused upon the completion of the task instead of being destroyed. The behaviour of threads is implemented in the library and automatically applied to the objects in the base view if these objects "*play*" the prescribed roles through specific projection transformations as illustrated in Figure 3. We discuss details of these transformations in Section 4.

Type: Domain independent. The TC view does not depend on any abstractions in the base view.

Roles and role relationships: The basic roles in the TC view are *Thread Owner* and *Thread Worker*. The *thread worker* contains the program logic to be executed concurrently, and the *thread owner* is an object in which the *thread worker* is created. Through projection, the *thread owner* role transforms the corresponding base view entities to different types of thread containers, and the *thread worker* role forces the corresponding base view object to conform to a uniform interface used by the internal threads of the library. Each role has two sub-roles to support the afore-mentioned two concurrency models.

Attributes: The common attributes of all thread owners are the base name of the thread, the thread group, and the synchronization primitive. This synchronization primitive is used if the execution thread of the owner needs to wait for the completion of the task in the thread. The *thread-per-connection* owner contains a repository of created threads. The *thread-pool* owner contains a repository of threads, a data buffer, and the size of the thread pool. No additional attributes are associated with the *thread worker* role. View-specific attributes are "mixed-in" with base view entities through AspectJ capabilities as shown in Section 4.

Role contracts: Each *thread owner* role is associated with a set of abstract operations and pointcuts. For instance, threads in the TC view are associated with states, much like Java threads. These states are often required to coordinate with the running state of the base view objects, e.g., observing the creation, the activation, or the disposal of the thread owners. The "`stateTranslate`" operation defined by the *thread owner* role forces base view objects which "play" this role to provide concrete mappings of base view states. Every *thread owner* is also associated with a set of abstract pointcuts, among which the most fundamental ones are to denote when threads need to be created and destroyed. In the case of the thread-pool model, an additional pointcut is used to define the point when the new data arrive, and a sleeping thread can be awoken to consume them.

3.2.2 Data type view

Description: Data marshalling/unmarshalling is an essential middleware functionality responsible for translating typed information in the middleware user application into an ordered array of bytes. The data type view is an aggregation of a number of primitive type views, each specializing in dealing with a single middleware data type.

Type: Domain-dependent. The data type view makes use of the *Buffer* abstraction in the base view.

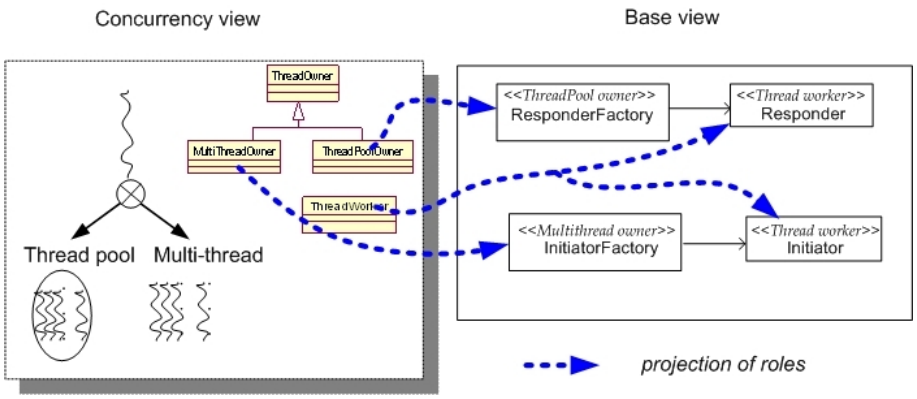

Fig. 3. Roles in concurrency view

Roles: The data type view consists of two roles, the *marshaller* role and the *unmarshaller* role. They represent entities responsible for encoding and decoding the user application data of the middleware.

Role relationships, attributes: No relationships are implemented between the *marshaller* role and the *unmarshaller* role as they represent two independent directions of data conversion. The data type roles do not have attributes because they are operation-oriented.

Role contracts: Both roles force the projected base view objects to implement an interface for retrieving the underlying data, i.e., a *Buffer* instance.

4 Transformation

The transformation process in Modelware consists of two independent operations: 1. *realization*, mapping base view models and aspect view models to concrete implementations; 2. *projection*, mapping aspect libraries to concrete implementations of the base view. The *realization* operation relies on an implementation library that stores *simple* and *coherent* implementation models. We discuss all the transformation within the Java language framework as it provides a mature environment for supporting both the base view and aspect views. The following sections describe the realization and the projection processes in detail.

4.1 Realization: PIM to PSM transformations

The PIM to PSM mapping is to establish mappings between abstract model elements and their concrete implementations through either sub-typing or direct substitution. Central to this process is the Modelware implementation library which aggregates two types of reusable components: functional implementations and public application programming interfaces (APIs). The implementation of models can be native, if it is part of the implementation library, or foreign, if

it already exists in third-party libraries. Proper adaptation of foreign implementations might be needed to conform to the operations of Modelware entities. For example, Modelware can leverage zero-copy buffers in the Java NIO libraries to achieve high performance I/O. The adaptation of the foreign component `ByteBuffer` to the `IBuffer` base view entity is simple leveraging the language facilities and the bytecode weaving capabilities of AspectJ.

Most of the native implementations come out of a crosscutting free version of ORBacus as a result of our long term refactoring efforts [23, 24]. A noteworthy characteristic of these implementations is that they are deliberately kept minimal by supporting simple behavior. For instance, the implementation of request handling assumes no response, and the transports are non-concurrent and incapable of handling fragmented messages. To reduce the coupling among concrete implementations, a number of patterns can be used including factories [8] and inversion of control (IOC) principles[9]. We currently use factories and are developing external dependency directives through either scripts or graphical tools.

The Modelware implementation library also contains modules defining public application programming interfaces. A particular set of public APIs represents a predefined "look and feel" for accessing middleware services. For instance, there are multiple public APIs for enabling the pluggability of network transports in CORBA such as the Extensible Transport Framework (ETF), defined by the OMG, and the Open Communications Interface (OCI), defined in ORBacus[10]. Conventionally, public APIs are typically hardwired to implementations at the development time by a type hierarchy. In Modelware, the base view models serve as a level of indirection between the implementations and the public APIs, so that public APIs can be plugged in and changed at post-compilation time. As illustrated in Figure 4, by separately managing the implementation and the interface, better flexibility and reusability can be achieved by creating the appropriate "look and feel" under external transformation directives.

4.2 Projection: Transformation of aspect views

The transformation of aspect models and views consists of both "realization" and "projection" operations. The purpose of the "realization" operation is to select concrete implementations for the aspect functionality. This is identical to the "realization" operation in the base view. The "projection" operation consists of two steps. We first determine the correspondence between entities in the base view and the roles in the aspect view. In the aspect library code, roles are represented by Java interfaces and instrumented with additional operations and states through AspectJ. Leveraging AspectJ's capability of type hierarchy modification, this mapping operation is straightforward and affects every concrete

[9] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern `http://www.martinfowler.com/articles/injection.html`

[10] ORBacus OCI `http://www.orbacus.com/support/new_site/manual/4.2.1/users_guide/index.html`

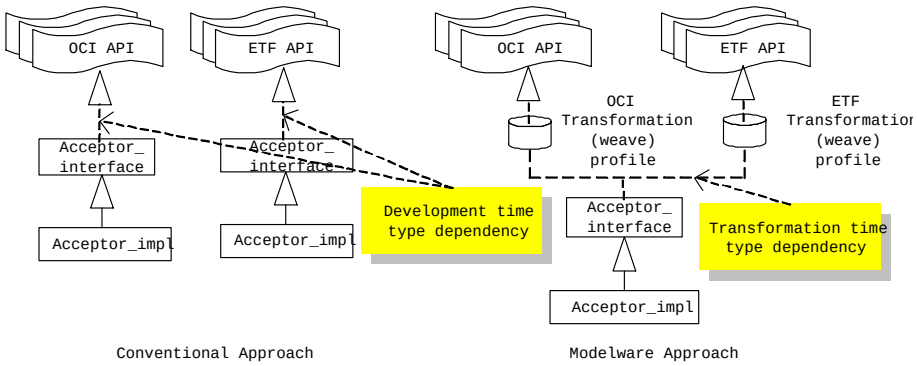

Fig. 4. Transform-time API dependency in Modelware

implementation of the mapped base view entity. Once the mapping is established, we need to fulfill the contracts declared by the aspect view. This is a process of locating concrete interception points and providing implementations of new operations for the base view classes, as the result of "role playing". Since contracts are composed of abstract programming elements, the enforcement can be accomplished by the AspectJ compiler.

To further illustrate the mapping process, we present a usage scenario of the concurrency aspect view. Figure 5 shows the Modelware implementation of the server-side request handling. While the focus is not the exact semantics of these statements, we want to illustrate a typical "simplistic" Modelware implementation – it is only about the operational logic of request processing. Many common concerns, such as iterative processing, thread safety, and concurrency, are absent. Instead of hardwiring into code as in conventional ways, we illustrate how we enable the "thread-per-connection" concurrency support with a minimal coding effort using the Modelware threading aspect library.

Figure 5(B) is a code snippet showing only the core operations of the thread library. Line A defines a contract using an `abstract pointcut` to capture the constructor invocation of the `ThreadWorker` made by the `ThreadOwner`. Lines B2-B5 create a thread before the constructor call, assign the newly constructed `ThreadWorker` to the thread, start the thread, and return the created `ThreadWorker` instance. The "thread-per-connection" concurrency model requires the server-side request handling, i.e., the operation `process` (line 5 in Figure 5(A)), to execute in a separate thread. Therefore, the base-view class `ProtocolResponder` plays the `ThreadWorker` role, and the base-view class `ProtocolResponderFactory` (line 4) the `ThreadOwner` role. Figure 5(C) shows the projection code: line 1-2 modify the type hierarchy of the base view entities; line 3 fulfills the *abstract pointcut* contract by specifying the constructor call of all subtypes of `ProtocolResponder`; line 4 cancels the invocation to the to-be-made-concurrent method "`process`" in the main thread, and line 5-6 fulfills another contract by specifying the method "`process`" is to be executed concurrently. The actual

```
1 public void ready() {
2    ITransport transport = acceptor_.accept();
3    IProtocolResponder responder =
4          ProtocolResponderFactory.instance().getResponder(reg_,transport);
5    responder.process();
6 }
```

A. Server Request Processing

```
public abstract aspect Threading {
A.       public abstract pointcut createThreadOnInstance();
B.       Object around(ThreadOwner owner):createThreadOnInstance()
1                    &&this(owner)&&!within(Threading){
2                WorkerThread t = this.makeThread(..);
3                ThreadWorker worker = (ThreadWorker) proceed(owner);
4                t.setWorker(worker);
5                t.start();
6                return worker;
         }
```

B. Generic Threading Library

```
1 declare parents: ProtocolResponderFactory implements ThreadOwner;
2 declare parents: ProtocolResponder implements ThreadWorker;
3 public pointcut createThreadOnInstance():call(ProtocolResponder+.new(..));
4 public pointcut etherizeActiveMethodCall():call(* ProtocolResponder+.process(..))&&!(within(Threading+));
5 public void ProtocolResponder.doWork(){
6                process();
         }
```

C. Projection code

Fig. 5. Transformation of the thread-level concurrency view

functionality of our thread library[11] is more complex including thread lifecycle management, state transition support, synchronization support, and others. Our experience, also as shown in this simplified example, is that, once the roles are mapped, the code needs to be created is simple and small in size. In addition, since the projection code itself is an aspect module, many different projections can be implemented to support additional concurrency models without intrusive changes to the base view entities. In addition, in scenarios where middleware threading is not required or cannot be used, the plain implmentations can still be used.

5 Evaluation

Our assessment of Modelware examines both the performance characteristic and the programming effort for the use of Modelware models and libraries in building common middleware operations. For this purpose, we choose to support CORBA interfaces as a case study, although Modelware is not designed specifically for CORBA. For the performance evaluation, we compare the Modelware

[11] Please visit Modelware website for details of the implementations. `http://www.msrg.utoronto.ca/code/Modelware`

CORBA (MORB) implementation with ORBacus using Benchie [20], an third-party CORBA benchmark suite. We also quantify the programming effort in three case studies: 1. creating CORBA-like middleware; 2. supporting functionality evolution of middleware in time; 3. supporting functional diversity in space, i.e., different computing platforms from Java Card, to mobile devices, and to desktop environment.

5.1 Modelware functionalities

The key base view elements of Modelware are implemented largely by generically reusing ORBacus components such as buffer, acceptor, connector, transport, and GIOP encoding/decoding algorithms. The following properties are implemented in aspect libraries: data types such as `long` and `char`, two way communication model, thread-level concurrency, thread safety, codeset support, Java NIO support(including reactive request handling), and many others. These properties are largely orthogonal to each other and can be flexibly combined. The base view elements, without any aspect libraries, are capable of handling remote invocations with octet and integer data types. The reliability of messaging passing is guaranteed at the network level, and the receiving side processes requests passively.

5.2 Runtime characteristics

In this set of performance evaluations, we primarily want to demonstrate the benefit of the architectural flexibility of Modelware in competing with ORBacus on the same set of benchmark measurements collected by Benchie. The performance delta should not be influenced much by algorithmic factors but mainly architectural ones since almost all of the critical Modelware functions, such as data marshalling and unmarhalling, GIOP protocol stack, and connection management, are just reused ORBacus implementations. The benchmark tests are performed on Pentium 4 2GHZ PC running Linux Redhat 8.0. We disable the concurrency protection of user applications for both MORB and ORBacus[12].

We present three categories of benchmark tests: a. roundtrip pings representing the minimum cost of CORBA stack traversals; b. data marshalling/-unmarshalling operations representing the performance of client-encoding and server-decoding capabilities; c. multi-server tests representing the dispatching capabilities of CORBA. We customize[13] MORB for these three categories as follows: since the concurrency support is not necessary for tests in categories `a` and `b`, "threading" and "thread-safe locks" become redundant and are configured out of the architecture. We denote this configuration as "MORB_A". We enable the "concurrency" support and disable all other features such as "interceptor" and "context" for category c Benchie tests in the "MORB_B" configuration.

[12] This is the default policy of ORBacus

[13] A reminder that our customization only involves changing the selections of compiled classes for bytecode weaving.

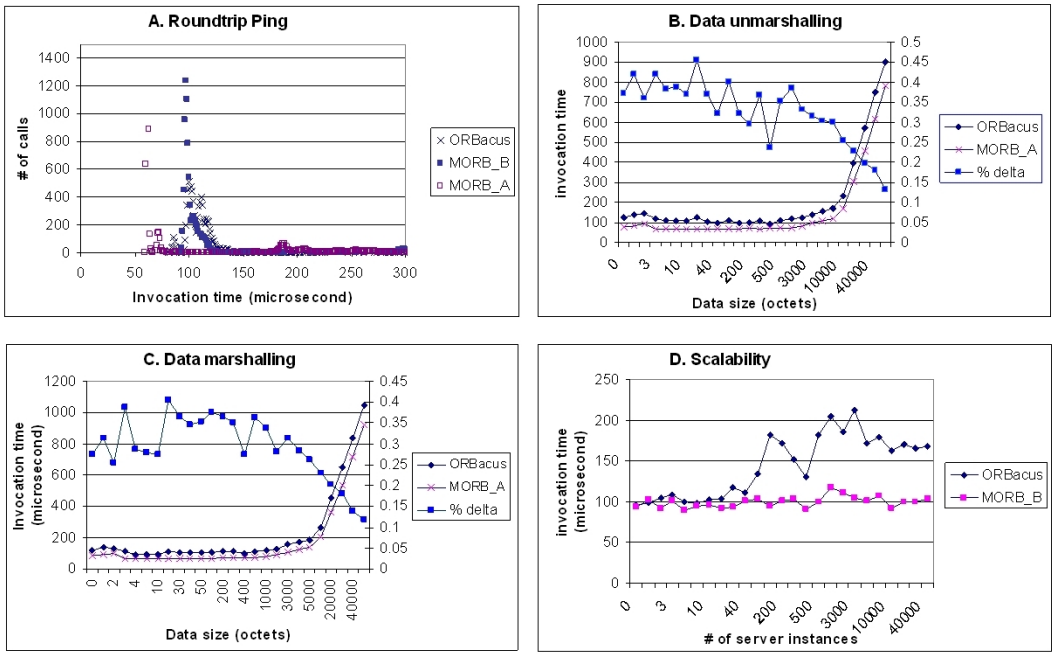

Fig. 6. Benchmark comparison of Modelware to ORBacus

We have created 12 configurations of MORB for the complete tests. Due to the length limit, we present the more detailed and complete benchmarking results in an extended version of this paper. We show the results of benchmark tests for both MORB configurations and ORBacus in Figure 6. Figure 6:A shows that, for 10,000 pings, MORB shows dramatic performance improvements over ORBacus, as the shape of the histogram of "MORB_A" shifts to the left of that of ORBacus. The average invocation time for MORB is 105 microseconds, a 43% speed-up comparing to 183 microseconds found with ORBacus. We believe this is primarily due to the Modelware's ability of lifting concurrency overheads since, once we enable "concurrency" and "thread-safe" features in "MORB_B", the average invocation time increases to 161 microseconds. In the marshalling and unmarshalling performance comparisons (Figure 6 B and C), the improvement decreases from 40% to 12%, as the descending differential curves on both graphs show. This confirms the fact that MORB reuses the encoding/decoding algorithms of ORBacus, and the performance difference tends to diminish, as the data exchange work dominates the request processing. Figure 6 D shows that, in the absence of facilities such as "interceptors" and "context", the dispatching can be more efficient in MORB compared to ORBacus, an architectural flexibility enabled by Modelware to optimize for performance.

5.3 Transformation for evolution in time: Platform evolution

In many performance-sensitive application domains, high performance is often a mandatory requirement in addition to the location transparency. This translates to low overhead and fast response for request processing in the middleware layer. TAO [18] is a successful example of high-performance implementations exploiting techniques such as zero-copy buffer, reactive communication models, and the high speed network I/O. For a lot of conventional middleware implementations, many such techniques are not employed because of the limitations of the underlying OS and VM at the time of the design. The evolution of OSs or VMs might lift these design limitations in the infrastructure but not easily in the middleware architecture. This is because leveraging new capabilities often requires systematic, i.e., crosscutting, changes to many middleware architectural layers such as the data representation and the network communication design. The new I/O introduced in Java 1.4 platforms[14] is an example of VM evolution having profound impacts on Java-based middleware architectures. Its zero-copy buffer and asynchronous I/O primitives can be used to dramatically improve the performance of traditional stream-oriented middleware message passing. In Modelware, this improvement is captured entirely in a separate aspect library and can be transparently applied to the base view at post-compilation time.

The core entities of the `Async` aspect library consist of a reactor and four roles: `AsyncAccpetor`, `AsyncConnector`, `AsyncTransport`, and `AsyncWorker`. The primary function of the library is to disable the blocking operations in conventional Java network I/O, initialize and install "channels" onto appropriate roles, and register these roles with the `Reactor`. The `Reactor` dispatches incoming data to corresponding `AsyncWorker`s based on their registration keys. There are two different approaches of projecting this library to the base view, one being mapping these four roles to the base view model entities. The "async" functionality thus affects all concrete implementations of `Acceptor`, `Connector`, and `Transport`. However, in foreseeing future non-socket based connection management in Modelware, we chose to project onto the concrete implementations instead[15]. The library is 30KB in zipped byte-code size. The projection code only involves base view models and their implementations. Therefore, no new code is created for MORB to become reactive except the mapping of an abstract pointcut. This mapping starts the `Reactor` when MORB is initialized by standard CORBA APIs.

To quantify the performance improvement, we simulate a multi-connection scenario as follows: we host MORB on a IBM ThinkPad T41 running WindowsXP, and we start a number of clients on a Pentium 4 2G box running the Linux 2.4 kernel. The two computers are on a wireless LAN. Each client uses 300 "oneway" calls to warm up, and the time is taken for the completion of the next 300 calls. All the clients are separate processes synchronized by a semaphore to try to create as many simultaneous connections on the server side as possible.

[14] Java NIO. `http://java.sun.com/j2se/1.4.2/docs/guide/nio/`

[15] As we mentioned earlier, projection is done through "declare parents" statements and very easy to modify.

Table 1 summarizes the average time for each scenario comparing the reactive MORB with the proactive version (unit is in milliseconds). Our results confirms the findings [15] that request processing based on asynchronous I/O greatly alleviates the middleware overhead of threading when the number of simultaneous incoming connections is large (over 50 in our case). In Modelware, these two communication facilities can be inter-changed at the bytecode level.

Number of clients	10	50	100	500
Ave. Proactive	43.3	476.98	250.49	620.59
Ave. Reactive	73.1	195.02	135.53	208.67
Improvements percentage	0%	60%	46%	66%

Table 1. Improvements of using Java New I/O in Modelware

5.4 Support evolution in space: Platform diversity

Application domains of middleware systems have diverged from traditional enterprise environments to mobile and embedded devices due to the popularity of ubiquitous computing. Differences of computing environments manifest in the middleware architecture as different APIs, communication styles, data types, and many others, even though the RPC semantic does not change. In conventional architectures, evolving middleware into different platforms or domains often results in non-modular modifications to the architecture so that the code reusability of common functionalities is dramatically reduced.

The focus of this experiment is to measure how well Modelware supports reusability in creating middleware platforms for three dramatically different application domains: smart cards (Java Card), mobile devices (J2ME), and traditional environments (J2SE). We measure reusability as the ratio of the code size (LOC) between reused components in the implementation library and the entire middleware implementation. We distinguish between two types of usability: inter-domain reusability, where components are reused in all three platforms, and intra-domain reusability, where components are selected for a specific platform. We have implemented three Modelware-based CORBA implementations: the Java Card platform (872 LOC and 56.1k bytecode size), the J2ME platform (1894 LOC and 219k bytecode size for the full configuration), and MORB (3346 LOC and 283k bytecode size for the full configuration). The J2ME version is created and tested using the Nokia Series 60 emulator[16]. The Java Card version is created and tested on the Sun Java Card toolkit 2.2.1. The Java Card implementation is significantly smaller than the J2SE and J2ME versions because Java Card applications always play a passive role in the master-slave model[17].

[16] Nokia Series 60 Platform. `http://forum.nokia.com`

[17] Java Card: http://www.javaworld.com/javaworld/jw-03-1998/jw-03-javadev.html

Therefore, we only implement the request processing functionality for the Java Card instance of Modelware.

In Table 2, we report our measurements of both inter-platform and intra-platform reusability for these three implementations. For each implementation, we also list the features being reused or created. Our experimental implementations show that different flavors of ORBs can be created with a high degree of reusability. The code to be newly created to support new platforms ranges from 2% to 9% of the entire ORB code size.

Platform: Standard desktop platform (J2SE)
Overall reusability: 91.36% (cross-domain 16.56%, intra-domain 74.8%) *Cross-domain reuse:* Buffer, GIOP Protocol, messages, stub, request, response, servant. *Intra-domain reuse:* Object reference, concurrency control, transport, type support, two-way communication, protocol initiator and responder, OMG interfaces *Newly created:* ORB interface impl, OMG interface adaptation for Modelware components
Platform: Mobile devices (J2ME)
Overall reusability: 97.5% (cross-domain 24.15%, intra-domain 73.28%) *Cross-domain reuse:* Buffer, GIOP Protocol, messages, stub, request, response, servant. *Intra-domain reuse:* Object reference, concurrency control, type support except float & double, two-way communication, transport, protocol initiator, protocol responder, OMG interfaces *Newly created:* J2me version of the ORB interface implementation, OMG interface adaptations as mentioned previously.
Platform: Embedded devices (Java Card)
Overall reusability: 97% (cross-domain 63.53%, intra-domain 34.27%) *Cross-domain reuse:* Buffer, GIOP Protocol, messages, stub, request, response, servant. *Intra-domain reuse:* Transport, protocol responder, Modelware hashtable, Modelware vector *Newly created:* Java card ORB interface implementation

Table 2. Reusability study of Modelware in supporting different application platforms

6 Conclusion

We believe one of the main reasons for insufficient component reuse in system software such as middleware is the presence of crosscutting concerns. We have observed two major characteristics of this deficiency. Firstly, many middleware abstractions, such as design patterns and usage idioms, live persistently across evolution stages, but their implementations do not exist as development artifacts

that can be directly reasoned and reused. Second, many designs and algorithms are repeatedly applied in conventional architectures. Unfortunately, due to their crosscutting nature, no effective ways exist in explicitly representing, evolving, and reusing them.

Our solution to overcome these difficulties is through Modelware in applying the model-driven approach to the middleware architecture itself. The foundation of our approach is to enable "multiviews" in the middleware architecture. That is, we explicitly represent the intrinsic properties or the internal logic of the middleware through platform independent models in the "base view" of the middleware architecture. The implementations of these abstract concepts, i.e., the Platform Specific Model, are stored in the implementation library. The transformation between PIM and PSM models are in form of dependency descriptions. In addition to the base view, we model and encapsulate crosscutting properties of the middleware architecture in individual aspect views. Aspect views dilute the density of the per-module design complexity by exploiting the orthogonalities among middleware design concerns. In our case studies, we are able to add new computing capabilities to Modelware through reusable aspect libraries. We have also illustrated that supporting functional diversification in space with Modelware only requires relatively small coding efforts.

We are currently continuing in evaluating Modelware approaches in the following directions: we are working fervently in supporting the complete set of CORBA functionalities through Modelware in order to conduct a more thorough comparison; we are working on facilitating the configuration process through tool support and automated reasoning. At the same time, we are also interested in how Modelware supports other flavors of middleware systems besides those based on RPC. Modelware will be serving as an important platform for experimenting with the properties of aspect oriented middleware – our long term research focus.

References

1. Chapter 5. Spring AOP: Aspect oriented programming with spring. In *www.springframework.org*, Accessed 05/2005.
2. Stephane Bonnet and Olivier Potonnie. A model-driven approach for smart card configuration. In *GPCE*, Vancouver, October 24-28 2004.
3. Bill Burke and Adrian Brock. Aspect-oriented programming and JBoss. In *ON Java.com*, 05/28/2003.
4. Siobhn Clarke and Robert J. Walker. Composition patterns: An approach to designing reusable aspects. In *ICSE*, pages 5–14, Toronto, Canada, May 2001.
5. Tal Cohen and Joseph Gil. AspectJ2EE = AOP + J2EE. In *ECOOP*, pages 219–243, 2004.
6. James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Softw.*, 15(6):37–45, 1998.
7. Frank Buschmann et al. *A System of Patterns*. John Wiley & Sons, 1997.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

9. Jan Hannemann and Gregor Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 161–173. ACM Press, 2002.
10. William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428. ACM Press, 1993.
11. Frank Hunleth and Ron Cytron. Footprint and Feature Management using Aspect-Oriented Programming Techniques. In *Languages, Compilers, and Tools for Embedded Systems (LCTES'02)*, 2002.
12. Elizabeth A. Kendall. Role model designs and implementations with aspect-oriented programming. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 353–369. ACM Press, 1999.
13. Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
14. Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Trans. Softw. Eng.*, 20(10):760–773, 1994.
15. D. C. Schmidt. ACE: An Object-Oriented Framework for Developing Distributed Applications. In *the 6th USENIX C++ Technical Conference*, Cambridge, MA, April 1994. USENIX Association.
16. Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture Patterns for Concurrent and Networked Objects*, volume 2 of *Software Design Patterns*. John Wiley & Sons, Ltd, 1 edition, 1999.
17. Douglas C. Schmidt, Aniruddha Gokhale, Balachandran Natarajan Sandeep Neema, and *et al.* CoSMIC: An MDA generative tool for distributed real-time and embedded component middleware and applications. In *OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Seattle, WA, November 2002.
18. Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the tao real-time object request broker. *Computer Communications*, 21(4), April 1998.
19. Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *OOPSLA*, pages 174–190, 2002.
20. Petr Tuma and Adam Buble. Open CORBA Bench Marking. *SPECTS 2001*. URL: `http://nenya.ms.mff.cuni.cz/~bench`.
21. Uwe Zdun, Michael Kircher, and Markus Volter. Remoting patterns. In *IEEE Internet Computing*, number 6, pages 60–68, November/December 2004.
22. Charles Zhang, Dapeng Gao, and Hans-Arno Jacobsen. Towards Just-in-time Middleware Platforms. In *4th International Conference on Aspect Oriented Systems and Design*, Chicago, IL, March 2005.
23. Charles Zhang and Hans-Arno Jacobsen. Refactoring Middleware with Aspects. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1058–1073, November 2003.
24. Charles Zhang and Hans-Arno Jacobsen. Resolving Feature Convolution in Middleware Systems. In *Proceedings of the 19th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, September 2004.