Thoughts on OMA-NG:

The Next Generation Object Management Architecture

Craig Thompson
Object Services and Consulting, Inc.

Ted Linden and Bob Filman
Microelectronics and Computer Technology Corporation (MCC)

OMG document ormsc/97-09-01

Abstract: The OMG Object Management Architecture (OMA) has proven to be a successful foundation for achieving distributed, interoperable systems. However, the OMA faces challenges, both internally and from competitors, to raise the level of support it provides. In the near future, it will no longer be good enough for an architecture to merely enable the desired behavior (program semantics). Instead, there are a host of other system properties, or "ilities" (e.g., reliability, interoperability, security, quality of service, manageability, extensibility, and scalability) that users will rightly come to expect the OMA to support.

The OMG Object Management Architecture Reference Model (OMA-RM) and the containing *OMA Guide*, once was (and still ought to be) the central architectural handbook of OMG. The Object and Reference Model Subcommittee (ORMSC) is presently considering updates to these documents. This architectural "green paper" proposes some areas for discussion that we hope will ultimately lead to improving the OMA-RM and OMA Guide. Our focus is on extending the ORB and services model of OMA to an integrated set of services that enable and enforce ilities.

  - code, agent management architecture, RM-ODP, patterns
  - **other extensions** - semantics, object model extensions, domain interface architecture, needed interfaces
- **Recommendations**

Problem, Objective, Impact

OMG is currently considering the problem of how to update its "Object Management Architecture Reference Model" (OMA-RM). The OMA-RM is a chapter in its *OMA Guide*, the central architectural handbook of OMG. The Object and Reference Model Subcommittee (ORMSC), reporting to the OMG Architecture Board, has been formed with that revision as part of its charter. This document is a position paper for consideration by ORMSC.

The OMA-RM has served OMG well since 1990. It is both an explanation of the current OMG architecture and a guide to the evolution of that architecture. However, seven years is a long time in the computer field. We assert that it's now time to discuss and revise the OMA-RM to reflect current reality and to facilitate future development. We hope this green paper will provide some useful insights on architectural directions, particularly with respect to componentware, composing components, building scalable systems, and the interactions of components in achieving software ilities. We discuss several specific candidate upgrades to the current OMA-RM and *OMA Guide* and provide a list of areas where more work is needed to better understand and extend the OMA.

The Object Management Architecture Today

OMA-RM and OMG architecture

The "Object Management Architecture Reference Model" was published in 1990 as part of the *OMG Object Management Architecture Guide*. In addition to the OMA-RM, the *OMA Guide* contains additional supporting material, including chapters entitled "Introduction" and "Overview" (describing OMG goals and benefits), "Technical Objectives", and "The Object Model," appendices "Glossary" and "Policies and Procedures." Revisions of OMA-RM in 1992 and 1995 made small additions, and a more substantial revision was approved in August, 1996, but is not yet part of the currently published *OMA Guide*. For the purposes of this paper, we will use the term *OMA-RM* to refer specifically to the reference model and *OMA* generally to refer to the entire body of OMG architecture specifications. These include the architectural principles, Common Object Request Broker Architecture, Object Services Architecture, Common Facilities Architecture, Domain Interfaces, and all adopted specifications (even specifications in process).

What the OMA-RM itself explains.

By design, the OMA-RM is a high level architecture. Its function has been "to map out areas to be addressed rather than to impose design constraints." Its modus operandi has been to identify major components without itself defining detailed interface specifications. The OMG technology adoption process takes care of such specifications. OMA-RM has evolved in several steps:

- OMA RM 1.0 - The original 1990 OMA-RM described how objects make requests and receive responses and abstractly described the Object Request Broker, Object Services, Common Facilities, and Application Objects. It included the distinction between object interfaces and implementations that permitted non-object-oriented software to be wrapped in objects that participate in the OMA.
- OMA RM 2.0 - The 1992 revision added the Interface Description Language and CORBA.
- OMA RM 3.0 - The 1995 revision (*the current official version*, ab/97-05-05) removed some examples of object services and common facilities, replacing them with explicit reference to adopted CORBAservices and CORBAfacilities. The glossary has been slightly reduced from the 1992 version. The object model chapter has been revised into the "Core Object Model."
- A 1996 revision of OMA RM by Geoff Lewis at the Madrid OMG meeting was approved by the OMG Architecture Board (I think). It removed examples of object services and common facilities, replacing them with explicit reference to adopted CORBAservices and CORBAfacilities; and added a paragraph on Domain Interfaces and three pages on Object Frameworks. The latter describes a hierarchy of application, domain, common facility, and object service frameworks that are compositions (e.g., by interface inheritance or client requests) of application, domain, common facility, and object service objects. This included the conceptual notion that higher level objects may call lower level objects (e.g., common facilities call object services). It says that the specification of an Object Framework "defines such things as the structure, interfaces, types, operation sequencing, and qualities of service of the objects that make up the framework." The 1996 OMA-RM was just seven pages long.
- Discussion of the Object Management Architecture (*http://www.omg.org/library/oma1.htm*) Another version of the OMA-RM and some of the chapters in the OMA Guide was developed for the CORBA Academy and appears in online. It includes the 1996 RM and includes descriptions of adopted object services.

What the OMA-RM does not explain

The OMA-RM does not explain nor does it preclude a long list of desirable architectural extensions. Some of these are presaged in the original OMA Guide "Overview" and "Technical Objectives" sections but are not yet realized in the OMA. They are candidates for addition to these sections. In the next section, we examine several such extensions and discuss the virtue of extending the OMA to cover them.

Possible Extensions to the OMA

We've organized our set of architectural extensions to the OMA into the following subsections, working our way from the most general principles of software architecture to the specifics of interoperating in the real world:

- **architectural principles and properties (the -ilities)** - architectural principles, properties (ilities) and frameworks
- **composability and componentware** - containment model, binding time, packages, interceptors, extensions, guards, economic model

- **scalability via federation**
- **performance guarantees and system management** - real-time, quality of service, system management
- **convergence with the competition** - componentware, Java, web, DBMS, KBMS, mobile code, agent management architecture, RM-ODP, patterns
- **other extensions** - semantics, object model extensions, domain interface architecture, needed interfaces

---

Architectural Principles, Properties, and Frameworks

Architectural Principles

The OMA-RM does not explicitly mention the "architectural principles" that were first described in the *Object Services Architecture* and now appear in the RFP template. It should refer to these principles explicitly. The architectural principles should probably become a chapter or section in the *OMA Guide*.

Architectural Properties

The OMA-RM does not currently say much about desirable architectural properties (beyond interoperability).

Software is designed primarily to meet *functional* specifications---that is, to achieve particular input and output behavior. However, a well-designed system achieves an appropriate balance of other properties: those that encompass its ability to exist in its environment and evolve as that environment changes. Following evolving practice, we call these *ilities* [Table 1]. Ilities are architectural properties that often can cut across different levels or components of a system's architecture to be properties of the whole. Unlike simple functionality, ilities can include tradeoffs---for example, enhanced interoperability may have been produced at the cost of greater security. Although ilities are sometimes invisible properties of the system as a whole, they represent real (though perhaps unspoken or even unrealized) customer requirements. The enterprise architect or system architect is responsible for ensuring such ilities, and a most appropriate task for an architectural framework is to support them. Currently, the OMA does address *security* and is beginning to address *quality of service*, *real-time*, and *composability*. It's time for the OMG architecture to address other ilities, such as interoperability through time and space---that is, the ability to evolve to meet new demands. We discuss specific ilities in greater detail below.

  - interoperability
  - composability
  - scalability
  - evolvability
  - extensibility
  - tailorability
  - security
  - reliability
  - adaptability
  - survivability
  - affordability

- maintainability
- understandability
- performance
- quality of service
- real-time
- nomadicity.

Table 1. Some Ilities

Frameworks

There has long been a question as to what the technical distinction is between *object services* and *common facilities*, even between *platform* and *domain* (e.g., is workflow Platform or Domain Technical Committee?). Historically, such technical distinctions were mostly ignored and have mainly been treated as convenient intra-organizational divisions of labor.

But the 1996 OMA-RM section on frameworks is three pages long [In contrast, the domain interfaces section is a paragraph just four lines long.], describes an explicit hierarchy of application objects, domain objects, common facilities, and object services, and gives no concrete examples of object frameworks so they remain abstract. Frameworks appear to be constellations of objects that operate together. But most object services and common facilities define more than one object class. So it is not really clear how common facilities and common facility frameworks would differ. For instance, are OODBs, workflow systems, KBMS systems, repository management systems, and BOFs frameworks or are common facilities? The section on frameworks needs to be re-thought since at present frameworks are really a research area at OMG.

Composability and Componentware

From the beginning, high level goals of OMG have included reusability, interoperability, extensibility, and dynamic evolution. OMG customers have wanted the ability to assemble applications quickly from standard parts, and OMA framework users were seeking consistent semantics for how shared services work when using the full suite of OMG specifications. We have made good progress on separable specifications but have said too little about composition of services and facilities. Current specifications do not provide enough information to realize the desirable the goals of being able to *mix-and-match* and *plug-and-play* with alternate implementations of services or facilities.

Composing services with application objects

Current OMG service specifications assume that the application developer understands all the relevant services and invokes them when appropriate. This means that calls to specific service functions (including implementation specific options) become deeply interwoven with the application logic. The interweaving of separate concerns is especially serious in the cases where the service requires insertion of functionality at the calling side of an interaction, as occurs with services such as encryption, authentication, and replication.

To make services appear as componentware within an application, we need better and more structured ways of adding services while minimizing their impact on the application code. Often, service functions can be implicit in application method invocations. Frameworks can protect the application code from being intertwined with calls to service functions. However, it is not clear how to add (chain?) or control implicitly specified services. Binding times for adding services are not specified nor is there any mechanism to control which users can add services dynamically.

Service Interactions

Before the ORB bus architecture was adopted in 1989-90, some people argued that both versions and DBMS functionality were needed. While it was clear that services needed to interact, no one was satisfied with any particular block diagram or calling tree showing particular services explicitly calling other services. The CORBA bus architecture, with its uniform set of services hanging from the bus seemed to be a good solution. It allowed separately specified services and deferred the issues of service composition. OMG has followed this course for the last eight years. But we still have little experience connecting the services on the bus together to do useful work. [According to the IEEE, an architecture is a collection of functions and interconnections. Interconnections reveal the relationship between the functional components. Ironically, by mapping calling trees to a set of functions communicating over a single message dispatch bus, OMG has provided only the definition of functions and not the information about interconnections.]

This legacy is visible in OMG discussions about specifying CORBAservices. Such discussions are marked by a tendency to state that one service might depend on another without explicitly committing to such dependencies. Another manifestation is seen in the use of IDL. An object's IDL encapsulates its implementation with no mention of service dependencies. (One must examine the implementation to discover them.) On the positive side, this provides implementation independence for clients of interfaces. On the negative side, it prevents service substitutability and leads to application bloat and large, monolithic ORB's. A side effect is that a high level component like a BOF or an application component may have no way to specify how it uses Object Services (if it does). OMG users expect higher level OMG objects like facilities to be composed partly from lower level services and they want the ability to spin out stable internal components for independent reuse.

***Uses* specification or Containment Model needed**

Some sort of *uses* specification is needed to declaratively relate implementations to interfaces on which they depend. Alternatively, some sort of explicit *containment graph* is needed to relate a higher level component to components it contains or depends on. Hopefully, the *CORBA Component RFP* will address this.

Binding Times

The OMA-RM does not address how dynamic the binding is for adding new services to existing frameworks (though the old Technical Objectives chapter of the *OMA Guide* did mention Extensible and dynamic changes like adding new implementations, replacing implementations where the interface has not changed, replacing implementations where the interface is changed, and changing the location of implementations).

Dynamic Dispatch, Interceptors and Extensions

Good news about the OMG bus architecture is that it promises flexibility. If object A calls object B using the bus architecture, then the call may be transparently local or remote and furthermore the call can be *intercepted* or *filtered* by any number of implicit operations which can be executed on the client or the server or somewhere in between to insure that concurrency control is respected if B is a shared object, that B is versioned or not, that A has authorization to access B, that B is coerced to a form that A can accept, that the replicated copies B' and B" receive copies of the request, and so forth. There is a long list of these implicit actions, or *extensions*. OMA needs a generic mechanism for specifying these implicit service actions without expecting the application programmer to understand and manage all this implicit functionality.

Many Kinds of Extensions, Need for Guards

The interceptors above are just some of a large family of implicit (or explicit) side-effect *extension* behaviors that set up *boundaries* that need *guards* to enforce that the extensional behavior is not bypassed. Others are:

- crossing intellectual property boundaries - license objects before use
- micropayment boundary - install payment and collection
- replication and consistency management - for availability and fault tolerance
- annotations - so comments and expertise from others can be added to a source object like a document
- derivation histories - so copied objects retain the source of the copy
- removing duplicate postings
- adding or filtering out advertisements (removing spam via keywords FREE, CASH, RICH, SALE, etc.)

More Packaging Technology Needed

Principled selection and decomposition into subcomponents can enable substitution and customization by developers other than the original designers. But there are reasons to hide containment information as well:

- **Who can install Interceptors?** If generic interceptors are used in applications, it must be that not just anyone anytime can install new interceptors. An obvious case is security. Here interceptors trap accesses to allow modular security checks. If anyone can add or modify a service's interceptors, then there can be no guarantees that the objects are guarded by interceptors. This implies that we need inviolate ways for application developers to package interceptors, controlling which can be added or removed and what information is available to each. And it must be understandable at a high enough level that it is easy to do and does not cause developer errors. In a way, instead of putting data objects *into* a database, one can think of the data objects as *guarded* by a set of extensional behaviors like persistence, security, replication, versioning, distribution. What was formerly a monolithic DBMS might become a collection of guards that form a sort of DBMS *exoskeleton*. The flexibility is in being able to add new services modularly to the list to create the kind of data store you want. A puzzle for OMG is how to represent the list of behaviors (before and after methods, a schedule, context variable, and so forth) to insure proper control.
- **Roles.** Another architectural concept related to packaging not mentioned in the OMA Guide is *roles*. Different classes of developers might want to install different interceptors:

- the security group may want to insure that a certain security policy is followed
    - the system management group may want to insure uniform system monitoring and may want a load balancing ability to move objects across distribution boundaries without affecting application semantics
    - the configuration board may want to insure versioning
    - a system administrator may want to insure that conversions occur to unpack compressed files

  In some cases, a developer wants to install a certain family of interceptors and then seal off the ability of anyone else to augment (that part of) the interceptor list. (In fact, this corresponds to how we build stovepipe systems today as particular compositions of particular capabilities with little access to how to add other capabilities or customize or extend the ones that are in the system.)

  If not everyone can replace at will any part of an OMG architecture at any time, then the abstract notion of roles is needed to complement the packaging idea. Roles might be global to an OMG system (e.g., CORBA vendor, developer, end-user) or local to an abstraction of the system (e.g., the system management role, the security administration role, the configuration management role, the DBA). At present in the OMA there is no structural way to associate roles with responsibilities over parts of the architecture. Roles appear to be related to packaging.
- **Policies**. If there are multiple implementations of services and variations on how the service can work then there also need to be *policies* that control the services' behavior.

Some related issues:

- **Footprint.** At present, applications or services are encapsulated and opaque. A user of several OMG applications whose implementations share common services may discover multiple copies of the same service in the execution environment. A containment model at least reduces this redundancy. The Java community has developed some approaches that help reduce this problem.
- **Debugging and run-time fault resolution.** One of the key problems with componentware is that system developers and run-time administrators find it very difficult to resolve faults that originate within "black-box" components. Finger pointing is a problem when there are N <=2 suppliers in a debug environment. To make componentware a reality, a set of services for pattern-directed tracing, logging, and monitoring will be useful.
- **Complexity.** Another problem with componentware is the same as with toolkits - too many interfaces are exposed. OMG provides a small forest of separable specifications. This has the benefits of conceptual isolation and potential reuse. However, it also has the drawback of complexity. Each application composes the services into the application in unique and idiosyncratic ways. Programmers are used to the complexity of subroutine and class libraries, and used to cursing the complexity of these libraries as they grow large. Distributed systems add the additional complication of requiring ilities to be implemented consistently, system-wide, throughout the application. It might make most sense if there are standard configurations of services and facilities (the analog of DBMS, KBMS, workflow, etc.) and interceptors dispatch to these (maybe that is what a common facility or a framework is, a named configuration?). The NIIIP Consortium uses such a mechanism, which it terms a virtual enterprise object - every ORB request is shunted through a VE gateway which governs a collection of common decisions made when the VE is constructed.
- **Tailorability.** How easy is it for a developer to tailor a system, framework, facility, or service developed by others to meet the developer's needs. What are the supported mechanisms? (e.g., subclassing, installing new interceptors, negotiation to be able to support a range of platform sizes) Given frameworks A and B with function f only available in B, how can users of framework A get

this extra function f if they need it?

- **Adaptability and Survivability.** Can large systems developed with componentware gracefully degrade and still survive to provide a useful subset of their original mission?
- **Multiple dispatch overloading mechanisms: are they all needed?** Interceptors, POA, and Trader all provide a mechanism for implicitly calling some side-effect behavior. Trader appears to be separately useful but are both POA and interceptors needed?
- **Higher level Dynamic Dispatch Mechanisms than interceptors and POA.** Rules, filters, and before-and-after methods might provide higher level means of specifying implicit actions rather than using a low level mechanism like interceptors.
- **Portability.** Even assuming technical solutions for component isolation, the question remains how will OMG state *environmental constraints* on components. Components implemented by C++ may not port easily to other environments. Most legacy code encapsulated as components will be pinned to the environment it is compiled into. Only Java or other languages compiled with a virtual machine will have the portability requirements for components across a variety of platforms.
- **Java Decompilation.** Java bytecodes contain names of classes, methods, and fields and reverse compilers can reconstruct class definitions. Companies worry how to preserve their investments in designs. Better compiler technology can protect against decompilers by making reconstruction harder for third parties by mangling names or rearranging statements. But the main point is, this is a case where the developer role has self-interest in exposing only some interfaces.
- **Visibility and Profiles.** The notions of *visibility* and *profiles* mentioned in the old *OMA Guide* Introduction seem related to packaging:
    - **Visible and Invisible objects.** *Visible objects* are identified and manipulated at service interfaces. *Invisible objects* are those whose behavior may affect behavior at the interface but are not directly identified or manipulated. It is not clear how to control visibility of objects in the OMA.
    - **Profiles.** Also mentioned in the old *OMA Guide* are *profiles*, which are collections of OMG technology (object model capabilities, services and facilities) that can be extended for use by some technology community (e.g., OODBs) or domain community (e.g., CAD). This notion may need to be revisited since the extensions that are community idiosyncratic directly lead to interoperability problems at the borders of a profile, e.g., as when ODMG-profile data might need to be distributed via CORBA. Also, limits on expectations need to be understood better -- can STEP be viewed as an OMG profile just because there is a mapping from STEP to IDL? Closer to home, if UML introduces 80+ semantically useful constructs that users begin to use widely to represent their problems, does that constitute a profile just because there is a mapping to IDL? Users of the results of the mapping might need to be aware of the UML semantics as well as the details of the mapping to maintain their resulting OMG code.
    - **Economic Model.** All of the considerations above are technical enablers to make it possible within the OMA to separately specify and control components. But it must be the case that an economic model for componentry exists as well. Even if it is possible to separately isolate components for reuse, there must be a marketplace for making them available. We view the emerging markets in ActiveX and JavaBeans components as an encouraging sign of this possibility.

Scalability and the Federation Pattern

Scalability is a desirable property of componentware: we want to build larger systems from components, unrestricted by the particular size or internal organization of the components.

One notion of scalability refers to the ability of a design to maintain a linear relationship between demands on a system and required resources. This is illustrated in the "Technical Objectives" chapter of the *OMA Guide*, where scalability is defined as "the effect on request processing times as the numbers of objects accessible to a client increases and the number of requests handled increases."

Build-time componentware definitions might cast scalability as the ability to build larger systems or systems of systems from component parts. An example would be composition as described above in which a common facility might be defined from object services, for example, an OODB as some composition of naming, persistence, query, transactions, security, etc.

An important variant to composition of parts to form a whole is the special case when the parts are similar. We can define *federation* to be the composition of like systems that operate together to form a higher level system. When there was just a single LAN-based CORBA, OMG could choose to not spend to much emphasis on federation interfaces. But with CORBA interoperability and Visigenics fanned out the the Netscape desktop, the number of machines connected to an ORB might go from 40 to 40 million. No longer should we assume that there is a single name service, a single transaction service, a single query service, a single trader, ... in such an environment. Minimally, we'd expect many instances of these services. And we'd expect them to interoperate with others of their kind, so that name services federate, transactions federate, and so on. Not only services but also facilities should federate so workflow, OODBs, simulations, etc. would intercommunicate with others of their own kind.

To achieve scalability via federation, OMG must address several architectural issues:

- In many cases, the result of a federation is recursively similar in interface to the base level services (e.g., a query service that has the same interface to child query services). But the result of a federation might be a different abstraction, so that a collection of softball players make up a team and team behave differently than individual players.
- Decentralizing control - While an architecture cannot guarantee a scalable system, certain architectural decisions can preclude it. Most obviously, positing a single, centralized locus of some activity imposes a ceiling (the point at which the demands on that locus degrade its performance) on the scalability of an architecture.
- Defining which interfaces of the components are interfaces of the federation as a whole and enforcing the interfaces required to participate in a federation. For instance, DBMS systems might federate by obeying the XA transaction protocol or they might federate via some mediating schema or both. The High Level Architecture is a federation architecture for simulations that abstracts out system time and inter-simulation events to allow new simulations to enter an HLA simulation by agreeing on the protocols that define these two area.
- Instances of a single service will vary by manufacturer, version, and features. We need these services to interoperate with others of their kind. For that to happen, services need *policy* interfaces to specify and later to negotiate and resolve semantic conflicts between confederates, e.g., no versioning and branching versioning, discretionary access control and mandatory access control. Given two services or frameworks that are alike in all ways except for some policy p that governs some service s, what can we say about how they interoperate modulo that policy?

Taken together, with both composition and federation as ways of building complex systems from simpler ones, one can come back to the question of ilities and talk about *end-to-end X* or *top-to-bottom X* where X = security, QoS/bandwidth adaptive, real-time guarantees, optimization, maintainability, reliability, affordability, licenseability for reasonable cost, testability, understandability, seamlessness, evolvability, survivability, etc. When a system, for instance, a communications network is a federation of similar subsystems, then we can talk about *end-to-end* guarantees. When a systems is composed of parts, we can talk about *top-to-bottom* guarantees.

Performance Guarantees and System Management

It is hard to keep your promises if those promises depend on things you can't control. So an application that makes some form of performance or quality of service guarantee needs its underlying infrastructure to expose enough information or controls to enable the keeping of those promises. More globally, if an enterprise system is to know what kind of system-wide performance is being achieved, the system must expose enough information to allow that monitoring.

Real-time and QoS

Engineering is the process of trading among system attributes. Performance, precision, accuracy, and reliability pervade many components of a system. *Real-time* and *Quality of Service* both involve capturing traffic, accuracy, and other meta data from trend data and current data in a system and providing guarantees on certain outcomes, e.g., that a certain class of user might receive some negotiated level of service. Much of the QoS literature focuses on the issues of getting the underlying components (principally the network) to provide QoS guarantees to applications. Higher-level applications are said to be *bandwidth-adaptive* or *QoS-aware* if they can take special advantage of guarantees made by lower levels of the system to perform better and make further guarantees to their own customer applications.

So far CORBA, CORBAservices, or CORBAfacilities are QoS- and bandwidth-insensitive. This reflects the appropriate emphasis on the semantics of actions, rather than their pragmatic attributes. However, pragmatics also matters; the architecture needs interfaces for collecting system performance data and for exposing QoS constraints.

OMG is already actively focusing on real-time and QoS via the active Real-time SIG and QoS Working Groups. A reason to highlight these -ilities in this paper is just to remind us that some kinds of information that must be transmitted between components may be passed in ways other than functional IDL interfaces of the kinds so far specified in OMG. An implication for portability of service implementations is that the first generation of services and facilities will not be QoS-aware because there are not yet standard ways to transfer this kind of information. Another implication is that it is likely that OMG compliant first-generation QoS-unaware implementations should remain useful for many QoS-unaware uses (the majority of uses today) even when QoS guarantees are specifiable (that is, QoS-unaware will be a degenerate, special case of QoS-awareness in a similar way to version-unaware is a degenerate case of version-aware where the policy is to keep only the current version and similarly for many other services.)

System Management

We note that one reason many large installations are slow to adopt CORBA-based technology is the lack of system management facilities. System management involves performance measurement, accounting, intrusion detection and security, failure analysis and debugging, and configuration management. These elements have enough in common and are sufficiently fundamental to suggest their treatment in the OMG architecture at more than the "service" level. System management services are critical enablers for componentware because components are black boxes; all debugging must focus on the interaction between components.

Compressibility

Compressibility could have a significant impact on scalability, reliability, affordability, understandability, performance, and QoS in several different ways that affect bandwidth. Compressibility can be achieved by following architectural specifications on the representation and characterization of the data - the idea is that the message attribute could just state that it could be compressed/decompressed by an operator X which would result in a compression ratio Y but it would need CPU resources Z at the sending and V at the receiving end. This could be done independently from the data and application type, other than pointing to X. Then, the decision and the possible compression invocation would be transparently done in the pipe. This raises the issue of whether to overlay the OMA with an attribute driven metadata architecture, which would be useful for other services beyond compression.

Convergence with the Competition

Several current and emerging representational paradigms compete with OMG. Rather than having user communities scatter to a Babel of architectures, OMG must either provide a migration path or subsume the concepts of these alternatives. OMG may have had the first distributed architecture, but being first is not a guarantee of survival. This suggests extending the *OMA Guide*, (either in the chapter on OMA-NG or in the chapter on "Technical Objectives") to provide a road map for convergence with the following:

A Market for Components

Componentware is the industry buzzword for small isolatable chunks of software that can be assembled into large applications. We discussed above some of the concepts we believe are appropriate to an OMG notion of composition and federation of components. The notion of an economy of componentware is that components are vended separately by potentially many vendors. OMG has not yet really demonstrated itself to be a componentware community though we have taken great care to adopt an *isolation principle* for orthogonal object services. While there are many ORB vendors, there are relatively few object service providers and those mainly vend a collection of components from one vendor. This is, one’s ORB vendor is often the source of one’s CORBA services. There is not enough evidence of portable object services or higher level applications that are compositions of collections of services. Componentware does not *per se* require a certain variety of object technology. We note that the ActiveX/COM and more recently the JavaBeans environments have active componentware sub-industries. OMG has just issued Component Model and Component Scripting RFPs to try to jump-start OMG into contributing to the componentware movement.

Microsoft ActiveX and COM

OMG has already provided guidance in this area. Of course, this is a 600 pound gorilla force to be reckoned with with an active componentware marketplace.

Java

With Java on servers as well as on clients, Java and OMG are increasingly addressing the same market. It is an open question as to whether the Java and OMG specifications will converge. Java and IDL are at the same level of representational adequacy, but Java definitions come attached to a programming language. IDL is language neutral and IDL mappings have been defined for several programming languages. Java benefits from the greater semantic specificity an underlying language provides, but also has an emerging need to directly supply standard *Java to X* mappings for legacy language applications. Some of the distance between OMG and Java is narrowing: JavaSoft is moving toward OMG by adopting into its Enterprise Java both CORBA naming and transactions and making accommodations for OMG IIOP. OMG is likely to adopt a JavaBean-like component model, Java-based scripting (pure speculation of course!), and harmonization with Objects by Value, Multiple Interfaces, and Portable Object Adapter. Already, Java libraries are extensive, implemented and conveniently accessible. Momentum is high. OMG has a reasonable collection of specifications for middleware services so far unmatched by Java or ActiveX. And OMG has defined an IDL to Java mapping. But what happens when the Java class library community implements all the OMG services? Will the specifications be compliant to OMG services and the defined mappings? Or will variants spring up? One example is collection class libraries. OMG collections are not the same as ODMG collections which are not the same as C++ STL collections. Java collection classes seem more likely to be based on C++ STL. Will there be a time when OMG permits specifications described natively in Java which use Java-to-IDL mappings to insure backwards compatibility to IDL? Will that become the norm? Should we just go ahead and use the IDL to Java mapping to convert all OMG specs to Java now to make convergence easier? Or should we permit OMG specifications written in native Java (when we have IDL to Java and Java to IDL mappings)? Java may soon have its own standards body, similar but not the same as OMG. Java has the potential for producing real working libraries more quickly than the OMG process. If that happens, OMG might find itself less a visionary leader into new directions and more a redundant afterthought.

Web

Information once available only in local file systems is now available via the World Wide Web. The web architecture consists of lightweight clients providing a desktop, servers providing document content, and a means of accessing backend applications that provide semantics often accessible via CGI scripts. The addition of Java to the client and server side enhances the architecture. All this has been accomplished without the complexities and simplifications of formal specifications (i.e., IDL.). The recent addition by Netscape of Visigenic's ORB provides one reasonable way to glue the web and ORB architectures together. Interestingly, it is possible to imagine adding services to a URL Request Broker without ever introducing IDL, CORBA, or OMA. For instance, one can add an annotation service on a client to access a URL and any URL annotation repositories and return annotated documents. This is very like the OMG interceptor architecture but with proxy based intermediary services interposed between client and server. There are many similar services (e.g., add advertisements, strip advertisements, micropayment, microlicense, PICS, etc.). The puzzle for OMG is how to converge the web and CORBA architectures so there will not become two parallel sets of services (e.g., web traders and CORBA traders, web versions and CORBA versions). A scenario for convergence might be if there is a web document guarded by the OMG security service, subject to the annotation service, that can use OMG versioning (if we had it). A fitting name for the addition of Internet/Web motivated services might be *Internet Services Architecture*, an extension to the OMA.

Database Management Systems

While the original "Introduction" to the *OMA Guide* listed Object Database Interfaces in scope for OMG and OMG has defined services for persistence, transactions, queries, naming, and collections, there is no OMG compositional facility for an OODB or Object-relational DBMS, nor are there gateway interfaces for SQL3, ODMG, or ODBC/JDBC interface defined by OMG except via a string API call to the Query Service (so for instance, there is no defined mapping from IDL to relations or vice versa). Since a lot of the world's data is in relational DBMS systems (and a little is in OODBs) there is a need for OMG interfaces to these kinds of DBMS systems. Furthermore, there is an opportunity for OMG to define federation interfaces that access multiple data sources in the network uniformly, as is done in various experimental communities (e.g., OMG Internet SIG's OTAM, DARPA I*3 and Data Server, MCC Infosleuth).

Knowledgebase Management Systems

Loosely, if a KBMS is a composite of objects, rules, constraints, before-and-after methods, etc., then one would like to find a path to grow OMG's object modeling capability to add these extra representational capabilities.

Mobile Code

For most of its lifetime, OMG has provided only a partial solution to distribution, namely object reference at a distance. Only recently has it added Call by Value for moving object state and Mobile Agents for adding mobile code. Java-based schemes are rapidly evolving and OMG's schemes are following Java.

Agent Management Architecture

OMG has issued and is considering responses to the Mobile Agents RFP. This direction could add mobile state, mobile code, and mobile objects to the OMG OMA. An architectural puzzle for OMG is whether agents subsume objects the way that objects subsume abstract data types. [Object systems might someday come to be viewed as degenerate agent systems, ones where the agents are passive and immobile. When that day comes, we'll need to change the OMG and OMA to the AMG and AMA.] Even if agents are just an "optional" extension to the OMA and not a core topic, the diverse family of technologies, ideas and directions of agent technology could lead to a collection of RFPs to cover autonomous action, cooperation and negotiation, acting on behalf of others, proactive and reactive control structures, intelligence in reacting to its environments and planning, scheduling, and/or learning.

RM-ODP

The Open Distributed Processing community has developed a number of concepts that complement OMG's architecture:

- *Distribution Transparencies* - these map differences in representation, invocation mechanisms, failure and recovery, persistence, transactions and consistency, location, migration and relocation, and replication. (There seem to be many more transparencies than these.) These could be added to the "Technical Objectives" in the *OMA Guide* since all seem to be OMG objectives.
- *Viewpoints*
  - the *enterprise viewpoint* contains the notion of a *community* (a configuration of objects formed to meet an objective) and an *X-federation* (a community of X-domains), *roles* played by the system (defining permissions, obligations, prohibitions and behaviors), *policy statements*, a system and its environment
  - the *information viewpoint* contains schemas, invariant predicates, and specification of allowable state changes
  - the *computational viewpoint, engineering viewpoint, and technology viewpoints* - these contain notions like cluster, interceptor, binder, checkpoint, recovery, channel, consistency rules and conformance statements among many others.
  - ODP Functions - categories of ODP functions include system management, coordination functions, repository functions, security functions, non-repudiation functions
- The OMG and RM-ODP architectures overlap. In some ways, ODP may provide the broader conceptual framework and OMG the more enunciated framework architecture. There seems to be some real opportunity to borrow some of the ODP framework into OMG (and possibly for OMG to standardized some of its services and facilities through ODP). But some substantial conceptual mapping would have to be done to do this wholesale since object services, common facilities, and domain interfaces plus OMG architectural principles do not seem to map exactly into ODP concepts.

Patterns

There is now substantial and useful literature on OO patterns, which are abstract specifications of the behavior of constellations of objects. Pattern collections can be mined for new OMG services. The patterns methodology can be incorporated into the OMG technology base. Puzzles involve rationalizing vocabularies for patterns and frameworks.

Other Extensions

The following extensions are already being discussed in OMG and we are not adding much to this discussion except to list them as significant candidate extensions to the OMA-RM.

Views of the Architecture

We need some additional views of the OMA architecture, not just the software bus view. Other views might explain how services are interconnected, how ilities are inserted, how visibility and packaging is supported, dataflow, physical distribution, etc. This is in addition or complementary to possibly adopting ODP viewpoints.

Semantics

There is already activity in OMG to address the need to say more about specifications than just their interface and an English explanation of their intended semantics.

Object Model Extensions

This paper doesn't currently address object model extensions needed except to mention some ideas:

- **Templates** - there is no IDL template or generic facility and no macro facility.
- **CDL or JDL** - the JBOF CDL extended IDL with ODMG ODL which includes ODMG relationships (which differ from OMG Relationships Service), extensions for collections of instances of a class and keys, and some other additions. Some modelers have found these extensions useful.
- **Rules** - Like relationships, rules could either be thought of as an extension to an object model or as a service. There are several kinds of related rules systems (ECA Rules, before-and-after rules). In the last year, the NIIIP consortium made a valiant but so far unsuccessful effort toward getting OMG interested in rules.

Domain Interface Architecture

The OMG-RM provides short descriptions of Object Services, Common Facilities, and Domain Interfaces. Separate substantial *Object Services Architecture* and *Common Facilities Architecture* documents currently exist. Historically, these have proved reasonably valuable to the Platform technical committee in setting the scope and road maps for the Object Services Task Force and Common Facilities Task Force. (On the other hand, there never was a predecessor *ORB Architecture* document apart from the growing CORBA specification.) The Domain Technical Committee has chartered a broad collection of task forces and SIGs but there is currently no *Domain Interfaces Architecture* (DIA) to guide OMG in how to populate the DTC space. This has led to recent (fairly emotional) discussions on the need for such a document. Problems have arisen between DTC groups on boundaries between, say, common business objects and manufacturing objects partly because there are no criteria for setting boundaries except one group "getting there first." On the other hand, having architecture documents has not guaranteed

- that they would be updated to reflect adopted specifications,
- that they would continue to guide road map discussions, or
- that some other group with a seemingly different charter might not generate RFPs apropos to another group (e.g., the Topology RFP generated by DTC Telecom appears to be a PTC specification),

Despite these problems, such architecture documents do substantially help OMG move forward by providing "stakes in the ground." The need for a DIA document could be a discussion point in ORMSC (or DTC) that could lead to a new work item. If so, someone would have to come forward with at least the sketch of a strawman document. [It appears that the Common Business Object Working Group of the Business Object Domain Task Force is preparing an RFI to develop a DIA.] A DIA document might mention the areas that OMG currently is committed to:

- business objects, electronic commerce, finance, insurance, manufacturing, transportation, telecom, healthcare, GIS and imagery, and gastronomy

and other potential areas like

- logistics, command and control, and crisis management

and provide guidance on domain scope. The potential space is very large. Other domain standards populate parts of that space (e.g., STEP).

Needed Services and Common Facilities

- **Change Management Service.** The *Object Services Architecture* has a service descriptor called Change Management that described version management, configuration management, and transformation management. The *OMA Guide* "Technical Objectives" chapter described a versioning requirement. OMG mailing lists have discussed whether to issue a Change Management Service RFP. Some of the discussion has revolved around what the subject of the service should be: instances, interfaces, or implementations or all three. The OMA does need to provide a uniform framework for managing change since all data and all long-lived systems evolve. In this area, OMG lags practice and will need to reflect emerging practice for incrementally downloading upgrades across the web (e.g., Marimba Castanet, NetCast-like models).
- **Replication Service.** A Replication Service has been a topic of conversation in OMG ORB and later ORB/OS meetings. But it has never made it to RFP status. One promising approach is based on *group objects* -- an object that acts as the surrogate for a collection of replicas so that operations on the surrogate cascade to the replicas via some control regime (e.g., master-slave, peer-peer, voting schemes, ...) and corresponding consistency maintenance policy, which may promise less than instantaneous consistency.
- **Rules Service.** This was mentioned above. Event-condition-action rules or before-after rules provided via some useful language construct would probably provide a higher, level easier to use capability than low level interceptors.
- **Planning Service(s).** Enterprise level command and control systems depend on these. So do intelligent agents. This might include the ability to represent possible futures.
- **Scheduling Service(s).** In many environments (e.g., factories) scheduling is used to optimize throughput or share scarce resources. Agents that plan may schedule the execution of their subtasks. Planning and scheduling probably warrant a Platform SIG of their own.
- **Uncertainty, Precision.** There is a need to be able to explicitly represent uncertainty. Also, for some applications (geographic), there is a tradeoff between precision and compression.
- **Other Services and Facilities.** Caching Service, Indexing Service, Parsing Service, Event Logging Service, Search Engine Facility, Information Access Facility/Object Transfer Manipulation Facility, ODBMS, KBMS,

---

Conclusion and Recommendations

OMG still offers the most comprehensive collection of specifications for distributed middleware and is an active community with strong industry backing. It must continue to evolve rapidly to survive since it does not stand alone -- there are rival technologies like ODP (more conceptual), agents (richer abstraction base), Java (broadly based implementations in rapid fire evolution), and DCOM (strong market penetration). OMG must develop convergence stories for each of these to best represent the needs of its communities.

The OMA-RM has historically been and continues to be useful "as is." The most conservative position is to change it as little as possible since it has served as a permissive guide to allow many architectural extensions to be layered on top as the entire body of OMA is populated. The original OMA-RM reflected a vision of the future. Now some of that future is in place. Its role may need to grow to explain that architecture. And its vision of the future may need to be revised to reflect the better perspective of six years of experience. As a starting point for those discussions, we recommend:

- Continuing the current practice of replacing references to abstract parts of the OMA architecture (e.g., object services in the 1990 OMA-RM) with concrete references to named adopted CORBAservices in the 1996 OMG-RM). The concrete portions serve as a high level reminder of the current scope of the OMA. We need to determine how to splice in abstract elements of the architecture in order to serve as a scoping guide to OMG directions while still distinguishing what has been done from what is planned.
- Only a few abstract extensions should be adopted into the body of the OMA-RM document itself. These should identify some key architectural directions for the entire OMA. Some additions to the OMA-RM we believe are needed are in the next revision are:
  - Mention the *architectural principles.* Add a short section on architectural properties (*ilities)* and mention scalability, evolvability, survivability, real-time, QoS, and systems management.
  - Add some supporting examples of object frameworks to show how they are different from (especially) common facilities (or drop the concept).
- · Changes that need to be made to the *OMA Guide*
  - Add a new chapter to the OMA Guide containing the architectural principles and properties.
  - Add glossary terms to the "Glossary" chapter. Candidates for additional terms include: domain interface, encapsulation, skeleton, stub, framework, component, containment model, external environment, packaging, identify, granularity, interceptor, extension, policy, governs or owns, boundary, seamless or implicit, exoskeleton, architectural principles, -ility, isolation, property, scalability, survivability, real-time, quality of service (QoS), naming, transaction, trader, and concurrency. Also see Componentware Glossary.
  - Update the "Technical Objectives" chapter to modularly include:
    - technical objectives on architectural principals and ilities (component model and packaging, scalability by composition and federation, evolution, real-time and QoS.)
    - technical objectives for ORB, services and facilities like trader, language mappings, scripting language
    - technical objective(s) on domain interfaces
    - interoperability with industry standards including the Web, Java and COM/ActiveX
  - Differentiate requirements that have been met from requirements that are so far unmet (like versioning).
- Keep the *OMA Guide* up to date (annually) and use it as a primary OMG architecture guide.

This research is sponsored by the Defense Advanced Research Projects Agency and managed by the U.S. Army Research Laboratory under contract

DAAL01-95-C-0112. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of the Defense Advanced Research Projects Agency, U.S. Army Research Laboratory, or the United States Government.

This research is sponsored by the Object Infrastructure Project of the Microelectronics and Computer Technology Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of the Microelectronics and Computer Technology Corporation or its member companies.

Acknowledgments: This paper is based on lessons learned primarily in the OBJS project *Scaling OSAs to the Internet* and the MCC project *Object Infrastructure Project.* The authors gratefully acknowledge inputs received from Art Goldschmidt on lessons learned in the National Industrial Information Infrastructure Consortium (NIIIP) and from Gabor Seymour (Motorola).