7

DESCRIPTION, CONVERSION, AND PLANNING FOR SEMANTIC INTEROPERABILITY

A. Rosenthal and E. Sciore
The MITRE Corporation , 202 Burlington Road, Bedford MA 01730 , (both), and Boston College, Chestnut Hill, MA , 02167 (Sciore)
arnie@mitre.org, (617)-271-7577; 271-2523 (fax).
sciore@cs.bc.edu (617) 552-3928, (617)-552-2097 (fax)

Abstract

The issue of *semantic interoperability* pervades distributed systems whose components are designed independently. For example, when a source intends 70 to mean "70 meters, with 20% error", the receiver must not interpret the result as "70 feet, exactly". We examine semantic interoperability problems in a distributed object management environment, i.e., for a system of distributed objects and services rather than for a database. A *semantic interoperability service* is defined, and its required functional components - argument describers, conversion functions, and a planner - are identified. We categorize levels of service that each component may provide, and how these choices affect system functionality and evolution. We also begin to examine the various ways to map the functional architectures to software components – sources, receivers, mediators, and request broker – and data administration tasks. Finally, we contrast with semantic interoperability for database clients, and discuss why the type system should not be the repository for conversion knowledge.

Keywords

Semantic interoperability, distributed object management, CORBA
Practitioner/Researcher: borderline (systems engineering, consulting)

1 INTRODUCTION

As networked, interconnected computers become the norm, it becomes possible to envision distributed systems in which applications "plug and play" in patterns that were not anticipated by their designers. This flexibility is not easy to achieve, however, due to the difficulty of finding

R. Meersman et al. (eds.), *Database Applications Semantics*

and invoking services, in passing arguments, and in interpreting the received arguments. *Distributed Object Managers* (DOMs) have been proposed as a way to resolve the first two difficulties. This paper is concerned with the last problem, namely, interpretation of the arguments.

1.1 Distributed Object Managers and Semantic Interoperability

A distributed object manager acts as an intermediary between a client application and the computing resources it accesses. It provides the client with a location-independent, object-based view of the available resources. By location-independent, we mean that the client does not need to know the location of a server; instead, the client is able to send and retrieve data as if the request were to be processed by a local resource. By object-based, we mean that a client interacts with resources by sending messages to perceived objects, independent of the server's operating system, invocation mechanism, or parameter representation. Remote request protocols exist in the database world for requesting services and data (e.g., RDA, DRDA [Cash94]), but are based on relations rather than object classes and are not location transparent. Services based on such standards for remote requests and data will be referred to as *middleware*, due to their position between clients and the actual databases and services.

For concreteness, we describe semantic interoperability in the context of a particular DOM standard, the Common Object Request Broker Architecture (CORBA) [OMG94]. CORBA provides a standard interface, called the Object Request Broker (ORB), through which clients can request operations on objects. Values and object references can be passed as arguments and received as results.

A DOM guarantees that if a system sends a simple value (say, the integer 70), the receiving system will also see a 70, even though the actual representations of the two values may be completely different. There is no guarantee, however, that the exchange is *meaningful*. For example, the receiver may expect that the value it receives means 70 km/hr with high confidence, when in reality the sender meant 70 m/sec with low confidence. In general, both the sender and receiver of a data value may have many implicit assumptions and expectations about the meaning of the value. If the respective assumptions conflict, the transfer of the data value is at best useless and at worst dangerous. Consequently, it is important that the semantics of the data being transferred be preserved; we call this property *semantic interoperability*.

Semantic interoperability is needed whenever the components of a distributed application have different expectations about data they exchange. These differing expectations create significant difficulties for application development and maintenance, and will become more prominent as users try to exchange and integrate data from increasingly large communities. The following four paragraphs illustrate the role of semantic interoperability.

First, consider an application that intends to display various images that it receives. For each source, it sends a message requesting an image file and some numbers indicating the size of the image. It expects the image to be in GIF format and the size numbers to be in centimeters. However, certain sources may provide PostScript or JPEG images, and send the size information in inches or pixels. Without semantic interoperability, such images are unusable. With semantic interoperability, the images would be converted somehow into the form expected by the application. Also, if the receiver specifies requirements that cannot be met (e.g., higher precision than is available, or high confidence in authenticity), it should be warned of the inadequacy so as not to produce incorrect results.

Second, consider a routing application that accesses several databases to receive information about a military mission. Various distances may be received, such as Target_Distance, Cruising_Range, etc. These numbers may have different formats and interpretations, e.g., great circle or route distance, nautical miles or kilometers. The differences in units can be resolved by automatically-called conversion routines; however, if direct and en-route distances are not convertible then users need to be alerted to the semantic mismatch, thereby preventing serious errors.

Third, error messages need to be understandable to applications, but there is no universal standard for heterogeneous environments; also it is necessary to permit server-specific return codes. Semantic interoperability requires that each service's error code would be mapped into the application-understandable code that best matches it, with warnings for poor matches.

Fourth, consider an application which interoperates with multiple information retrieval services. Some services may require that a string have a fixed number of characters, left padded with blanks; others may have variable length strings, or pad on the right with zeros. Escape characters and wild cards may differ, and so on. Semantic interoperability should be able to resolve these differences automatically.

There is a common theme among the above examples. With current technology, the clients and servers need to be aware of too many details of each other's interface, so the information required grows as the number of possibly-communicating pairs. A more abstract interface would be more desirable. Ideally, the interface would require an application to describe the *meaning* it intends (e.g., Target_Distance) and the assumptions it makes about data values being transmitted. That is, the abstract interface between systems would be in terms of values that described their own representation and assumptions. Furthermore, the descriptions might often be in a knowledge base, rather than being passed explicitly.

Most current research on semantic interoperability has been performed in the context of heterogeneous database systems [ACM90]. An exploration in the DOM context may help us understand the issues by separating fundamentals from artifacts of the database usage. In addition, we hope that awareness of the other applications will lead to designs of semantic interoperability services that are more generally useful.

1.2 The Scope of This Paper

We define a *Semantic Interoperability Service* (SIS) to be the middleware that supports semantic interoperability in a DOM environment. (A similar service among databases will be called a database SIS; Section 3.2 briefly compares DOM and database SISs, and discusses the extent to which they can share components.) An SIS aims to insulate requests from the details of representation decisions and data semantics. With such a service, the requester can act as if the server uses the requester's own conventions. A few software products currently provide some of the services required by an SIS, and undoubtedly more will be developed in the near future. However, there has been no framework in which these various approaches can be analyzed and compared.

In this paper we provide such a framework, which we call a *reference model.* We use the reference model as a way to organize observations relevant to several major questions: How might a software vendor scope, specify, and build a semantic interoperability service? What metadata must an organization acquire to drive an SIS, and how may this information be

maintained? On what basis should an organization choose an appropriate kind of semantic interoperability service to employ?

The framework is being developed as an aid to advising MITRE's customers about the spectrum of interoperability approaches that may coexist in a large system, and about possible migration paths. Our taxonomies include components having less-than-ideal capabilities that a designer starting with a clean slate would reject; interim solutions often exhibit such imperfections. To make the advice easier to express, we seek to describe mechanisms and capabilities within a common reference model that makes it easier to avoid inessential distinctions.

Our hope is that the paper will be of use both to researchers and to those who are currently plotting their organizations' path to greater interoperability. For the latter, we include some discussions of information resource management consequences of various choices.

This paper is organized as follows. Section 2 examines the functional components of an SIS, and discusses the various design decisions available for each component. Section 3 considers architectural issues in the design of an SIS and compares with a database SIS. Section 4 examines the relationship between types and properties, and Section 5 presents our conclusions.

2 THE COMPONENTS OF A SEMANTIC INTEROPERABILITY SERVICE

This section examines the functional components of an SIS; collectively, these functional components define the reference model for describing an SIS.* After a short overview (Section 2.1), we examine the interfaces and capabilities of major SIS functional components. Our goal is to understand alternative levels of service from each component, dependencies among them, and how these decisions affect performance, administration, and evolution of the entire system. Section 3 will then examine ways that the components can work together to process a request.

2.1 SIS Overview

Semantic interoperability is needed when the sending application has different expectations than the receiving application. The actions that the SIS performs between the sending of an argument value v and the receiving of the converted value v' are called *mediation.* In brief, mediation involves determining the expectations that the two applications have for value v, choosing how to convert value v to v' (or raising an exception if conversion is not possible), choosing where and when the conversion shall occur, and arranging for the execution of both the conversions and the user-invoked function.

These mediation activities can be divided into four kinds of functional areas: a collection of *argument-describers* (one for each server function or client request known to the SIS), a library of *conversion functions*, a *planner*, and a *request broker.*

*A functional component may actually be spread over several software modules, as explored in Section 3.1. For example, the description of an application's arguments might combine knowledge from within the application, as well as from outside administrators. We are currently examining alternative implementation architectures.

An argument-describer reveals the expectations that an application (either client or server) has for an argument in a request (such as "display(mapOfBoston, 8.5, 11) in context of client myApp"). The describer is a function that may examine context (e.g., other arguments in the same request, contents of various databases, location, or user id) to return an *argument descriptor*, which describes the assumptions about this particular issuance of this particular request. A more detailed treatment of argument descriptors appears in Section 2.2.

A conversion function (e.g., to convert a document from LaTex to RTF format) is code that converts a value from the form described by one argument descriptor to a form having a different argument descriptor. In general, an SIS will have many conversion functions available, which may be provided by individual applications, outside sources (such as an ontology service [Collet91]), or even the SIS itself. In this paper we assume that the conversion functions available for mediation are stored in a *conversion library*. Section 2.3 discusses conversion issues more completely.

The planner is the component of an SIS that compares the client and server argument descriptors for each argument in a request and produces a *conversion strategy,* which is a sequence of calls to conversion functions. Planner issues are discussed in Section 2.4.

The request broker coordinates the activities of the other components, invoking them across the network. It includes the functionality of the Object Request Broker (ORB) in CORBA [OMG94]. The mediation-aware request broker can be implemented as an extension to a conventional ORB process, or as a separate process.

Sections 2.2–2.4 describe alternative interfaces and capabilities for each of these components. We omit the request broker, which would presumably conform to a standard such as CORBA, and also omit internal structure of each component and interface details. Our goal is a high-level understanding of how the design decisions affect SIS structure and behavior.

2.2. Argument Describers and Descriptors

For each argument value in a request, it is necessary to express the application's assumptions about the argument's meaning and representation. The argument describer is a function that determines the assumptions; the representation used to convey these assumptions will be called an *argument descriptor*.

The knowledge required by an argument describer must be supplied either by the application-developer or by SIS administrators; it may be expressed in an extended interface definition language, or may reside in a generalized, extensible knowledge base. This knowledge may also be provided declaratively about entire applications, particular users, users of a particular code module, and other convenient aggregations.

Designers of an SIS must choose the structure of an argument descriptor that is visible to the planner (issue D1, below), how an argument-describer determines the descriptor for a given argument value (issue D2), whether *all* arguments must have descriptors (issue D3), and how one manages the vocabulary used in descriptors (issue D4).

2.2.1. Issue D1: What is the structure of an argument descriptor?

The expressiveness of argument descriptors affects the ability of the SIS to plan conversions and to capture additional distinctions that may become apparent when new applications are added.

An SIS requires a descriptor format that will be very widely acceptable; it should be easy to understand, extend, and process. We therefore choose property-value lists as our candidate for wide adoption. While representations with greater power would occasionally be desirable (as discussed later), their greater complexity may reduce industry acceptance.

We now examine alternatives both weaker and stronger. Consider two applications that interchange documents via a request argument. We presume that they agree on the basic meaning of the argument (e.g., that the indicated file is to be edited and displayed) but not on details. To make sense of the transmission, the *partners* (sender and receiver) might need to mediate two kinds of detailed information about the document: the vendor format (e.g., WordPerfect, MacWrite, PostScript), and whether the document is uncompressed or compressed.

One possibility is for the applications to combine this descriptive information into a single string whose internal structure is not known to the SIS. In such a case, each partner must provide routines that map all necessary format information to and from the descriptor, and all partners must use compatible encoding schemes. In our document example, if only the above mentioned three products are supported, the applications could use six string values to encode the possible formats -- say, "WP/U", "WP/C", "MW/U", "MW/C", "PS/U", and "PS/C".

Unfortunately, such encodings hide information that an SIS could exploit. The planner would therefore be unable to understand that the conversion library function "compress" maps "WP/U"->"WP/C", "MW/U"-->"MW/C", and "PS/U"-->"PS/C". The above situation can be modeled more clearly by structuring the argument descriptor as properties that are visible to the SIS: *product* and *compressionStatus*.

In general, a *property* is a category of information that describes an aspect of the argument's semantics; this aspect may be essential in deciding how to use the data (e.g., measurement date, data source), or may just describe representational details (e.g., units, datatype) that conversion functions can hide. A *property-value* is a value for a property. The relationship between properties and types is explored in Section 5. An argument descriptor is conceptualized as a *set* of (property, property-value) pairs [Sciore94].

In the above example, a compressed WordPerfect document could have the one-property argument descriptor {(*docFmt*, "WP/C")} or the two-property argument descriptor {(*product*, "WP"), (*compressionStatus*, "yes")}. The latter descriptor is more expressive; description and conversion tasks are now *decomposable*, greatly simplifying their administration and use. Argument descriptors can draw on libraries for predefined, sharable property names and values. Conversion functions can be written to handle one property at a time, e.g., (*compressionStatus,* "yes")—>(*compressionStatus,* "no"), (*datatype*, string) —> (*datatype*, float), or (*units*, miles) —>(*units*, km). A moderately clever planner will generate conversion plans by composing these single-property conversions.

One can have argument descriptors with at least three levels of complexity – exactly one property, a fixed list of properties, or an extensible list of properties. The first level seems plausible when one seeks a quick extension to an existing request broker. Support for the second level allows for the decomposability described above. The third level allows an application to adapt to an evolving environment, in a way that we now describe.

When new applications join the network, they may make distinctions that were not anticipated when the encoding and property names were defined. If the list of property names is fixed, it can be extremely difficult to accommodate such additions. In the document example above, differences in the Windows, Macintosh, and UNIX versions of a product might require adding a property *platform* to the descriptors. For another example, a client-server order entry system in

the US might implicitly understand all currency values to be in $US. When a new customer in another country gets added to the system, all relevant argument describers need to be identified and then modified to support the property *currency*. This is much simpler if the property lists are extensible.

We therefore conclude that designers of a new SIS should plan on an extensible list of property names. An SIS product would then presumably include a starter set of property definitions and conversion functions; for example, the DoD standard 8320 defines properties such as units and precision. It would also include a mechanism for defining new values for old properties (e.g., the value "lightYear" for property *distance_unit*), new properties, and new conversion functions.

It is possible that some argument descriptors could have additional structure beyond what we have already described. For example, consider a statistics package or other server application that can accept input arguments in several different vendor formats. What argument descriptor should the describer for this application produce? Using the previously defined structures, the describer would choose the "native" format of the application, and have all conversion to this format performed by the planner. A new possibility is to allow the describer to return a *set* of argument descriptors, one for each format understood by the application. In such a case, the SIS would be able to take advantage of the conversion capability inside the application itself.

Another important case is that a recipient might describe acceptable arguments by a predicate on the descriptor, rather than by a descriptor. For example, if machinery weights have a meta-attribute error_bound, then an aircraft-loading application might insist that Weight arguments have (error_bound < 10%). Similarly, if one assigns different confidence to different news sources, some applications might exclude tabloids by insisting that (confidence > .5).

Finally, we note that there are reasonable situations that are not straightforwardly representable by a set of independent properties. Consider a system in which documents can be encrypted and can also be compressed. If both operations are performed, their order affects the result. In this case the describer would be better off returning a descriptor that is a *sequence* (not a set) of (property name, property value) pairs. The use of sequences would make it possible to distinguish "compressed, then encrypted" from "encrypted, then compressed".

2.2.2. Issue D2: When is each argument's descriptor determined?

For each client request (i.e. message instance) and each server's reception of a request, it is necessary to understand the assumptions being used by each partner. An *argument describer* can be thought of as a function that takes two inputs - a request (including context) and an argument of the request - and returns the descriptor for that argument. The describer must determine what properties are in the descriptor, and what values the properties have.

Documentation of server capabilities is roughly analogous to documentation of the contents of a shared database. A more daunting prospect is that, in principle, the assumptions for each argument in each a client request needs to be described. Not only will the calls be numerous, but they may be written by programmers for whom robustness is not a concern, and in languages that provide no means for describing the assumptions. Furthermore, the client may be simply a tool that is passing on arguments that it does not understand. These difficulties are serious, but there

are also some encouraging factors. First, one may be able to provide rules that apply very broadly (see item b). Second, client programmers have always needed to be aware of their assumptions when talking to a server, but this information is typically not documented, and is often lost.* We are not asking the programmers to perform new work, but just to document in a more reusable way work that they are already doing.

We discuss five options for where and when each property name or value may be determined, ranging from very early to very late binding.

- Property information may be implicit, hardcoded into an application. For example, all clients and servers of a home-shopping network might agree that prices are expressed in $US. This option has the drawback that it is hard to add a new client or server that does not share this assumption. Neither the argument describer nor the SIS administrator has easy access to knowledge that would enable mediation among partners using pesos, $Cdn, and yen.
- Property information may be determined by rules stored in a knowledge base. For example, a system for interchanging information about traffic violations might have the rule that US traffic officials report all speed properties (e.g., *speed_limit*, *measured_speed*) with the property value "milesPerHour". A Canadian administrator might supply an analogous rule saying that Canada uses the property value "km/hr". Because a rule can apply to many argument values, this option can substantially reduce administrative effort.
- Property information may be stored with the value. For example, explicit descriptors may be encoded in the name of the object (e.g. the suffix of a file name) or inside the object (e.g. the header of a file). In a relational or other record-oriented database, property values might be stored as additional attributes in a tuple.
- Property information may be negotiated when the applications first connect. For example, in the protocols Distributed Relational Database Architecture (DRDA) and HyperText Transport Protocol (HTTP), one or both participants provide a prioritized list of the formats they support.
- The property values may be generated at request time. For example, consider a brokerage system that displays prices in the national currency of the exchange on which the stock is quoted. The *currency* property of the result argument depends on the stock selected.

The number of properties that an argument describer chooses to place on data depends on the conceptual distance between the application and its partner in the request, and on the uniformity of the information being passed. For example, information passed to the Belgian military might need explicit descriptions for properties that are standardized and implicit among US Air Force units.

The SIS administrators may face practical limitations when supplying information. They may have no way to modify knowledge supplied with an application, since source code may be unavailable, or the application's host may deny write access. In such circumstances it is very helpful to have the ability to supply additional knowledge using external declarations. For security or performance reasons, these declarations may reside on the administrator's system rather than the application's. One may also mix the options, e.g., assume (*confidence*, high) but store document format explicitly.

---

*In client-server approaches, one often writes both client and server as part of the same application system; they build in knowledge of each other's idiosyncrasies. In a future world of component-based software, clients and servers are built independently, and will need to describe themselves better.

2.2.3. Issue D3: Are properties described for *all* the arguments in a request?

We have tacitly assumed that all values in a request are mediated by the SIS. However, less advanced SISs might convert only arguments that have explicit descriptors, or only the primary argument of an operation (as in Smalltalk dispatching). For example, one can imagine systems that convert the format of a self-describing "document" that is to be displayed, but not other arguments that describe window size and colors.

2.2.4. Issue D4: How does an SIS guarantee that both partners agree on the meaning of property names and values?

The meaning of an argument descriptor needs to be understandable to all participants, and its definition must be able to evolve. Thus the administrator who provides a describer for a new application needs to choose the property names and values from a vocabulary understood by the other applications' argument describers. It is necessary to avoid homonyms and to minimize or exploit synonyms, as discussed below.

When two applications use the same property name or property value to denote different things, we have the *homonym* problem; for example, describers from a land application and a naval application might both use *mile*, but would mean respectively conventional and nautical miles (roughly, 5300 versus 6100 feet). Such homonyms *must* be prevented, because they lead to wrong results. *Synonyms* are different names for the same thing; for example, if one application describer uses a property named *measurementUnit* to describe a value and another application describer uses a property named *distanceUnits*, then the SIS will not know that these properties are synonyms, and the planner will report failure.

An often-proposed (and occasionally implemented) approach towards avoiding such problems is for the argument describers to use a controlled vocabulary. That is property names and values must be chosen from an *ontology* -- that is, a set of concepts that are assumed known to all participants and provide a point of reference for all descriptions [Collet91]. For example, the ontology might include the concepts of Position, of Latitude-Longitude coordinates, and of Position represented using Lat-Long coordinates, of the values "kilometer" and "miles", as well as basic types like String and Float.

Administration of the ontology will be a key issue. New applications will require extensions to the ontology, but centralized control may create a bottleneck; also, systems that span autonomous organizations may have no central authority. When one wishes to combine systems that have different ontologies one can start by considering all their terms distinct, and gradually identify synonyms. Mechanisms will thus be needed to prevent homonyms and to identify and exploit synonyms; these may draw respectively on mechanisms that keep a network's site names unique (e.g., hierarchical naming) or that handle synonymous values in application data.

Additional problems may result if there is a new sender whose behavior the receiver did not anticipate. For example, the sender might send quantities with a status indicator ("operational" versus "exercise"), a scale factor, or a quality indicator such as "confidence". If the receiver were unaware of some of these conventions (e.g., data with descriptors "status=exercise" or "scale_factor=1000" or "source_confidence=0.5") then it may react very inappropriately. When one permits requests from a new source, one needs a way to add checks on additional properties.

The alternative of rejecting data whose descriptors contain unknown properties blocks requests senders who provide better than expected documentation of their data. (The receiver or sender might provide some advice about what may normally be ignorable.) Even more seriously, it also provides no easy means of protecting against senders who document poorly.

2.3 Conversion Routines

A *conversion function* is a routine that can transform the sender's value v to a new value v' that is semantically consistent with v but whose descriptor has different property values. The administrator who registers a conversion function vouches that the result is appropriately consistent with the input. Especially for complex types (e.g., maps, documents), conversions may be *lossy*, e.g., omitting information that cannot be represented in the result format.

Administrators provide a *conversion library*, which is a set of conversion functions. The planner may compose these to generate additional conversions (called *conversion plans*), either when conversion functions are registered into the library or during processing of an individual request.

We examine several aspects of the conversion library relevant to SIS designers: the amount of change that each conversion accomplishes (issue C1), how one describes the behavior of each conversion (issue C2), and especially the "topology" of the conversion set (issue C3). Section 2.4 will examine how conversion complexity impacts the planner.

2.3.1. Issue C1: How wide are the effects of a conversion function?

Although we defined a conversion function as a map between one argument descriptor and another, many conversion functions only affect a single property of a single argument value. A *property conversion* function takes as input an argument value v, an argument descriptor for the sender, and the desired property value for the receiver; its output is an appropriate value v'. For example, an application that takes GIF files as input and produces JPEG files as output is an *ImageFormat* conversion function. For another example, C-code that multiplies a floating point number by 1.6 is a *units* conversion function (from miles to km).

Although composing property conversion functions is a common and modular way to do conversion, there are cases where a more general conversion function is needed. Conversions that affect multiple *arguments* are important in MITRE applications, e.g., a conversion from rectangular (x,y) coordinates to (range, bearing). Other conversions transform one argument to multiple arguments; for example two sender's arguments, Latitude and Longitude may be mapped to a single receiver argument in Universal Transverse Mercator coordinates.

Conversions that affect multiple properties may also need to be described. For example, a single function might simultaneously convert two properties (e.g., *units* and *absolute_error*, where the latter is expressed in the indicated units). Also, legacy conversion programs (those written for use without an SIS) are likely to combine many tasks, including conversion of multiple properties.

The property values may be computed by the conversion, rather than being specified in the call. For example *absolute_error* above would be computed by the same conversion that handled units. Even for a single property, a conversion of numeric input strings might yield an integer or a float depending on magnitude and whether the integer is 16 or 32 bits.

2.3.2. Issue C2. How are conversion functions' limitations described to the planner?

For applications to be delivered to end users, it is often necessary to determine before run-time whether the conversion plans will execute successfully. The underlying DOM registers functions' argument types, but the SIS needs more detail. For static requests, the planner needs information about preconditions and postconditions for each conversion function. The SIS must thus provide a mechanism for registering the capabilities of the various conversion functions.

To describe such restrictions, the conversion library might provide predicates in the DOM's constraint language (e.g., the ODMG query language [Cattell93] or SQL) that describe each function's preconditions. It seems important for a planner to understand conversions' preconditions on the value v and its property values. Post-conditions are also needed, such as a specification of the descriptor for the value after conversion, as either a constant or a function of the incoming descriptor.

As examples of preconditions on the value v, note that large floats cannot be converted to type Integer, and only a limited set of 3-character strings convert to a two-digit integer. The same phenomenon arises in richer domains, e.g., a document converter that handles only text will lose information or fail if drawings or videos are embedded.

As examples of preconditions on properties, note that convertibility of one property often depends on the other properties. For example, if the sender's value has the format {(*encryption*, RSA), (*datatype*, integer), (*units*, meters), (*confidence*, .9)}, it is meaningless to apply a units conversion before the value is decrypted, and many conversion functions may work only on floats. It appears reasonable to allow preconditions that reference any argument or descriptor in the request; however, we suspect that simpler patterns will be common, as discussed in Section 2.4.

Frequently there is a simple value for a property that acts as a *base value* for conversions of other properties. For example, the base value for the property *encryptionStatus* is "none", because an unencrypted document does not constrain conversions of any other properties. If a base value can be found that satisfies the preconditions for many other properties, the planner will have a much easier time determining a good conversion plan. (However, at the end of 2.3.1 we saw that base values need not exist; e.g., compression may interfere with encryption).

A final sort of limitation is the degree of information loss associated with various conversion functions. Users may describe the maximum lossiness they can tolerate, and the planner would warn them. Information loss may include loss of precision (e.g., truncating trailing digits), of granularity (e.g., sales aggregated by day or by week), of certain kinds of information (e.g., labeling of keywords in a document), and no doubt many other dimensions.

2.3.3. Issue C3. Should there be a "standard" interchange format for each property?

Sections 2.3.1 and 2.3.2 discussed the possibility of single-property conversions, and how conversion plans could be generated from them. We now examine what patterns of single property conversions might be present in the conversion library. The issues raised apply also to systems without automated mediation.

Standardization of the interchange format is commonly touted as the best way to achieve interoperability. For a single property p, this means selecting a value p_0 as the "standard" through which communication should pass. Informal terminology makes it unclear exactly what adoption of a standard format means, though experts (both researchers and practitioners) typically have a good intuition. Due to the widespread advocacy of this approach, we felt it worthwhile to provide more explicit definitions and distinctions, and to analyze the consequences.

A *convertibility graph* for a property P shows the available conversions between values of P. The graph contains one node for each legal property value p_i, and an edge from p_i to p_j if there is a p_i-to-p_j conversion function for P in the conversion library. If there is also a conversion in the reverse direction, the edge is shown as bidirected. Figure 1 shows several examples of convertibility graphs.

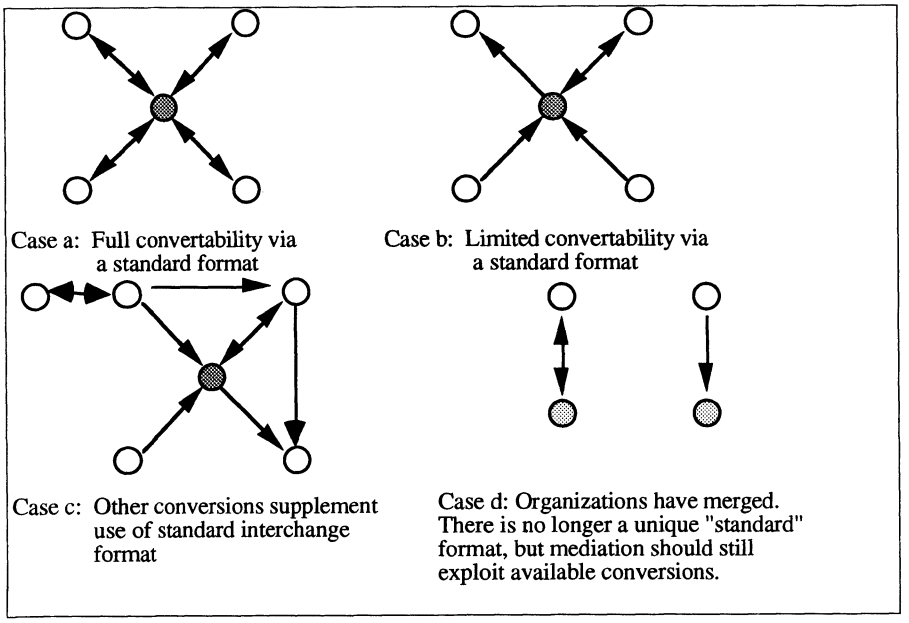

Figure 1 Various Convertibility Graphs.

Note that the picture is intended to describe conversions supplied by human administrators and SIS-developers. Conversion functions that are automatically produced by composition (either by a configuration-time administration tool or a run-time mediator) are considered to be products of the planner, and are not considered here.

In Figure 1, Case a shows a convertibility graph that might arise if P has a standard interchange value p_0 (shown as the shaded node in the center), and every other format is convertible to and from the standard. Case b also has a standard interchange format, but the conversion library is incomplete, perhaps because resources are insufficient to write all desired conversions. For

example, it may be that one imports documents (e.g., for display) but never exports them. Case c has conversions between nonstandard formats, and also a format that is not directly convertible to the standard. Case d shows a graph that might arise upon a merger of two organizations. Each organization had some interconvertible formats for property P, but there is not yet any conversion implemented between the two. Depending upon the usage history, one might get other patterns of available conversions.

To sharpen the discussion of the various alternatives, we split the issue into three subissues: First, should the SIS impose *prohibitions*, disallowing conversions that do not go to or from the standard form? Second, should the SIS impose *mandates*, requiring that for every nonstandard value of every property, the conversion library *must* include conversions to and from the standard form? Third, if there is a standard format, should it be expressive or simple? Our goal is to identify tradeoffs and to support our main conclusion, which is that neither the mandates nor the prohibitions should be imposed by the SIS architecture. Instead, administrators should supply whatever conversion functions they believe may be useful, and users should be kept informed of the degree of interoperability the SIS can provide.

Issue C3a: Prohibitions

First consider the prohibition of conversions that bypass the standard format. The simplification provides some benefits to the various communities. *SIS vendors* need provide only a simple planner, whose one tactic is to convert to the standard format and thence to the target format. *Application suppliers* and SIS *administrators* have fewer choices to make about what conversions to supply, and the number of conversions will not grow quadratically.

But the prohibition imposes severe costs, especially if the standard format is quite different from an application's native one. From the *user's* viewpoint, the resulting conversion plan may be slow and lossy. For example, suppose one needs to translate two rather similar formats (e.g., the Windows and UNIX version of some product). The simple direct conversion is prohibited, even if available from the vendor. Instead, one needs to use a conversion via a rather distant standard, with the likelihood of substantial overhead and information loss. The prohibition is even more debilitating when two organizations possessing different standard formats merge, since one organization's conversions will, by fiat, be unusable in this combined system. *Administrators* also suffer from the prohibition. They may now need to write a translation to the distant format.

A likely outcome of prohibitions is that administrators and users will circumvent them by building new interfaces to provide conversions that bypass the SIS, and hence cannot easily be reused. We therefore conclude that designation of a standard format should be advisory rather than prohibiting use of others. Management of a conversion library does not appear difficult, and in section 2.4 we see that a simple planner can still handle some plans that include nonstandard conversions.

Issue C3b: Mandates

We interpret a *mandate* "p_0 is the interchange form for property P" to be a requirement that the conversion library *must* contain conversions from each sender argument's value of P to p_0, and from p_0 to each receiver's form. Since conversions are not created instantaneously, this requirement implies that a sender or receiver's argument list may not be mediated by the SIS until the appropriate converters are available.

Such a mandate has several advantages. First, users know that requests will not fail for lack of a conversion function. Second, if the interchange format is also the "base" value with respect to other properties, then a conversion plan is at most two steps. Finally, the mandate gives clear direction to the administrator and to the provider of a new application. It seems quite reasonable to incorporate such mandates into acquisitions of new systems. However, applying the mandate to all participants may be a serious mistake.

The worst difficulty (seen by users) is the "all or nothing" nature of the mandate - full interoperability or nothing. Until all the mandated conversions are built, a new application is not legally part of the SIS, and hence *no* requests to the application will be mediated. In the extreme case when organizations merge (case d), they will vie to get their formats designated as standard; the loser will get no benefit from the SIS until all necessary conversions are provided. Again there seems to be a waste. It seems far better to design an SIS to do its best in the face of imperfection.

The mandate also raises difficulties for administration. The first difficulty is to get agreement on a standard format. This may be possible within a single organization, but tends to be very difficult across user communities and nations. Evolution is also a problem, since alterations to the standard format may require manual changes to the conversions supplied by many organizations. Also, for conversions that are not performance critical, it may be easier to provide a converter to a nearby format, that then can be further converted.*

With both prohibitions and mandates, evolution of the standard forms appears quite difficult. It seems likely that one will need to use a versioning strategy that violates the letter of the prohibition or mandate. Hence these restrictions seem better used as statements of direction than as actual requirements.

Issue C3c: Richness of the standard interchange format

If it has been decided to define a preferred interchange format, the next question is what it should look like. Should it be rich (very expressive) or plain (less expressive, perhaps a least common denominator)? The issue trades the interests of the different administrators versus users, and among different classes of users. The questions involved are not new, but may be of interest to information resource administrators.

Several kinds of richness arise, with orthogonal dimensions of complexity and precision. A rich format for documents might allow the document to contain many datatypes (e.g., tables, images), might provide many distinctions and useful connections (e.g., identifying section titles, or associating table or picture captions with the referenced object), or might simply be high precision (for images or numbers). In some cases it is difficult to provide a reasonable mapping to a plainer format . To the extent that a property identifies vendor formats, it is unlikely that one will be uniformly richer than the others. However, one can still talk about relatively rich or plain formats.

The general advantage of a rich format is that one can have other formats translate to it, without loss of information, giving users good service. Commercial products often provide rich kinds of documents, maps, etc. However, a rich format raises serious difficulties for administrators of both rich and plain argument values.

*While an administrator could provide a script that converts to the nearby format and thence to the standard, writing and maintaining this script will still require programming effort (especially if the conversions are located remotely).

First, it can be harder to agree on a rich format than on a plain one that acts as least common denominator. Every application-provider wants the interchange format to be rich enough for their product, structured in a way compatible with their product, and to omit features that their product does not exploit. Second, owners of plainer systems suffer. A converter from the rich format must add complexity to deal with all the constructs in that format. For example, if a rich format (e.g., LaTex) were used as the interchange format for all text, converters for mail notes or spreadsheet comments would need to handle all constructs in this rich format.

An SIS whose preferred format is plain may face the opposite problems. For example, conversion from a text processor to ASCII text is likely to lose the connection from table captions to the referenced tables. Typically many values in the rich format map to a single value in the plain format, so the lost information may not be recoverable if one attempts to translate to another rich format.

Therefore neither rich nor plain formats seem fully satisfactory, and no simple guideline can be issued. Instead we expect some organizations to adopt rich interchange formats, some poor, and some to use both types.

2.4 The Planner

The planner's power and design are dependent on the expressiveness of argument descriptors and the contents of the conversion library. We consider two issues: How the planner determines the appropriate way to convert from one property-value to another (issue P1), assuming no interference from other properties; and how the planner creates a conversion plan for several properties (issue P2).

2.4.1. Issue P1: How are single-property conversions planned?

There are several modules of functionality a planner might provide, many of which have appeared in products or prototypes. The simplest planner would simply look to see if a direct conversion existed, and otherwise fail. An alternative tactic (consistent with the prohibition in C3) would convert to and from the standard interchange format.

A planner that combines both of the above tactics offers *much* more power to meet users' needs, without significant extra complexity. Such a planner would need to choose among the alternatives; the choice could be heuristic (e.g., "direct is best"), or it could consider estimates of CPU time, lossiness, license fees, and network overhead (for converters that may not be available at all sites, e.g., decryption). Note also that it does not greatly complicate the planner to deal with imperfection – finding conversions if they are present and otherwise reporting failure. Hence even a simple planner with the two tactics mentioned can work with cases c and d of Figure 1.

More sophisticated planners could generate alternative conversion plans that do not use the standard form, or that include more than two steps. They would succeed with an individual property as long as the graph had a suitable directed path. Planning technology from artificial intelligence (e.g., [Wilkens88]) could be used to deal with interference between different properties' conversions, and with other pre- and post-conditions.

2.4.2. Issue P2: How are multi-property conversions planned?

We now consider the role of the planner in converting between multi-property argument descriptors. The simplest planner implementation would require that all properties be independent of each other, in the sense that converting one property does not affect the convertibility of any other properties. In this case, the planner is able to compose the property-level conversions (from P1) arbitrarily.

A slightly more sophisticated planner would also require independence, but would use auxiliary information (such as how much the conversion costs to run, what information is lost, what precision is lost, etc.) to determine the best composition of property-level conversions.

At the next level of complexity, the planner will be able handle interference between properties. One approach would be for the planner to require that administrators identify certain desirable behaviors for conversion functions. For example, property names might be orderable by resistance to interference, i.e., so that later properties (e.g., units or datatype) do not affect conversion of earlier properties (e.g., encryption). Another approach would be to specify a "base" value (as in C2) for each non-independent property; such a value would not interfere with any other conversions (e.g., once a document is decrypted, all other conversions can apply). If these resistance-ordering and base-value assumptions are known to hold, the planner can create a conversion plan to convert the most interference-resistant property to its base value, perform the other conversions (recursively), and then convert the property to its target value.

For example, suppose that the sender format is {(*units*, inches), (*encryption*, RSA)} and the receiver wants {(*units*, feet), (*encryption*, RSA)}. The planner at this level would determine that the property encryption has the base value "none"; it then would generate the plan that first decrypts the value, then does the units conversion, and then encrypts it back.

3 ADDING SIS FUNCTIONALITY TO A DOM

Section 2 examined the potential capabilities of each SIS functional component. This section examines the different ways in which this functionality can be mapped to the architecture of a distributed object system. The builder of an SIS must decide which system component is responsible for invoking and for supplying (and hence maintaining) each kind of functionality.

Section 3.1 examines alternative assignments of functionality to three different components of a distributed object system – the sending appliction, the ORB, and the receiving application. For each component, we discuss the range of SIS functional parts they can implement and invoke. We comment on how each choice affects evolution and performance, as well as the expressiveness needed by the intercomponent interfaces. A next step in our research will be to examine the specimen architectures that reflect the different choices . Section 3.2 explores possible relationships between a DOM SIS and a database SIS.

3.1 Capabilities of Each Implementation Component

Suppose that a sending application issues a request having an argument value v. In order for the receiving application to be able to see the appropriate value v', the following tasks must occur: Argument descriptors must be produced for all arguments, describing the sender's assumptions. Similarly, each argument requires a descriptor (or acceptability predicate) for the receiver. Finally, the planner must produce v', drawing on the conversion library if necessary. Each of

these functions may potentially be invoked by and (independently) supplied by either of the applications or by an extended ORB. We consider these possibilities in turn.

First, consider the applications, focusing on their ability to cooperate in one or multiple SISs. Many applications provide descriptors for their results, in either the name of the value or in its header. Applications also may provide some degree of planning and conversion. For example Microsoft Word 5.0 can write out documents (with some information loss) in roughly a dozen formats; on reading, it checks file descriptors or headers and can then read a similar number of formats. An SIS builder may choose to take advantage of these capabilities. However, the level of mediation functionality supported by legacy applications is typically low. For example, the descriptors produced by applications often consist of just one property, and conversions must be taken directly from a library (rather than composed). Also, if an application is unaware of the SIS, it will not invoke the SIS planner or external knowledge bases. Such applications would not know, for example, that the SIS supports a knowledge base of property rules (as in Section 2.2.2, item b).

In a distributed object system, a legacy application may have a *wrapper*, which is code that allows the application to interact with the DOM. It is possible to add code to the wrapper to provide describers for an application's arguments. For example, the wrapper could generate an argument descriptor by invoking an external knowledge base of property rules. Alternatively, code for computing argument descriptors can be embedded entirely within the wrapper, avoiding extra messages. Unfortunately, if a new client or server that requires a fuller argument description joins the system, existing wrappers that send or receive that argument may need to be recoded.

Wrappers can also build in some planning capability. For example, the receiving wrapper can compare a descriptor attached to a received value against the descriptor generated by the received application. If there are no discrepancies, such a wrapper could simply invoke the application; otherwise the wrapper might arrange some simple conversions, or call an external planner.

Now consider the ORB. Mediation can be transparent to the participating applications if the ORB invokes mediation and controls the necessary knowledge. For invoking mediation, one could define triggers upon each CORBA invocation and return event; it may be possible to do this over the existing CORBA standard (so that it will be portable). The knowledge base of argument describers and conversion function properties might be defined as an extension to the broker's interface repository. It is also possible to share responsibility, e.g., the ORB could supply mediation but applications could decide whether to invoke mediation, and could be responsible for providing part of the describers.

We have identified two difficulties here. First, an invoker (e.g., a statistical routine) may act as a blind conduit for some arguments it has received from other sources. If the mediator tries to reason solely from the identity of the invoker, it will be unable to determine all the assumptions that apply. Perhaps it will be necessary to restrict mediation to the case where invoker and receiver are knowledgeable. Second, under the current CORBA standard, if the invoker sends an integer and the receiver expects a numeric string, the compiler for the Interface Definition Language will raise a datatype-mismatch exception. Mismatches in number of parameters (e.g., Date versus Month, Day, Year) will also block compilation. We want to relax these restrictions without discarding type-checking completely.

Finally, applications and the ORB could share responsibility for invoking the functions. When an SIS is built gradually over existing interoperability mechanisms, the semantic information it contains is likely to be encoded using a mixture of techniques. For example, properties

considered when an application is defined (e.g., *datatype*) may be declared in the application, while other properties (e.g., *confidence*) may be available only from external knowledge bases. At times such distinctions should be hidden, since often an administrator will need an integrated view of all properties, regardless of where they reside.

There are also several implementation-level choices that might be transparent to applications: Should some mediation run within the same process as the requester or server wrappers to minimize overhead (e.g., to handle simple cases where no conversion is needed)? How should one distribute the knowledge about argument descriptors, the conversion library, planning, and the invoking of conversion plans? Should the various steps be computed when the request is executed, or in advance and shared among many requests? How can an SIS's administrator be helped to manage these decisions?

3.2 Openness to Other SISs

An organization might possess several distinct distributed object systems (under control of distinct SISs); for example, one might have both distributed databases and distributed services. Furthermore, an organization's objects might need to participate as service requesters or providers in a system that spans multiple organizations. It therefore seems important that the tailoring of applications and conversion libraries needed for the first system not be repeated in setting up others. Vendors of SISs will want to reuse software modules; in an ideal world, the conversion library and most modules in the planner would be shared between DOM and database SISs.

Some guidelines for promoting reuse are simple. For example, it is better to encode knowledge declaratively or as the result of externally-invokable services, rather than hiding such information within an application. If the ORB component is used to perform some or all of the mediation, for example, we saw that there was a choice of how closely to couple mediation to the existing DOM, and how much to make externally visible. Our view is that mediation knowledge and code can be important organizational resources, and should be accessible even when the ORB is not used.

There are two ways to present a database to the outside world. In the first approach, the SIS has no particular awareness of databases. Instead, a data administrator defines views and parameterized queries, and then wraps these as DOM-invokable functions. Parameters of such queries become parameters of the function that wraps the query, and the SIS will mediate different assumptions about arguments passed to these parameters. In this alternative, there may be tools to aid with the wrapping and in producing argument descriptors, but the SIS does not otherwise care that the server is a database.

An alternative approach is to build a database-aware SIS that understands the query language, complex arguments such as relations, and the descriptions of individual attributes. Much of the knowledge and tools that derive an integrated schema could be shared with a DOM SIS. A database SIS could permit a client to issue a query in terms of client relations. The SIS would use detailed knowledge of semantic connections between different databases (e.g., to resolve structural and naming heterogeneity), or between each database and a standard database. It would then produce an optimized query against server relations and translate results to client format. For example, the SIS might notice a clause "Where Speed > 100" in a US traffic application (which uses miles/hr) and convert to "Where Speed > 160" for execution within an SQL request to a database that uses kilometers. Such techniques are explored in [Sciore94, Daru95].

One could also imagine sending a DOM request whose major argument was of type SQL_string, and the SIS mediation of this "value" would alter the string to include the necessary conversions. However, such conversion is in essence a database SIS, and seems to gain little benefit from the DOM capability of mediating argument lists; furthermore, the mediated argument might need to be a script that included conversion routines in a conventional programming language in addition to SQL.

It is interesting to contrast the DOM SIS with one created for mediating user access to individual databases or to an integrating view in a federated database, e.g., [Daru95, ACM90]. With database requests, the task of understanding a received tuple is analogous to understanding a received argument list. In both cases one has a fixed length list of values, each associated with a different role, and needs application describers and a conversion library. However, it is not appropriate to issue a separate DOM request to retrieve and convert each output tuple; instead, one wants a single DOM request to return a tuple set, and then each tuple in the set can be converted as needed. Also, often all tuples in a result require the same form of conversion; in such cases, a single plan suffices for all result tuples. Finally, while one can pass a database reference as an argument, the descriptor of this reference does not provide the description of each individual attribute.

4 TYPES VS. PROPERTIES

Our proposed DOM has two means of describing data – types and properties. We have so far focused on the use of properties to encode semantic distinctions. This section describes reasons why we have not used types for this purpose.

Consider, for example, a value of type *document* having the property *docStyle* (e.g. LaTex, RTF, etc.). The style of the document could equivalently be represented as a subtype of *document* - that is, we could have subtypes *latex_document, RTF_document*, and so on. The conversion functions of Section 3 could be redefined to map a value from one subtype to another; in this sense, they would generalize the common notion of type coercion.

Figure 2 shows two ends of a spectrum of design alternatives. The dark arrow represents conversions performed by a mediator that examines properties of the sending and receiving applications and determines appropriate conversions. Alternatively, application properties can be captured as a set of type definitions, with mediation occurring through type coercions.

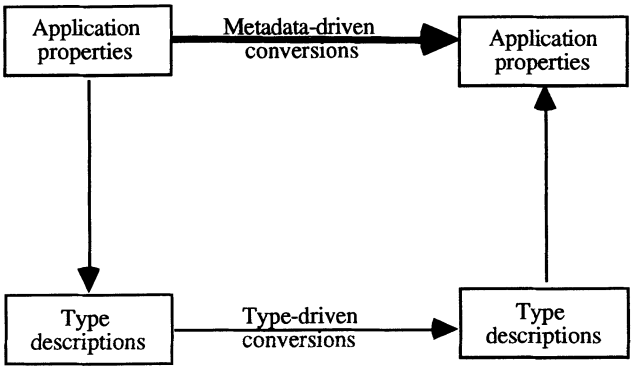

Figure 2 Design Alternatives.

It is tempting to use type descriptions to drive conversions. After all, types are already a part of a DOM, and the number of new concepts should be minimized. Moreover, type-driven conversions are familiar in many programming languages, and there is technology to infer result types and to determine whether compile-time type checks are possible. However, we believe the counter-arguments are stronger. In particular:

- Type conversion is not provided by an ORB - an ORB only provides independence from machine and programming language differences for a simple set of types and constructors; it does not provide even simple type conversions, such as from Integer to String. Extending the ORB to handle type conversions (especially for user-defined types) seems little easier than adding a modular service that converts based on metadata that is outside the type system. Furthermore, we do not want to tinker with the existing CORBA 2.0 standard.
- Because each CORBA object must be an instance of a single type, the number of types will grow explosively, potentially as the product of the number of property values. The type hierarchy would no longer be appropriate for its main task of organizing and browsing object classes' attributes and methods.
- For example, consider the above document example, and suppose that documents also have the properties *source* (e.g. IEEE, ANSI, DoD, etc.), encryption (e.g., RSA, Clipper), and alphabet (e.g., Latin, Arabic, Kanji). A type-based encoding of these properties would result in subtypes such as *latex_ieee_rsa_latin_document.* Like the format encoding that was a single string (Section 2.2.1), this collection of types hide the true structure. In addition, one would need to alter type browsers and other tools to display the essential structure.
- A type hierarchy's semantics (IS-A relationships) may not be appropriate. IS-A provides inheritance only for declarations and signatures for attributes and functions. For knowledge about properties, one often wants to inherit property values (e.g., that distance_unit=miles). Also, hierarchies seem ill-suited to bidirectional conversion functions.
- Conventional type conversions (as performed in C, for example) are within a single set of type definitions. However, in a large information system (e.g., the Internet) there will be no single set of types known to all systems. The semantics and implementation for exception handling

and run-time binding of conversion functions may also be inappropriate. Hence it may be difficult to import the technology typically associated with types.

Thus it seems unreasonable for an SIS to try to do everything through the type system.

The opposite approach is for the SIS to ignore the type system, and to drive mediation exclusively by property information. This has been the implicit assumption of Sections 2-3 of this paper. The best alternative is probably for an SIS to combine type-based mediation with metadata-based mediation. For example, one might provide tools that extract relevant information from type descriptions and produce appropriate property-value pairs.

5. SUMMARY AND FUTURE WORK

5.1. Summary

We have described many of the functional and architectural options for a Semantic Interoperability Service. We addressed questions stemming from three perspectives – SIS vendors, providers of applications, and user organizations that wish to build an interoperable system of distributed objects.

Our main goal is to serve large organizations that would use an SIS to mediate an interoperable system of distributed objects. Our categories can help such organizations to describe their current arrangements, the desired cleaner and more flexible arrangements, and various intermediate stages. The many examples and the tradeoff analyses can help in identifying consequences of choices among the many alternatives, for each decision.

Since we cannot predict all future interactions, we have emphasized that knowledge and code should be reusable, modular, and extensible [Rose94]. This requirement suggests strategies that provide declarative knowledge (from both wrappers and external knowledge bases). The knowledge should be available to multiple planners, possibly in different SISs. One wants conversions that bypass the standard interchange format, conversions that handle one property at a time, and the ability to introduce new properties and mediation without changing existing applications. (For example, cross-border requests may require definition and mediation for an unanticipated property, speed_units). To keep the knowledge consistent, the organization should maintain an ontology of property names and values, and a means of recording partners' use of synonymous terms. When new applications are acquired, they (or their wrappers) should be required to provide argument describers that use this ontology.

SIS vendors can use the same formulations to make decisions and to explain these decisions to their customers. Decisions that vendors are likely to make include the default place for knowledge to reside, the contents of the built-in conversion library, how rich a formalism is used for describing arguments and conversion functions, and the detailed responsibilities of each kind of component. For providers and clients of application objects, we identified some descriptive information that each application (or wrapper) should make available to any SIS in which the object is embedded. As ontologies of properties grow, the descriptions should be extended.

Finally, mediation among distributed objects should not be addressed in isolation. Interoperability facilities will need to grow to include database queries, so we discussed overlaps with mediation functionality for databases.

5.2. Future Work

As part of MITRE's Distributed Object Management Integration System (DOMIS) project, we have encapsulated several large (> 10^5 lines each) application systems and established communication using CORBA. We hope to rearchitect some of these applications as collections of services, to improve the ability to reuse part of the functionality in an application. After such a reorganization there will be more service requests, so mediation of service requests (a DOM SIS) will become more important.

We have implemented a rudimentary database SIS, and are implementing a DOM SIS using many of the same components over Iona's ORBIX request broker. ORBIX provides hooks for trapping the receipt or completion of a request, but it still may be necessary to modify the requester to provide knowledge of the assumptions. We also expect performance to be an issue. Currently we are considering a technique analogous to precompiled database queries – if a request is expected to be issued frequently, the planner may run in advance and cache its plan at either the sender or receiver site.

Issues of administration, scalability and interoperability still need to be confronted. For administration, one needs methodologies and tools for gathering and maintaining the semantic knowledge. Also, the descriptive constructs require some extensions, e.g., to model lossy conversion functions. For scalability, the knowledge base will need to be very distributed, and we need specifications about how organizations with different ontologies should interoperate. Protocols for interoperability among SIS vendors will be needed, e.g., to address standards for exchanging descriptors, and shared responsibility for providing knowledge and invoking mediation.

Based on our implementations and on paper analyses, we will continue examining semantic interoperability issues. We hope to determine how much flexibility can be obtained over the current CORBA specification, and to suggest extensions if necessary. As a reality check, we will examine the utility of such a service among several existing Air Force planning systems. Finally, we intend to explore how the presence of such a service affects the acquisition of component systems and the responsibilities of a system integrator.

6 REFERENCES

[ACM90] *ACM Computing Surveys, Special Issue on Heterogeneous Databases*, S. March (ed.), Sept. 1990.

[Cash94] Cashin, J. and Gilhooly, K. "Lack of Open Systems Standards Giving Apps that Layered Look", *Software Magazine*, Sentry, August 1994.

[Collet91] Collet,C., Huhns, M. and Shen, W "Resource Integration Using a Large Knowledge Base in Carnot," *IEEE Computer,* Vol. **24,** No. 12, December 1991.

[Cattell93] Cattell, R. (ed.), *The Object Database Standard: ODMG-93.* Morgan-Kaufmann, San Mateo, CA, 1993.

[Daru95] Daruwala, Adil, Hian Goh, Cheng, Hofmeister,Scott, Hussein, Karim, Madnick, Stuart, and Siegel, Michael. "The Context Interchange Network", IFIP WG2.6 Working Conference on Database Semantics (DS-6), 1995.

[OMG94] Common Object Services Specification, Object Management Group, Framingham MA, 1994.

[Rose94] Rosenthal, A. and Seligman, L. "Data Integration in the Large: The Challenge of Reuse", *Conf. on Very Large Data Bases*, Santiago Chile, Sept. 1994.

[Sciore94] Sciore, E., Siegel, M. and Rosenthal, A. "Using Semantic Values to Facilitate Interoperability among Heterogeneous Information Systems," *ACM Transactions on Database Systems,* June, 1994.

[Wilk88] Wilkens, D., *Practical Planning*, Morgan Kaufmann, 1988.

Acknowledgments: The work described in this paper was performed by The MITRE Corporation under the Distributed Object Management Integration (DOMIS) project which is funded by the Air Force Electronic Systems Command, and at Boston College. The project office is at Rome Laboratory, Griffiss Air Force Base, Rome, New York. The project officer is Carl DeFranco, RL/C3AB. We have received very useful suggestions on this paper from Thomas Brando, Sandra Heiler, Myra Prelle, James Scarano, and Leonard Seligman.

7 BIOGRAPHY

Arnon Rosenthal works on federated and object-oriented information systems at the MITRE Corporation. His research interests include heterogeneous databases, object-oriented databases, query processing, database security, and database design. Earlier interests included discrete algorithms (especially dynamic programming) and complexity theory with applications to networks and reliability. He was previously employed at Xerox Advanced Information Technology (formerly part of Computer Corporation of America), ETH-Zurich, Sperry Research, and on the faculty of the University of Michigan.

Edward Sciore is an Associate Professor in the Computer Science Department of the Carroll School of Management at Boston College. He received a B.S. from Yale University in 1976 and a Ph.D. from Princeton in 1980. His research has covered a diverse range of topics, including data modelling, database query languages, query optimization, object-oriented databases, and distributed databases. Prof. Sciore is a member of the Association for Computing Machinery, the IEEE Computer Society, and Computer Professionals for Social Responsibility.

Questions & answers

Question []:
(property name value) triples do not work for derivations. Why not create a property for a derivation tree?

Answer [Arnon Rosenthal]:
It would not be meaningful.

Question []:
Do you put that information in the conversion function?

Answer [Arnon Rosenthal]:
We want everything to be understood in the conversion functions (see paper for examples).

Question []:
How are planning and execution interleaved?

Answer [Arnon Rosenthal]:
This is beyond the scope of our paper for various reasons such as the fact that we use shelf bought optimizers.

Question []:
How does the system know who to send a request to?

Answer [Arnon Rosenthal]:
We assume the requestor knows who to send a request to. The server must be bound at some level.

Question []:
In your architecture, is it not true that the knowledge base needs to know about every application?

Answer [Arnon Rosenthal]:
The knowledge base must be modified for each application. The knowledge that would be embedded in program code is encoded in the knowledge base.

Question []:
Why not include properties in the type system?

Answer [Arnon Rosenthal]:
We chose not to overload the type system. In some cases we could overload the type system, but the products we are developing do not fit these cases.

Question [Michael Siegel]:
Have you considered how this will scale-up for a large programming environment?

Answer [Arnon Rosenthal]:
This is beyond the scope of our research. We have considered these issues at a high level (e.g., the organizational level), not the programmer level.

Question [Sham Navathe]:
What type of technology is the broker problem?

Answer [Arnon Rosenthal]:
Enabling technology.

Question [Arnon Rosenthal]:
You have addressed a large number of issues. How do all of these issues fit together?

Answer [Arnon Rosenthal]:
A modular service approach is better than integrating all of these problems together.