Exploiting Structured Data in Wide-Area Information Systems

John Ockerbloom

August 1995

CMU-CS-95-184

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This is a reprint, with minor corrections, of a paper that informally circulated in March 1995.

The research reported here was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330; by National Science Foundation Grant CCR-9357792; and by a grant from Siemens Corporate Research. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ARPA, the National Science Foundation, Siemens Corporation, or the United States Government. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon.

Keywords: information system applications, distributed applications, abstract data types, data storage representations, object-oriented methodology

Abstract

Information produced by outside parties, such as in the World Wide Web, is increasingly important in many software applications. Effective use of this information requires the ability to exploit its semantic structure. Unfortunately, existing wide-area information systems force data to be distributed in either a lowest-common-denominator form, or in a form meaningful only to programs designed around a particular application. The former results in significant loss of information, while the latter severely limits the information available.

One solution is to develop a way to describe information types with rich semantics, allow new type descriptions to be easily added and related to existing ones, and construct agents that can distribute and interpret this type information. The extra weight of general-purpose network object systems like CORBA and OLE is not required. Providers can create and manage their data and data types as they wish, and clients can adapt them to their needs.

The Typed Object Model (TOM) implements such a solution. It treats information as typed abstract data objects, with encodings defined for compatibility with existing infosystems, such as the Web. Mediator agents register and relate new data types, and locate agents that can interpret or convert unfamiliar data formats.

1 The Problem of Widely Distributed Structured Data

The size and rapid growth of the Internet has made it into a vast store of information of all types. The NSFNet portion of the Internet carried 21 Terabytes of traffic in January 1995, more than half of it related to information services.

Many software applications are now being designed to work with this information. Many of them require this information to have a rich semantic structure. For example, a medical researcher may want to examine blood pressure readings from a clinical sampling and correlate them to heart attack occurrences, using the structure of patient medical histories. A scientist may want to find books in several libraries about plate tectonics, using catalog entries and search indexes. A software engineer may wish to find and examine C++ modules for processing SQL queries, using the structure of program archives and descriptions.

Within a small project, or tightly controlled application, this rich semantic structure can be determined by simple agreement. A few common data formats and standards are defined, agreed on, and then used for further analysis and interchange.

When information comes from a wide range of sources, though, such consensus on high-level data formats does not exist. Instead, data is transmitted in a lowest common denominator form. The explicit structure is often simply ASCII text; or, it may be a slightly more structured form, like HTML, PostScript, or one of a few dozen MIME [BF92] types. This level of abstraction, when available, is typically still well below the levels of abstraction needed for analytic applications. For example, a set of medical histories in unstructured plain text or PostScript is harder to analyze than a structured document where significant diagnoses and clinical measurements can be automatically extracted.

Continuing the current practice of defining standard data formats will not solve this problem. Even if higher level standards are adopted, this process can take years, and will still only cover the most common applications. (And many of these "standards" will remain unrecognized by many tools, or eventually replaced by other standards.) New applications of semantically structured data are being continually introduced, much faster than any standards body can be expected to handle.

Now suppose that we distribute this "standardization" process. Instead of using a centralized standards body for data types and formats, we allow anyone on the Internet to define a new "standard" for a useful data type, and relate it in various ways to existing data types and formats. Furthermore, suppose we allow this type information to be easily propagated and shared, and introduce mechanisms to allow applications unfamiliar with a new type or format to use objects of that type, either by remote operations or by conversion to a familiar type. If the present-day World Wide Web has made on-line information (of a few selected types) ubiquitous, this decentralized approach would make type information ubiquitous as well, and potentially raise the level of semantic information interchange on the Net significantly.

The following sections of this paper will describe the issues, design, and implementation of such a system. First, though, I briefly summarize some relevant related work.

2 Related Work

There is a rapidly expanding body of research on distributed information systems. However, there are five areas that are most closely related to the work in this paper:

Multimedia Information Retrieval: The World Wide Web [BLCGP92] exemplifies the current state of the practice in wide-area information retrieval. It allows a wide variety of popular information formats to be served via several common protocols. The Web defines a new data type (HTML) for its own use, and can handle other types via the MIME [BF92] typing system. The Web's ability to encompass existing Internet protocols and common data types, its support of multimedia data types, its straightforward (if simpleminded) document referencing scheme, and its minimal assumptions about underlying data storage have helped make it the fastest growing service on the Internet at this writing. However, the range of information types the Web can handle is limited, both by the limited semantic expressiveness of HTML, and by the limitations of the MIME typing system (described below).

Data format standards: The MIME typing system, described in [BF92], was invented to allow various types of multimedia files to be sent via electronic mail. It is now also used in the Web and in Gopher+ [Adi94]. Some of its design features are worth noting: it organizes type names into a simple hierarchy (for instance, the names of all image types start with "image/"); it recognizes that the same type may be encoded in different ways; and it has an informal mechanism to allow anyone to add "experimental" types. However, these concepts are not taken as far as they need to go. The hierarchy is controlled centrally, with no way of using "experimental" types except by informal word-of-mouth convention or by central registry. The encoding schemes are also fixed, with only a single level of encoding used to translate 8-bit data into ASCII forms suitable for mailing. More general notions of encoding, and multiple levels of encoding, are needed. (This is already a noted problem in WWW, where, for instance, there is no universally accepted way to identify the form of a compressed PostScript document.)

Distributed Objects: A number of systems allow objects to be manipulated by arbitrary machines over a network. The CORBA proposal [Gro92] of the Object Management Group is probably the best-known example; its core Object Request Broker standard is available today. While CORBA is a useful model for general-purpose distributed programming, it is less useful as a basis for world-wide structured information dissemination. Many services expected by CORBA, such as lifecycle management and event services, simply aren't supported by many information sources one might want to interact with. Some desired services, like the migration and replication of information objects, are difficult to handle in CORBA. Essentially, the requirements of systems that want to retrieve and analyze data from a wide range of sources, and those that want to modify data in place, are different.

CORBA proposes "interface repositories" that can hold information about new object types. The repositories are passive, though, and not designed for global use.

Systems with type expertise: Expert agents for information handling have been proposed in many papers, such as in Wiederhold's mediators paper [Wie92]. Rufus [SLST93], a system developed at IBM Almaden to manage semi-structured information, includes such an agent for abstract type services. Rufus creates structured abstract objects as "proxies" for unstructured data files that encode the abstract objects. The expert agent, known as a "classifier", analyzes the file contents to select a type to use for constructing the object. The paper [SS94] describes an algorithm that allows the classifier to learn to classify an arbitrary number of new types. Rufus is designed for a single site, and in its current form does not scale up to Internet-wide information systems.

Extendable types: Most distributed object systems allow an unlimited number of structured data types, but typically lack general run-time services to assist applications in using unfamiliar data types. A number of systems, however, give more support. SGML, a well-known text markup convention described in [Pub86] and elsewhere, allows syntactic descriptions of new data types (known as DTDs) to be passed along with data objects, so that arbitrary applications can parse them, as long as the object format follows certain basic markup conventions. The DTDs are essentially syntactic, not semantic. While they allow a program to extract the structural elements of the SGML files, they do not specify what can be done with the structural elements.

3 Elements of a Solution

When designing methods to take advantage of semantically rich data structures, we can take advantage of the following four principles:

1. *What can be done with data is generally more important than the particular format used for the data.* If a program needs to display an image, or analyze a graph, it matters little what the format of the underlying image or graph data is, so long as the program can do the display or analysis. Hence, it is useful to treat data *abstractly* when possible, so as to avoid concern with unnecessary details. This implies that one can treat information types similarly to the abstract data types of programming languages.

   Data formats generally represent an *encoding* of abstract data types as a lower-level type. Operations on an abstract data type are often performed by manipulating the encoding. Many common data types are encoded in just one format, as a sequence of bytes, but this is just a special case. It is possible for the same piece of abstract data to be encoded in multiple ways (e.g. an integer being encoded as big-endian or little-endian byte sequences), and to have several layers of encoding (e.g. a complex number encoded as a pair of reals, encoded as a sequence of bytes.)

2. *Unfamiliar data formats are still usable if they can be related to familiar ones.* This can be done in several ways: by changing the unfamiliar data format into a similar, familiar format (e.g. converting a GIF image into a JPEG image); by treating the object via a known abstract interface, and letting someone else handle the implementation details (e.g. searching an opaque database with SQL commands); or by operating on the object's encoded form rather than its abstract form (e.g. compressing a byte sequence representing an object, without knowing or caring what the byte sequence represents.) We will see later how these different techniques can be exploited.

3. *The Internet includes not only many data types, but also many potential expert agents.* People routinely install information servers on the Internet to deliver various types of data via WWW, Gopher, or WAIS. With a little more effort, one can not only deliver data, but also interpret it. Someone who defines a new data type, or is simply interested in using it, can offer to interpret its interface to outside users as well, or convert it to other data types, if the right infrastructure is in place. Indexing the data types and services in well-defined places allows the appropriate services to be located.

4. *Focusing on the data itself, rather than on its storage or modification, is sufficient in many cases, and simplifies issues.* As distributed database researchers know, applications that modify remote data must overcome a number of challenges. A system must be constructed that guarantees that remote agents will change the data in the manner expected, and that properties of remote transactions like atomicity will be honored. The data types themselves must carefully specify what their modification semantics are.

   But in many applications, remote mutation is not necessary; it is enough to gather information, analyze it, and generate derived information. If we restrict our outlook to this domain, we become much less concerned with the underlying storage models of remote sites, or their ability to engage in complex transactions. As long as a site delivers us data in some well-defined fashion (whether it be through an existing protocol like HTTP or WAIS, or a newly designed protocol) it can participate. Our focus can then be on the data itself, rather than on where and how the data is stored.

Many of the principles of decentralized data type registration apply both to systems that mutate data remotely and systems that do not. The system described on this paper, though, is of the latter kind. We will see later in this paper how this allows us to interoperate with existing systems and relate types in ways we might not otherwise be able to.

To sum up, these principles inform our design thus: Principle 1 leads us to use encoded abstract data types as the basic model for information. Principle 2 leads us to organize the data types into relational graphs. Principles 2 and 3 lead us to support remote services on data, and build agents that can store type information and mediate for remote services. Principle 4 leads us to consider models that treat information as values, rather than as variables.

4 TOM: Design of a Solution

4.1 Overview and Example

With the principles above in mind, we now introduce the Typed Object Model (TOM), as a way to manage a growing set of data types scalably. Briefly stated, if one wants to introduce a new type of information, one simply creates an abstract interface for it, and one or more encodings if the type's objects are to be transmitted over the network. One then registers this information with a mediator agent called a "type oracle". [1] One can relate the new type to other types by subtyping from a known type, encoding the new type as a known type, or creating conversions from the new type to other types. One can also put up servers that implement the abstract interface of the type or carry out conversions, and register the servers with the type oracle as well. Type oracles can exchange type information among themselves, and interested clients can invoke the type oracles for help in dealing with a new type. In this way, the universe of usable semantic data types can grow in a distributed fashion. Figure 1 gives a conceptual picture of the computation model.

An example will help in understanding the model. Suppose that one wishes to create a new type for citations of on-line research reports. One could define an appropriate interface (with attributes like "authors", and methods like "find a copy of the report"), devise a suitable format for encoding the object as text, and then register this information with a type oracle. The type could be a subtype of a more generic "bibliographic-citation" type, if the appropriate interface was supported. To help in using the type, servers could be set up that implemented the new type's methods, and converted instances to and from related types like "MARC record" and "BibTeX entry".

Suppose a client looking for citations comes across an object of this new type, and is unfamiliar with it. The client consults a type oracle to find out more information about it. Using the information in the type description the type oracle returns, the client might see that the new type is a subtype of "bibliographic-citation", a type it knows about, and then proceed to use the object through its supertype's interface, invoking remote servers to compute attributes and methods as needed. Or, it could decide to convert the object to the known "BibTeX entry" type and then deal with the object locally. Or, if it simply wanted to display the object, it might decide to simply print the text used in the encoding the object.

While the basic ideas of abstract types and type oracles are relatively simple, the details are nontrivial, and crucial to the success of the model. The basic design concepts are summarized below, and described in more specific detail in the subsections that follow.

[1] Type oracles should not be confused with the omniscient, imaginary "oracles" of theoretical computer science. They are actual computing agents that manage type information.

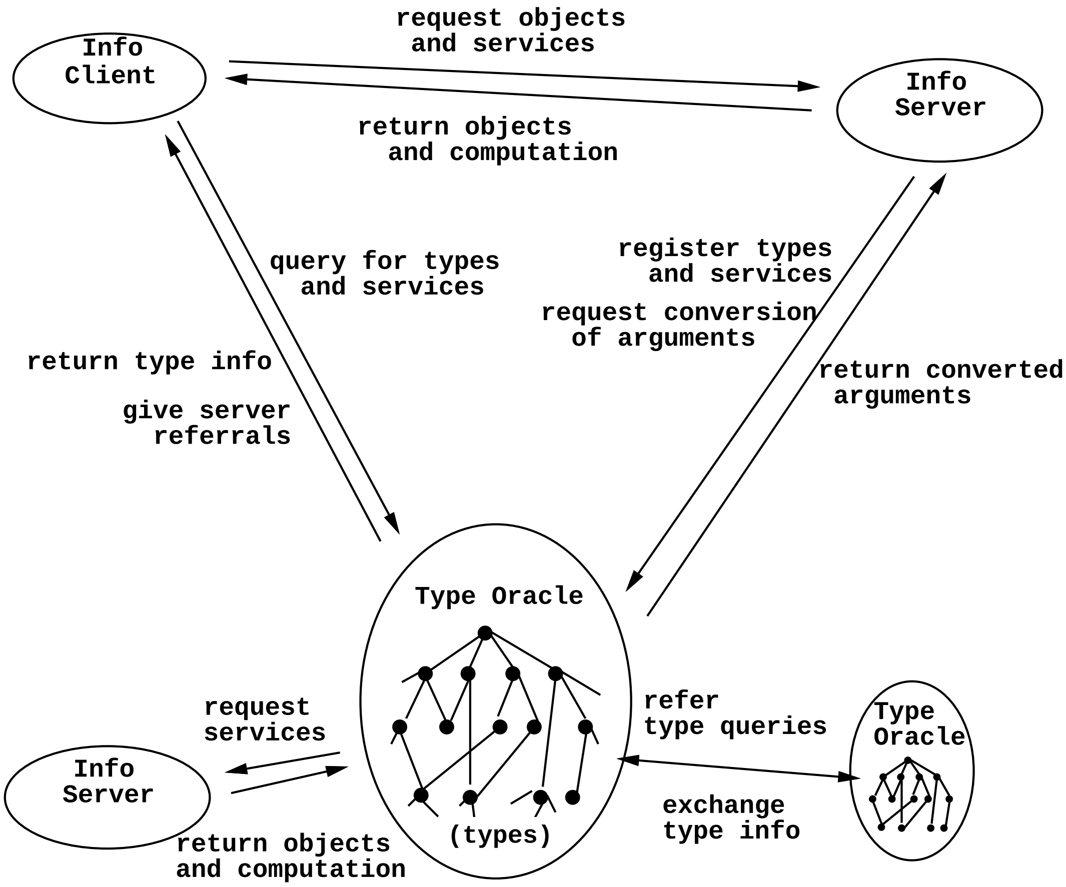

Figure 1: Agent interaction in TOM

In TOM, information is modeled as typed abstract data objects, where objects are values rather than storage locations. Abstractly, each type has a set of attributes and methods, and the set of supported methods is allowed to grow, with certain restrictions. Concretely, types have a number of encodings that allow objects to be shipped across the network, treated at a lower level of abstraction, or obtained from systems that don't directly participate in TOM. Types can be related by subtyping, encoding, or conversion. Type oracles are used to record and disseminate information about data types, and locate agents that can perform services on objects. Type oracles can also use their knowledge of the type graph to recommend particular courses of action.

4.2 Objects, Types and Encodings

In TOM, all information is considered to be an object of some type. Even sequences of bytes are objects. (They have attributes like "length", and methods like "get the n'th byte".) In contrast to the practice of many object systems, objects are values, not variables or locations. Hence there is no such thing as mutating an object, just as there is no such thing as turning the integer 3 into the integer 4. On the other hand, copying an object, or obtaining an object from an arbitrary source, is easy.

Many object types have *encodings*, which are relations between objects of one type and objects of another, generally more "concrete", type. (The concrete object must contain enough information to reconstruct the original object.) A *format* is a particular abstract type and sequence of encodings that, when composed together, represent objects of the abstract type as a sequence of bytes. Once objects are encoded as byte sequences, they can be copied and shipped over the network, as long as information about the abstract type and the encodings is shipped along with the encoded byte sequences.

In order to pass along this information, we need ways to unambiguously refer to particular types and encodings. Therefore, every type has one or more globally unique names. These names can be used to look up information on a type, once the type has been registered with a type oracle. Similarly, every encoding defined for a type has its own name.

Uniqueness of type names is enforced by the use of namespaces, denoted by the start of the name. For example, names starting with "e:" are assigned by the creator of TOM to types essential for the operation of the system. ("e:byteseq" is the name of the basic type for sequences of bytes.) On the other hand, names starting with "net:" are assigned by individual type oracles, which can generate unique names similar to the unique Message-ID's that are assigned to mail and news articles. More namespaces can be created and managed in various ways, if more readable type names are desired.

4.3 Object Attributes, Methods, and Semantics

Type names and encodings help us classify data, but don't say what can be done with it. When a type is registered, one can give specifications for attributes and methods that the type supports.

Attributes are functions that, given an object of the appropriate type, return another object. For instance, a "Gregorian-Date" type might have a "day-of-month" attribute that returns an integer from 1 to 31.

Methods are functions as well, but they can take a number of additional arguments, and can depend on some implicit context for their results. (For instance, a "days-from-now" method on our date type might depend on the current date.) Methods can also fail when their preconditions are not satisfied. Methods can return objects, but cannot mutate them (since objects are just values).

Thus, methods can be thought of as "call-by-value". Like attributes, methods can be computed by any agent on the network that knows the appropriate object values and context.

The behavior of an attribute or method is constrained by its semantics. Semantics can be associated with individual attributes and methods, or with a type as a whole. (Semantics can also be associated with other aspects of a type description.) To allow maximum flexibility in specifications, we mandate no particular format for semantics. They can be expressed as any type of information object desired, whether it be a formal notation or informal text. However, since agents can implement attribute and method functions in any manner consistent with the stated semantics, it pays to be formal and precise in semantics expressions. Using formal semantic objects can also allow a limited amount of automatic checking that would not otherwise be possible.

Methods and attributes can be invoked with references to objects rather than the objects themselves. This can be useful when it is impractical or impossible to ship an argument from client to server. References are just another kind of object, with a "fetch" method that returns the object the reference points to. The result of the fetch method is highly dependent on the context and the type of reference used. For example, the result of "fetch" on a reference of type "URL" depends on the state of a particular Web server, and has no necessary relation to the results of subsequent fetches on the same URL object. Other types of references can specify different behaviors in their semantics.

4.4 Relations: Subtyping

TOM supports a subtyping relation. Subtypes must be *substitutable* for their supertypes. That is, they must support all of the attributes and methods of their supertypes, and satisfy the same semantic constraints. They may support additional methods and attributes, or refine the specifications of their supertypes. Since subtypes can carry more information than supertypes, they do not inherit encodings. Types may have as many supertypes as desired, as long as they satisfy all the appropriate constraints.

This notion of subtyping is more restrictive than that found in many programming languages like C++ [Str91]. But because it lets subtype objects substitute for supertype objects, it allows agents to use any subtype of a known supertype, even if the agents are unfamiliar with the particular subtype. Furthermore, because objects do not mutate, in practice the subtyping rule is much less strict than normal substitutability models (like that expressed in [LW94]), which have to worry about any history properties. Hence, it's perfectly legal for, say, a sequence of integers to be a subtype of a sequence of objects. (In a world of mutable objects, this would not be allowed under strict substitutability, since an arbitrary object can be added to a sequence of objects but not to a sequence of integers.)

Not all types in the subtyping graph have to be instantiated. Indeed, it may be useful to invent new abstract supertypes just to capture the common properties of a set of existing types.

4.5 Relations: Conversion and Encoding

Support for conversion between types and formats allows agents to work directly with objects they might otherwise have to use remotely. Each type can have any number of named conversions between its formats and other formats, either of the same type or a different type.

Many conversions involve some degree of information loss. A programmer with knowledge of the various conversions can choose to use conversions that preserve desired information. But sometimes conversions need to be chosen automatically. In many such cases, the type hierarchy itself can be used as a measure of information loss. For any conversion from object O_1 to object O_2 there exists a "top" type T, such that O_1 and O_2 are indistinguishable if viewed through the interface of T. For instance, a conversion from one format of T to another format of T will generally have T as its "top" type. An abstraction mapping from an object of type T to a supertype S will have S as its "top", as might a conversion from T to another subtype of S. The "top" type of a conversion that makes no guarantees about information preservation is the top of the type hierarchy (an object type that has no attributes or methods; and whose only semantic "property" is object-ness). There are many sorts of information loss that are not well-expressed in this fashion. But specifying the "top" type is adequate in many instances when automatic conversion is required, particularly if abstract supertypes have been created to express common properties of two types.

Note that type encodings provide yet another way to relate two types. If type T is encoded as type U, there is in effect a trivial conversion from T to U. Clients not familiar with the abstract type T can treat objects of type T as type U if desired.

4.6 Type Oracles: Maintaining the Type Graph

Type oracles are the agents that allow all of this type information to be used effectively. Type information described in previous sections is represented as objects that are registered with a type oracle.

In addition, agents that can perform the computations required for methods, attributes, and conversion can register with type oracles. (Type oracles can also themselves be computation servers for some types.) Clients can then query type oracles if they need help working with a particular type, and be told about servers they can consult. Type oracles can also exchange type information among themselves as new type information comes in, either manually or by automatic propagation methods like those used in Usenet news.

Type oracles, by virtue of their knowledge of large parts of the type graph, have many uses. They can be browsed by application programmers looking for suitable types to work with. They can be used by programs that need to work with objects they can't handle by themselves. They can derive appropriate chains of conversion steps from an object of one format to an object of a desired format. When they themselves are unfamiliar with a type, they can refer to other type oracles that might have information on it.

It is important that the type graph remain consistent over time, even as new type information is added. Once a type is "published" through a type oracle, its basic interface and semantic guarantees cannot change (since clients, and agents that implement the interface, may depend on the published definition). However, new methods may be added, if they are shown to be derivable from the type's basic methods and attributes. New encodings and conversions can be added freely, and agents can be registered to compute methods, attributes, and conversions. Subtyping relations can be added if the appropriate substitutability constraints hold.

If a type's definition does need to be "changed" at some point, a new type can be created that is then related to the old type, perhaps through subtyping. Or, in cases when someone wants to experiment with a type before giving a complete interface definition, a type can be registered as "experimental". This allows it to be placed in the graph and assigned encodings, conversions, methods, and attributes, but no guarantees are made about the stability of the type definition. Use of the type in the interface definition of established "published" types is restricted accordingly.

It is always possible for incorrect or inconsistent information to be entered. However, the type oracle has registration mechanisms that allow the type's creator, and the maintainer of the type oracle, to screen new type information, and contain the undesirable effects of bad information. Full details of these mechanisms, while important scalability concerns, are beyond the scope of this paper.

A simple stream-oriented protocol called TOP has been designed to register information with type oracles, query them, and request objects and services from type oracles or other TOM-savvy agents. These agents often have some knowledge of similar protocols as well, such as HTTP. Since TOP can be used both for getting information about types and agents, and for invoking the services of agents, an agent can act both as computation server and type oracle for the data types it defines.

5 TOM: Implementation and Application

As of late February 1995, prototype type oracles have been implemented, speaking the TOP protocol and managing type description objects for about 30 types. (This number could be much larger in a heavily used type oracle.) The oracles also offer services for some of the types, including a reference type ("s:url") for fetching objects from the World Wide Web. Programs are also being built for communicating with the type oracles via Web browsers.

The first application implemented with TOM is a network-based data format converter. Using filenames or URLs, users can specify files they wish to convert, the format they want them converted to, and some optional additional information on the desired conversion. The converter program (which has no knowledge of the Web, or of the type graph, beyond a simple lookup table for format names and suffixes) contacts a nearby type oracle. The type oracle looks up the types, searches the graph for a suitable conversion path, and invokes the appropriate servers for the conversion. (In some cases, the type oracles can perform conversions on their own.) The oracle and servers can dereference URLs as needed.

This application shows how TOM can be useful even where TOM is not widely adopted explicitly. When the conversion client is installed, users get access to conversion programs they might not have known about, or might not have available on their local machines. Users do not have to know anything about TOM or TOP to use the converter, and the information they are converting does not have to have been designed with TOM or TOP in mind. At this writing, the application is only available within the CMU computer science department, but will be made available to a wider community in the future.

Other applications will be built to use the abstract interfaces of types. One possible application is to build search-and-retrieval services for computer science research documents. Abstract objects can be designed for existing archives and databases, mediated by TOP servers. Since different information repositories have somewhat different access and data models, the challenge will be to unify information from similar but heterogeneous data sources, using suitable abstract type interfaces.

TOM can also be used in specialized information applications. Application builders can publish full interfaces to their semantic data types, so that programs familiar with the system can take full advantage of the information structure. They can make the types subtypes of more general, well-known types, or provide encodings and conversions to other types, to allow other general-purpose programs to use the information as well.

6 Future Work

Much work remains to be done on TOM, in areas such as evaluating its applicability and scalability, refining the model, and in developing applications of the model.

Case studies of the proposed applications described above will test the usefulness of abstract interfaces to information types, both for information in existing systems and in systems designed with TOM in mind. The growth of the type graph, and of the population of type oracles and servers, will test the scalability of the system, and the degree to which the type graph can grow and evolve in a distributed fashion. The semantic leverage the system provides can be tested by comparing applications that use it to those that use other wide-area information systems like the Web, or more general-purpose object systems like CORBA.

Other aspects of the model that deserve more research include developing useful (and usable) ways to represent semantics of types, and finding ways to improve a type oracle's ability to find useful conversion chains and other services. The concept of registering type information could also be extended to registering other types of objects with servers, and thus expand TOM's potential application domain while still preserving the flexibility of TOM's "read-only" object system.

The software itself can be further developed. The TOP server code is currently an early research prototype, and can be further developed. An abstract programming interface to the services of TOM, above the level of the TOP protocol, would be useful. The protocol itself can be extended to support security and commerce.

7 Conclusion

The work described in this paper represents an important first step towards raising the level of semantic sophistication in networked information systems. TOM provides a way for anyone to define and publish descriptions of the abstract structure of information, and relate new structures to existing structures and formats through subtyping, encoding, and conversions. TOM thus allows information to be used in its full semantic expressiveness (either locally, or with assistance from outside agents), or in a related form that a client is prepared to handle. Type oracles allow this complex graph of types, and servers that can work with the types in the graph, to be usable throughout a global information system. The treatment of objects as values rather than as storage locations, while limiting the range of direct applications of TOM, allows a wider variety of information services to be used, and also makes it significantly easier to create substitutable subtypes, than in a general-purpose distributed object system.

8 Acknowledgements

I wish to thank my advisor, David Garlan, for his many helpful comments and advice on the ideas in this paper, and on the paper itself. Jeannette Wing, Bill Scherlis, and the various members of the Software Systems Study Group at Carnegie Mellon also helped critique and refine the concepts behind TOM.

References

[Adi94] C. Adie. Network access to multimedia information. Internet Request for Comments (RFC) 1614, May 1994.

[BF92] N. Borenstein and N. Freed. Mime (multipurpose internet mail extensions): Mechanisms for specifying and describing the format of internet message bodies for the format of arpa internet text messages. Internet Request for Comments (RFC) 1341, June 1992.

[BLCGP92] T. J. Berners-Lee, R. Calliau, J-F Groff, and B. Pollerman. World wide web: The information universe. *Electronic Networking: Research, Applications, and Policy*, 2(1):52–58, 1992.

[Gro92] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. QED Publishing Group, Wellesley, MA, omg document number 91.12.1, revision 1.1 edition, 1992.

[LW94] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):52–58, November 1994.

[Pub86] Association Of American Publishers. *Standard for Electronic Manuscript Preparation and Markup*. Association of American Publishers, Washington, DC, 1986.

[SLST93] K. Shoens, A. Luniewski, P. Schwarz, and J. Thomas. The rufus system: Information organization for semi-structured data. In *Proceedings of the 19th VLDB Conference*, Dublin, Ireland, 1993.

[SS94] Peter Schwarz and Kurt Shoens. Managing change in the rufus system. In *Proceedings of the 1994 International Conference on Data Engineering*, Houston, Texas, February 1994.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, second edition, 1991.

[Wie92] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, March 1992.