Mediating Among Diverse Data Formats

John Ockerbloom

January 1998

CMU-CS-98-102

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee

David Garlan, Chair
William L. Scherlis
Jeannette Wing
Peter Schwarz, IBM

*Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy*

This research was sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreements numbered F33615-93-1-1330, F30602-97-2-0031, and N6601-95-C-8623. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

Keywords: information system applications, mediators, distributed systems, abstract data types, data formats, conversion, object-oriented methodology

In memory of Nico Habermann

Abstract

The growth of the Internet and other global networks has made large quantities of data available in a wide variety of formats. Unfortunately, most programs are only able to interpret a small number of formats, and cannot take advantage of data in unfamiliar formats. As the Internet grows, new applications arise, and legacy data persists, the diversity of formats will continue to increase, worsening the problem. Current approaches to data diversity fail to scale up gracefully, or fail to handle the full heterogeneity of data and data sources found on the Internet.

I have developed a data model and a system of mediator agents that support the widespread use of diverse data formats much more effectively than current approaches do. In this thesis, I describe and evaluate the design and implementation of this data model, known as the Typed Object Model (or TOM), and the system of mediators that supports it. TOM is a read-only object-oriented data model that describes the abstract structure of data formats, their concrete representations, and relations between formats. TOM is supported by a distributed network of mediator agents (known as type brokers) that maintain information about data formats, and provide uniform access to conversions and other operations on those formats. Type brokers plan complex conversion strategies that can involve multiple servers, and ensure that conversions preserve information needed by clients. Data providers can also register new formats, operations, and conversions with type brokers in a decentralized manner, and make them usable anywhere on the Internet. TOM type brokers now work with hundreds of data formats, often through integration of off-the-shelf programs. TOM also supports a wide variety of applications and interfaces, such as the Web-based TOM Conversion Service, that have users worldwide.

Acknowledgements

This thesis would not have been possible without the help and encouragement of many people. I probably would not have started it without the help of Nico Habermann, to whom this thesis is dedicated. As my original adviser, he encouraged me to pursue new research directions that captured my interest, and taught me how to "think big" and pay attention to the fine details at the same time. I am grateful to him both for his guidance in finding a thesis topic, and for the inspiration he gave me in his research and in his life.

I probably would not have finished this thesis without the support of my thesis adviser, David Garlan, and my wife, Mary Mark. In guiding me through the long process of producing the thesis, Prof. Garlan gave me many useful ideas and suggestions on shaping my design concepts and plans, and he also read and critiqued countless drafts of my writing. As an adviser, committee chair, and role model, his help has been invaluable.

Mary has given me invaluable support as well. Not only did she take the daring step of marrying a graduate student in the midst of a thesis, but she helped me to keep going, and finally wrap up the thesis work, in countless ways. I am deeply grateful to her both for her patience and her encouragement.

I owe a debt of gratitude to all of my thesis committee members. Bill Scherlis posed important questions, and gave useful suggestions to me for building a firm scientific basis for my work. Peter Schwarz was the source of a number of the ideas I further developed in my thesis, and was gracious in sharing his thoughts and suggestions with me. Jeannette Wing's insistence on precise thinking and clear writing helped me immensely in sharpening my ideas and my expression. I am also grateful for her work with Liskov on subtyping, which provided a crucial part of my model, and for the opportunity to collaborate with her in further developing this work.

I am also grateful to David Garlan, Jeannette Wing, and Mary Shaw, for sponsoring the TinkerTeach project and for giving my work a central role in that project. Through TinkerTeach, we developed practical applications of my work, such as the TOM Conversion Service, and were able to test them out on a wide audience. I thank Norm Papernick for implementing the interfaces for the TOM Conversion Service, and many of the conversions used in it, and providing much documentation and support. Thanks also go to Greg Zelesnik, who played an important role in designing and managing the development of these applications.

Many others in the School of Computer Science also supported my work in a variety of ways. I am especially grateful to Raj Reddy, Robert Thibadeau, Jim Morris, Sharon Burks, Catherine Copetas, the members of the ABLE group, the Software Systems Study Group, and my officemates for their support, as well as many other people I don't have room to name here. Thank you all.

Contents

1 Introduction **1**

1.1 The unused potential of data on the Internet . . . 1

1.2 Data diversity . . . 1

1.3 Current approaches to data diversity . . . 2

1.4 Requirements to successfully handle data diversity . . . 3

1.5 TOM: A new approach . . . 5

1.6 Key contributions . . . 7

1.7 The plan of the thesis . . . 8

2 A Quick Tour of TOM **9**

2.1 Converting unknown data types . . . 9

2.2 Working with data through abstract interfaces . . . 10

2.3 Disseminating new data types and formats . . . 13

2.4 Summary . . . 15

3 Related Work **17**

3.1 The architecture of Internet information systems . . . 17

3.1.1 Component models . . . 17

3.1.2 Data models . . . 18

3.2 Examples . . . 19

3.2.1 Client-server systems . . . 19

3.2.2 Systems with mediators or expert agents . . . 21

3.2.3 Data description mechanisms (non-object-oriented) . . . 22

3.2.4 Object-oriented data description mechanisms . . . 25

4 The Typed Object Model **27**

4.1 Overview of the data model . . . 27

4.2 Objects, values, and types . . . 29

4.3 Encodings, decodings, and formats . . . 30

4.4 Getting concrete: shipped objects . . . 32

4.5 Subtypes . . . 34

4.6 Type interfaces: methods and attributes . . . 35

4.6.1 Methods: functions and specifications . . . 36

4.6.2 Class methods . . . 38

4.6.3 Attributes: functions and specifications . . . 39

4.6.4 Attributes and methods in subtypes . . . 40

4.7 Conversions . . . 42

4.8 Conversions: notes on their design and use . . . 45
4.9 Semantic specifications: notes on implementation . . . 45
4.10 Examples of TOM types . . . 46
4.10.1 Example 1: Uniform Resource Locators . . . 46
4.10.2 Example 2: Sequences . . . 48
4.10.3 Example 3: Microsoft Word documents . . . 48
4.11 Summary . . . 50

5 Type Brokers 51
5.1 Overview of TOM's component architecture . . . 51
5.2 Invoking operations on TOM objects . . . 52
5.2.1 Basic invocation: rationale and description . . . 52
5.2.2 Variations on basic invocation . . . 53
5.2.3 Server implementations . . . 55
5.2.4 Error recovery: when invocations fail . . . 55
5.3 Maintaining information about types . . . 56
5.3.1 Type description objects . . . 57
5.3.2 Reasoning with the type graph . . . 59
5.4 Registering and propagating type description objects . . . 61
5.5 TOP: The communication protocol for TOM . . . 61
5.5.1 Expressiveness . . . 62
5.5.2 Simplicity . . . 63
5.5.3 Efficiency . . . 64
5.5.4 Interoperability and gateways . . . 65
5.6 Summary . . . 65

6 Scaling Up: The Growth of the Type Graph 67
6.1 Adding new types . . . 67
6.1.1 Naming a type . . . 67
6.1.2 Coexistence of new types and old types . . . 70
6.1.3 Propagating a new type . . . 70
6.2 Augmenting existing TOM types . . . 70
6.2.1 Fixed aspects of a type . . . 72
6.2.2 Handling change in the basic definitions of types . . . 72
6.2.3 New methods and attributes . . . 73
6.2.4 New encodings and conversions . . . 75
6.2.5 New subtype relations . . . 75
6.2.6 Agent information . . . 77
6.2.7 Naming conflicts . . . 77
6.3 Method dispatch in evolving type graphs . . . 78
6.3.1 Introduction to method dispatch . . . 78
6.3.2 Problems with method dispatch in distributed object systems . . . 79
6.3.3 How method dispatch works in TOM . . . 81
6.3.4 Implementing the method search . . . 82
6.3.5 An example . . . 83
6.3.6 Summary of TOM method dispatch . . . 84
6.4 Chapter summary . . . 84

7 **TOM in Action** **85**
7.1 Conversion services . . . 85
7.1.1 The conversion problem . . . 85
7.1.2 TOM conversion applications . . . 87
7.1.3 Review and retrospective . . . 95
7.2 Frame Services . . . 96
7.2.1 Raising the common denominator for data . . . 96
7.2.2 The framing of the Web . . . 97
7.2.3 TOM types for frames . . . 98
7.2.4 Summary . . . 103

8 **Evaluation** **105**
8.1 Expressiveness . . . 105
8.1.1 Breadth . . . 105
8.1.2 Depth . . . 109
8.1.3 Limits on expressiveness . . . 113
8.2 Composability . . . 114
8.2.1 Data heterogeneity and legacy data . . . 114
8.2.2 Computational heterogeneity . . . 115
8.2.3 Computational heterogeneity: hard cases . . . 117
8.2.4 Incremental integration of TOM . . . 118
8.3 Feasibility . . . 119
8.4 Scalability . . . 122
8.4.1 Scaling up usage and services . . . 122
8.4.2 Scaling up the type graph . . . 122
8.4.3 Imperfect scaling: format identification . . . 125
8.5 TOM amidst the Internet: 1998 and beyond . . . 125

9 **Conclusion** **129**
9.1 Contributions . . . 129
9.2 Future work . . . 130
9.2.1 Extensions of TOM's design and implementation . . . 130
9.2.2 Further research involving TOM . . . 131
9.2.3 Some directions for further research . . . 132

A **Specification of TOP: The Typed Object Protocol** **135**
A.1 Status of this specification . . . 135
A.2 Summary of the protocol . . . 135
A.3 Request descriptions . . . 136
A.3.1 PROTO: Protocol negotiation . . . 136
A.3.2 NOOP: Synchronization . . . 137
A.3.3 QUIT: Terminates a session . . . 137
A.3.4 OPER: Carries out a remote object operation . . . 137
A.3.5 ATTR: Fetches an object attribute . . . 140
A.3.6 CNVT: Gets an equivalent of an object in a new type or encoding . . . 140
A.3.7 TYPQ: Gets information about a type . . . 140
A.3.8 TYPL: Lists types that have been registered or changed recently . . . 141
A.3.9 REGI: Registers information about a type . . . 141

A.3.10 AUTH: Gives authorization 143
A.4 An example session 144
A.5 Summary of reply codes 145

Chapter 1

Introduction

1.1 The unused potential of data on the Internet

The tremendous growth of the Internet and its World-Wide Web has allowed millions of people to provide and view vast quantities of data. Much of this data, however, cannot be used effectively by a large audience. While users may be able to retrieve data, they may not be able to interpret it, or even display it intelligibly.

Structural mismatch is largely to blame for this problem. Far too often, the form or structure of data does not match the needs of a user, and cannot easily be adapted into a usable form. The format, or representation, of the data may be unintelligible not only to the user, but worse, to the user's programs. For example, documents compressed in an unusual manner may be uninterpretable. The continual introduction of new formats exacerbates this problem. For instance, Word 6.0 data files are unreadable by Word 5.0, even though Word 6.0 and Word 5.0 files both represent word processing documents produced by two different versions of the same product. Even if the data itself is intelligible, it may lack the structure that enables programs to use it effectively. For example, a numeric table in ASCII form, readable by a word-processing program, might not be interpretable by spreadsheet programs that could analyze the table.

Structural mismatch is likely to occur in any large-scale information system that lacks centralized control, since different users, acting independently, will design and use different data formats. Mismatch is especially problematic on the Internet, since the Internet has a large and diverse population of users, and includes data from many thousands of independent sources. While mismatch represents a substantial barrier to effective use of Internet information, it can be overcome with an appropriate conceptual framework and set of tools.

1.2 Data diversity

Why does data come in so many different forms? Part of the reason is that data is used to describe many different kinds of things, which may call for the data to be conceptually structured in different ways. We use the term *data types* to denote conceptual structures and interpretations of data. These structures may be quite simple. For example, *sequence of bytes* is a type that can be used to describe data streams, such as those transmitted over the Internet. Types may also be more complex. For example, *searchable catalog of bibliographic entries* is a type that can be used to describe the Library of Congress on-line card catalog. The structure described by a data type can be explicit (e.g., the relations in a database), or implicit (e.g., the time between points in a bus schedule, which is not explicitly stated, but deducible from times given for adjacent points).

Data types are also represented in a variety of ways. We use the term *data formats* to denote representations of data types as sequences of bytes, the underlying representation of all data transmitted over the Internet. Some data types are represented by multiple data formats. For example, bibliographic citations in journal X may use different punctuation conventions from the same bibliographic citations in journal Y, because the two journals prescribe different citation styles. The citations in both journals convey the same information (so they have the same type), but represent it slightly differently (and thus have different formats). Similarly, on the Internet multiple formats often represent the same data type; for example, there are multiple ways of representing pixel maps. Formats are often optimized for particular needs, such as compact representation (e.g., compressed formats), easy parsing (e.g., SGML formats), or preventing errors in transmission (e.g., formats that use limited character sets and checksums).

The ability to interpret the data types and formats available on the Internet is highly useful, especially when dealing with the large volume of data available on the Internet. Programs that can interpret data types can extract, derive, compile, and condense useful knowledge from a larger set of raw data. They can search for relevant information in a large data set, and intelligently filter out irrelevant information.

Unfortunately, a typical client, whether a user or a program, can work with only a small fraction of the many formats used on the Internet. A client cannot take advantage of useful data in incompatible formats. In some cases, a user may able to find a program which will convert data into a format that the user's programs understand, or a user may find a new program that can interpret the data in its native format. But even if such conversion and application programs exist somewhere, it is often extremely difficult and inconvenient to learn of the existence of an appropriate program, install it, and use it.

1.3 Current approaches to data diversity

How do current information systems handle data format diversity? There are five basic approaches in wide use today:

Specialized niches. Applications can make up rich formats for their own use, and may also support conversions to a few common formats. Many proprietary programs work this way. This approach allows for very expressive types, but does not support heterogeneity. The specialized formats can be used only by applications specially written for them.

Lowest common denominator. Applications that want to share data widely may use commonly-used simple formats, like ASCII text, or HTML. While this "lowest common denominator" approach allows a large number of programs to use the data, expressiveness suffers. Many useful data types, such those found in databases, or types like maps that use spatial structures, cannot be represented in these common formats without unacceptable loss of information, or of mechanically interpretable structure.

Simple data typing schemes. There are a few commonly-used conventions for identifying data formats. DOS, Windows, and Unix programs often use file-name suffixes for this purpose. Since the suffixes are assigned in an informal and ad-hoc manner, their meanings are not well-defined, and the same suffix may mean different things in different places. Data typing schemes involving central registries (such as Apple's creator and file type registry) are better at describing a set of formats, since they allow a definitive definition of a format. These and other registry mechanisms, like MIME, are still not scalable, due to the dependence on a central registry. Moreover, the type registries in use today are not expressive enough. However, even if all formats are clearly and unambiguously labeled, clients may still be unable to do anything with unfamiliar formats.

New standards. New formats can sometimes come into widespread use after a standard has been defined and publicized by a respected standards body. However, the standards process alone cannot deal with the wide heterogeneity and scale of Internet use, where applications are written for many different needs, and new data types (and new versions of those data types) are introduced much faster than any standards body has been able to handle them.

Polyglot client applications. It is sometimes suggested, explicitly or implicitly, that client programs will eventually be able to interpret all relevant formats. While some large programs do interpret many relevant formats, this solution still does not scale. The number of potentially useful data formats is large, and continues to grow. Programs would need constant upgrading (and enlarging) to handle them all. A single static client program may have expertise in a few relevant formats, but some other mechanism is needed for handling and translating all possible relevant formats.

All of these solutions, in short, are unable to handle the wide variety of data types that will always be present on the global Internet. To make this universe of types manageable, new solutions are required.

1.4 Requirements to successfully handle data diversity

If clients need to be able to interpret an arbitrarily large number of data formats on the Internet, they will need the support of a system that can interpret these formats, rather than relying on fixed standards or programs that rapidly become obsolete. Because there are so many data formats on the Internet, defined by so many different groups and used by so many different people, this system needs to be *distributed*. A distributed system can take advantage of information and resources throughout the Internet, and work with the Internet's varied data formats in a scalable manner.

This distributed system needs to fulfill four basic requirements. The first is *feasibility*, or the ability to carry out its tasks correctly, robustly, and efficiently. The second is *composability*, or the ability to work with a wide range of past, present, and future data formats, as well as with a wide range of programs that provide and use this data. The third is *expressiveness*, or the ability to describe in detail the function and representation of unfamiliar data types, and the services available to work with them. The fourth is *scalability*, or the ability to handle a growing number of data formats defined by people and groups from all over the Internet.

We can elaborate on each of these requirements as follows:

Feasibility. These are basic requirements for any system to be usable in practice:

- The system must be *reliable* and *robust*. Clients requests should be answered correctly whenever possible. If errors occur, their effect should be minimized and localized, and it should be possible to recover from errors when detected.
- The system needs to have *acceptable performance*, so that it is usable for everyday, practical tasks. Specifically, clients should be able to get responses to requests in reasonable time, without the system imposing excessive overhead. The system should also conserve bandwidth, avoiding unnecessary transmission of large quantities of data.

- The system's behavior must be *predictable*, *consistent*, and *modular*. Clients should be able to predict the effects of their requests without needing to know which server is carrying them out, and without needing to know about other parts of the system unrelated to their request.

Composability. A widely distributed information system must be able to interoperate with a wide variety of systems and data structures, including existing or "legacy" data and programs. It should also be easy to create new programs and interfaces to the system that take advantage of its features. Some of the implications of this requirement include:

- *Support for data heterogeneity.* The system should be able to interpret data from a wide variety of sources. In particular, existing data sources on the Internet, such as the World Wide Web, should be usable through this system.
- *Support for computational heterogeneity.* Programs that manipulate data use a wide variety of programming languages, operating systems, and user interfaces. The system should support clients and servers with all of these platforms and user models, without requiring the adoption of a particular language (as does Java) or operating system (as do current implementations of OLE).

  The system should also support a wide variety of execution models. Programs interpret and manipulate data in several different ways. They can operate directly on the bytes that represent the data, as C or Perl scripts tend to do. They can operate on the data via an abstract, object-oriented interface, without directly manipulating the low-level representation. Or, they can ask a server to operate on the data, following a client-server computation model. Each of these models best suits certain information-processing applications and not others. Therefore, the system should allow all of these execution models, rather than limiting its applicability to only one of them. The system should also be composable with a wide variety of user interfaces.
- *Support for incremental integration.* The success and usability of the system should not depend on its mass adoption, or on its superseding existing practices. Rather, users should be able to reap benefits from the system *without* having to make significant changes to their overall information management practices. The system should therefore specify only a minimal set of rules for a data provider to follow. This principle lets the system work with existing data repositories, without having to exert excessive control over the behavior of data providers.

Rather than requiring that clients or providers of data manage data in a particular way, or through a particular program, composable systems should *accommodate* a wide variety of data formats and management practices. Given sufficient information about how a provider's data is managed and organized, it should be possible to integrate the provider's data into the system. Similarly, clients should be able to receive data in the form they find most convenient, without having to adopt a particular "standard" data format or use one of a small set of "standard" programs.

Expressiveness. With respect to data format descriptions, expressiveness has two basic dimensions: breadth and depth. These can be defined as follows:

- The description mechanisms must be broad enough to *describe a very wide range of data formats* that are now in use, or that may be designed in the future. For instance, it should be possible to describe all formats that are defined in the popular MIME data description scheme, as well as other commonly used formats and interfaces. I refer to this requirement as a "breadth" requirement. (This requirement is similar to the "data heterogeneity" requirement described above, but focuses specifically on the descriptive power of the system.)

- It should be possible for programs and programmers to *get detailed descriptions of data* to support sophisticated interactions with data in various formats, without having to know in advance the details of the data's internal representation. I refer to the requirement that data descriptions should be rich and detailed as a *depth* requirement. For example, clients should be able to discover that a "URL" is a kind of reference to an object. Clients should be able to ask for the object to which the URL refers without having to know how to parse URLs. Clients should also be able to find out that the structure of the URL contains specific information such as the protocol and servers that can be used to fetch an object.

Scalability. The system must be able to handle a growing number of data types and formats, and continue to function robustly as more clients, servers, and data types are introduced. In particular:

- The *scale of the network should help the growth of resources* and information in the system, rather than hinder it. For instance, users should be able to take advantage of the large number of servers and providers that can provide data and operate on it. At the same time, the growing number of types and formats the system handles should not produce force a similar growth in the resources required to execute a typical operation.
- The system should be able to *grow robustly and consistently.* New types, new information about types, and new services should be easy to add to the system, and not make it slower or more brittle. In particular, the effects of a well-defined operation should remain consistent even as new capabilities are added to the system. (This is not always the case in some object-oriented systems, where the introduction of new supertypes can affect the semantics of subtypes in unexpected ways.) These issues are particularly important in distributed systems, where growth of the server network and the network of types is not centrally controlled. For instance, if one person registers some information about a new type, it should not invalidate other information registered by someone else. Or, if someone provides erroneous information about types or services, the consequences of this error should be minimized. The error should not affect other types and services that do not logically depend on it.
- The system *should not depend on centralized entities* that could cause bottlenecks. No single computing component should have to be a sole mediator for dealing with a particular data format. Similarly, data providers that wish to define new data types or services should not have to wait for a centralized body to act before they can offer their definitions and services to the Internet.

1.5 TOM: A new approach

My thesis represents a new approach that allows users to cope with the ever-growing set of data types and formats used on the Internet, and satisfies the requirements listed above. The approach has two parts:

1. An object-oriented model (the Typed Object Model, or TOM) that describes the abstract structure of data formats, their concrete representations, and relations between types.

   The model is expressive enough to encompass formats already in use, as well as new types of complex objects. TOM's data model is simplified in some respects from standard object-oriented models. Most notably, TOM treats objects as immutable values, rather than as mutable variables (though it can still cope with changes in the environment caused by others). This principle allows TOM to work with a wider range of servers, delegate computation more flexibly, and take advantage of more options to relate different types and formats than the standard object-oriented model allows.

2. A distributed network of mediator agents (known as *type brokers*) that collect and disseminate type and format information, and give clients access to servers that convert and operate on data in various formats.

   By an "agent", I mean a program that communicates with other programs.[1] Agents in TOM include clients, servers, and type brokers (the mediators). Type brokers allow users and applications to look up type information, add new types, use data in unknown formats, and convert data to known formats (while controlling information loss). They also allow information providers to define new types and formats, and register new services that operate on these formats. The brokers act as go-betweens connecting client requests with appropriate services.[2]

The central claim of this thesis is as follows:

> It is feasible to design and implement a data model and a type broker network that allow individual data providers to define complex data in a wide variety of formats, and make these data formats widely usable throughout the Internet. A system that uses this model and this network is superior in dealing with diverse data formats, as compared to previous widely-deployed systems, in terms of its expressiveness, its support for composability, and its scalability.

What do I mean by superiority in these areas? Specifically, I claim:

Expressiveness: TOM allows an unlimited number of data formats to be defined. The definitions include information about both the abstract interface and the concrete representation of the formats. The definitions can be retrieved from anywhere on the Internet. Semantics of the types and formats can be tightly or loosely specified, new types can be related to old types in well-defined ways, and services like conversion and data interpretation can be automatically invoked. All existing static document formats on the Internet, as well as newly defined types, can be described using TOM's object model.

Composability: TOM allows clients to use a wide range of heterogeneous data structures and formats, including formats the client has never seen before. It also can be used with a wide range of existing data servers, including World Wide Web servers, and a wide range of clients. TOM also supports incremental integration.

[1] Although some writers use the term "agent" to imply artificial intelligence capabilities as well, I do not.

[2] For simplicity, I will use the term TOM to describe both the data model and the network of software agents described above. Both the model and the actual agents are key contributions of this thesis. It will usually be clear from context whether TOM is being used to refer to a model or to computer programs, but I will clarify meaning when necessary.

Scalability: TOM's distributed type broker network allows the universe of known types and formats to grow to a much larger size than can be handled by any single entity. New type information does not hide or invalidate old type information. Rather, clients can use old types as an aid in handling new types, through relations like subtyping, conversion, and encoding.

Feasibility: Thousands of people worldwide have used TOM since its implementation. In this thesis, I show that TOM-based applications are acceptably robust and efficient in comparison with related systems in use today.

These are the principal aspects of TOM that allow it to satisfy the claims above:

The power of distributed expertise. A distributed system can potentially harness millions of servers to work with various data formats and types. TOM makes it possible for someone to define a new data type and its interface, implement the interface on behalf of outside users, and convert its formats to other formats. Users all over the Internet can use all of the types and services provided in this manner.

The flexibility of mediation. TOM's type brokers are mediators that bridge data format mismatch between data servers and clients. They work with existing servers and clients as well as those written specifically for TOM. They provide a common protocol that clients, servers, and brokers can use to interact. Type brokers keep track of data types, formats, and services, allowing appropriate services to be located. They also share this information with other mediators, making the information widely available.

The power of an expressive model. TOM's data model gives both an abstract, object-oriented view, and a concrete, byte-oriented view of data. In contrast to the opaque "black box" view of data provided by many systems that use data abstraction, TOM supports the simultaneous description of data at multiple layers of abstraction. This approach allows clients to manage data at a high level or a low level as appropriate.

A careful design and implementation. TOM's type brokers and protocols have been carefully designed, tested, and refined in applications used every day. Attention to detail – both in terms of technical considerations and in terms of the people using the system – is crucial in building a system that deals with diverse data structures in a scalable manner. Many of the design concepts TOM uses are not original to TOM. The novelty of TOM lies primarily in the way that familiar concepts are integrated in a working system that enables data formats to be widely used in a rich and scalable fashion.

In this dissertation, I demonstrate my thesis claims as follows: First, I analyze the design and use of existing Internet information systems. I then describe the design and implementation of TOM, analyze the design, describe how TOM has worked in practice, and compare it with other existing systems.

1.6 Key contributions

The main contributions of the thesis are:

- For information systems researchers: a better understanding of how mediators can allow abstract data types to be widely used in loosely organized distributed environments, such as the Internet. Wiederhold predicted in 1992 [Wie92] that mediators would become an increasingly important part of future Internet information systems, but mediators have not yet taken the central role that Wiederhold predicted. In this thesis, I show how type brokers can mediate to interpret data formats. My analysis of the design of type brokers, and my comparison of TOM with existing systems, show the important contributions mediators can make as part of the architecture of distributed information systems.

- For object-oriented systems researchers: an extension to the standard object-oriented model that explicitly considers representations of objects as well as their abstract interfaces. I show that objects can be defined in a disciplined manner both abstractly and representationally, in a way that preserves many of the benefits of "black-box" objects while allowing representations to be converted to fit the needs of application programs. I show how a new concept called "intersubstitutability" can be used to specify information preserved in conversions.

- For designers and programmers of information systems: a working prototype of a type broker, and an analysis of the strengths and weaknesses of its design and construction. I also describe two case studies in this thesis. One of them shows TOM's usefulness in converting documents found on the Internet. The other shows how TOM introduces new types to the Internet more cleanly than current practice does.

- For information providers: a graph of useful data types and services for common data formats. In the process of implementing TOM, I have built the beginnings of a type graph, and other users have enlarged it further. The repertoire of data types and encodings is useful both to people using TOM directly, and to people wanting to incorporate new data formats into their own systems. The full type database can be obtained by querying type brokers at Carnegie Mellon.

1.7 The plan of the thesis

In Chapter 2, I give a brief guided tour of TOM, to illustrate the capabilities of the system and to provide a basis for detailed discussion of TOM.

In Chapter 3, I discuss related information systems, focusing on their architecture and their models of data. I introduce general concepts that are useful in analyzing TOM. I also provide a baseline of systems with which TOM can be compared.

The next two chapters detail the basic design of TOM. Chapter 4 contains a detailed description of the Typed Object Model, including formal definitions. Chapter 5 describes type brokers, focusing on the information they maintain and their mediation between clients and servers, and discussing how they use their type knowledge to plan complex conversions.

The remaining chapters are devoted to analyzing the overall design and demonstrating the claims of the thesis. Chapter 6 discusses issues related to the growth and scalability of the system, including detailed discussion on type evolution and method dispatch. Chapter 7 details two case studies of the system. Chapter 8 summarizes my analysis of the system, and shows how it justifies my thesis claims. Finally, Chapter 9 sums up the contributions and conclusions of the thesis work, and suggests further work and extensions.

Chapter 2

A Quick Tour of TOM

In this brief chapter, I illustrate from a user's point of view what TOM can do to make a wide range of data formats usable. Specifically, I show specific examples of TOM in action. I also highlight the basic features of TOM that are used in these examples. Detailed description and analysis of these features will come in later chapters.

2.1 Converting unknown data types

People who frequently look for data on the Internet often find it in an inconvenient format. A user's application programs might not support the image, sound, database, or document format used for a piece of data of interest to the user.

TOM can be used to convert data from one format to another, without installing any special programs for the unfamiliar format. Suppose, for instance, that one retrieves a file about owls in Adobe Acrobat (PDF) format from the Web, and wishes to view it or print it out. Suppose that an Acrobat viewer is not available (or easily installable) on one's own machine, but converting the document to another format (such as Postscript) would make it viewable. A simple TOM program called `netconvert` does the job. On a Unix machine, one could convert and print out such a PDF file with these commands:

```
% netconvert -t postscript owl.pdf > owl.ps
% lpr owl.ps
```

To convert files in other formats, such as audio or graphical formats, one calls `netconvert` in exactly the same way, with the desired destination format used in place of `postscript`. (The source file's original format can be identified either through the file suffix or through an explicit argument supplied to netconvert.) The netconvert program supports an unlimited number of conversions, since it can use any of the conversions supported in TOM's type broker network.

Netconvert is implemented very differently from other conversion packages like the PBM tools package or Debabelizer. Those packages have built-in knowledge of a certain number of formats, and include code to translate from one format to another, possibly through intermediate formats. If these packages do not know about a particular format, they can do nothing with it unless one adds a module to the conversion package (if this is even possible), or updates the package as a whole.

In contrast, because it uses TOM, `netconvert` is a very small program, with no knowledge of the details of particular formats. It does its work by communicating with a *type broker* on the

Internet. The type broker has information about different types and formats, as well as conversions available on various servers. The type broker itself does not have to know how to perform particular conversions; it simply needs to know where it can find other servers that can perform them. As the type brokers learn of more formats and more servers that can handle conversions, the number of available conversions can grow without bound, especially since type brokers can combine multiple conversion steps to produce a new "composite" conversion.

TOM's type brokers allow a client with finite resources to take advantage of the ever-growing resources of the Internet to handle different forms of data.

2.2 Working with data through abstract interfaces

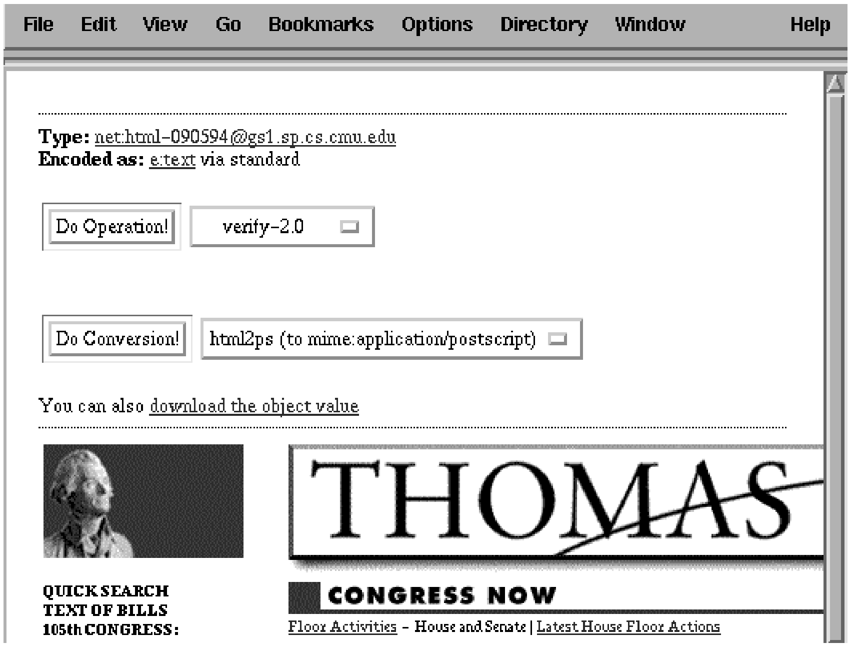
With some data, conversion is not a major issue; effectively using the data is the main issue. For example, if presented with a large, multi-gigabyte collection of information, one may wish to search it for particular relevant parts, rather than retrieve the entire collection. Alternatively, a client may need to get information from a document that has a format that cannot be converted into any form the client can parse without unacceptable information loss.

TOM allows providers to define a data type with an object-oriented interface, including attributes and methods. A user or a program calls a method of this interface by sending a request to a type broker. The type broker then finds an agent implementing the method, invokes the method by sending that server the data needed to execute the method, and finally returns the results back to the client. Using the brokers this way, clients can use and analyze information from the Internet that they cannot interpret themselves.

Figure 2.1 shows a document from the Web displayed by the TOM Object Browser, a TOM-aware program that displays objects from the Web and the interfaces of their types. The TOM Object browser displays the object (in this case, an HTML file from a U.S. Congress Web server named THOMAS) below an area called an "interface bar" that identifies what type of object is being viewed, how it is encoded, and what can be done with the object. In this case, the type of the document being viewed is HTML (shown here with its full TOM type name, `net:html-090594@gs1.sp.cs.cmu.edu`). That type is represented (or "encoded") as plain ASCII text (`e:text`). Below this information about the type and its representation, the TOM Object Browser shows attributes of the type, if any (for this type, there are none), methods of the type (called "operations" in the figure), and available conversions. Below the interface bar, the TOM Object Browser displays the object itself, if it is displayable in a Web browser. The user can use the interface bar to fetch attributes, call methods, or conversions on the object. (If the object is an HTML document, as this one is, the user can also click on the links in the HTML document to view the linked objects through the TOM Object Browser as well.)

One useful method to call on HTML documents is to verify whether they conform to a particular HTML standard, like HTML 2.0 or 3.2. Documents that conform to these standards will be displayed in a uniform fashion by nearly all HTML browsers, and are not dependent on a particular Web browser's behavior. The `verify-2.0` method carries out this useful and nontrivial evaluation for the HTML 2.0 standard, and returns an HTML document that either reports success or that reports any "errors" where the HTML document diverges from the standard. In Figure 2.2, the user has called the `verify-2.0` method on the object in Figure 2.1, and the method has returned an HTML document reporting "errors". The TOM Object Browser contacted a type broker with the method request and the object to verify, and the broker then passed it along to a server, which then ran a program supplied by a third party to do the verification.

Besides verification, one can define other methods on an HTML document, such as methods to



Type: net:html-090594@gs1.sp.cs.cmu.edu
Encoded as: e:text via standard

Do Operation! verify-2.0

Do Conversion! html2ps (to mime:application/postscript)

You can also download the object value

Figure 2.1: An HTML document viewed through a TOM proxy. The user can browse the links normally, or use the interface bar at the top to invoke methods and conversions on the HTML document.

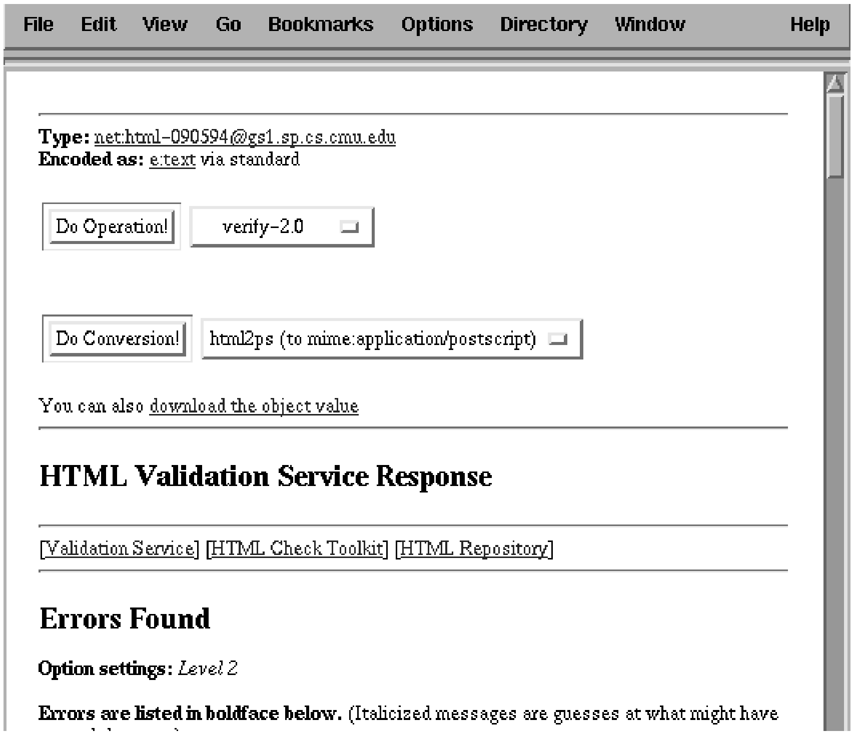
follow a particular link, or to determine which links no longer point to valid Web addresses. There are also conversions available from HTML formats to various other formats, like Postscript or plain ASCII text.

Note that the interface to HTML was added to an existing data format. We did not have to redefine HTML in any way to make it work within TOM. It is sufficient to describe what HTML as it exists can do, and what encodings exist for HTML.

The TOM Object Browser is just one user interface giving access to the type interface of an object. Users and programs can also call methods, fetch attributes, and carry out conversions on objects through other interfaces (such as the `netconvert` program described earlier) or by connecting with a type broker and issuing commands in TOM's native protocol.

Through the use of *subtyping*, another object-oriented concept, a client can work with new, unfamiliar types of data through well-known interfaces of more familiar types. For example, many different data types (MIME multipart files, multipage Postscript documents, etc.) consist of a sequence of objects. These can all be made subtypes of a generic *sequence* type. That generic type has an interface that includes an attribute for the number of elements in a sequence, and a method for retrieving the *n*th item from the sequence.) A client can work with objects in any subtype of *sequence* through the interface of the generic sequence type.

There are other examples of common supertypes: *references* that can be resolved to obtain an object; *images* that have width, height, and color values defined for every coordinate within their extent; *audio files* that have a certain playback time and a frequency and amplitude value for any point in the playback time; *catalogs* that are given a search term and yield descriptions and references to "hits". All of these supertypes are reflected in various image, audio, and index formats; through TOM, a client can use these formats through their basic interfaces without having to know the details of the formats.



Type: net:html-090594@gs1.sp.cs.cmu.edu
Encoded as: e:text via standard

You can also download the object value

HTML Validation Service Response

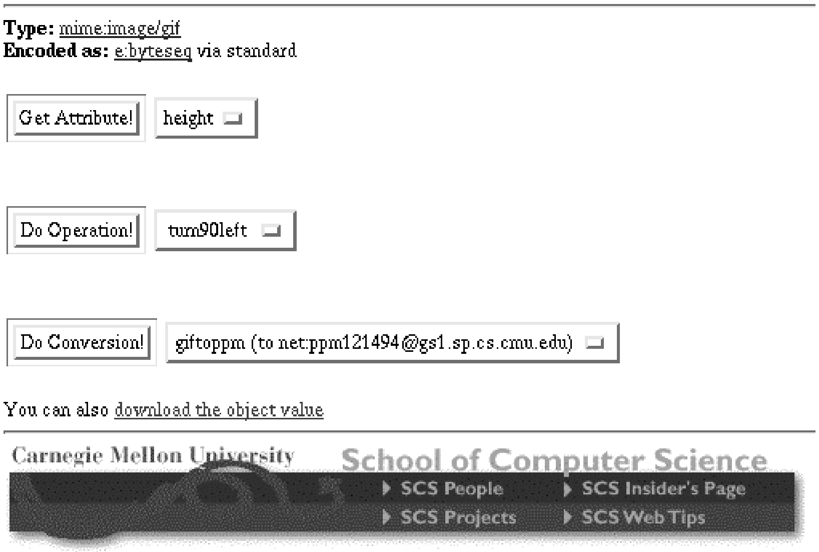
[Validation Service] [HTML Check Toolkit] [HTML Repository]

Errors Found

Option settings: *Level 2*

Errors are listed in boldface below. (Italicized messages are guesses at what might have

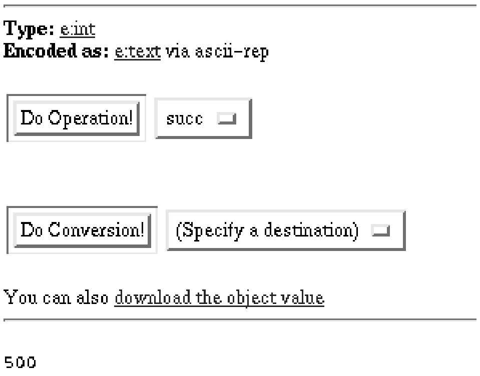
Figure 2.2: The results of the `verify-2.0` method when invoked on the HTML document in Figure 2.1.



Type: mime:image/gif
Encoded as: e:byteseq via standard

You can also download the object value

Figure 2.3: A GIF image viewed through a TOM proxy. The user can select the `width` attribute to find the image width in pixels.



Type: e:int
Encoded as: e:text via ascii-rep

You can also download the object value

500

Figure 2.4: The user has fetched the attribute of the image, and is told that the width is 500. The return value is itself an object of integer type (named `e:int` by TOM).

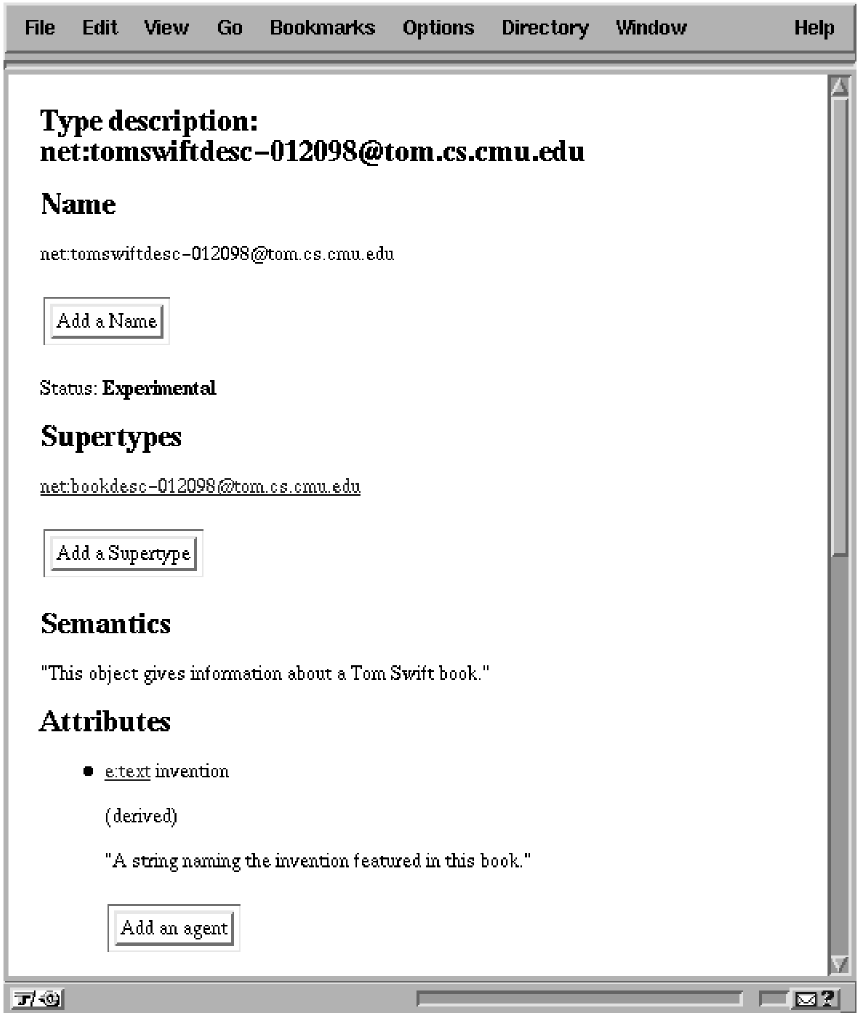
In Figure 2.3, for example, the TOM Object Browser displays an object of type *GIF image*, and allows the user to fetch attributes like `width`, or call methods like `turn90left`, on the image. If the user requests the `width` attribute, the TOM Object Browser sends this request to the server, yielding the result shown in Figure 2.4. Neither the user nor the TOM Object Browser needs to know anything about how GIF formats represent width; they only need to know how to request the attribute.

2.3 Disseminating new data types and formats

TOM provides useful services for *providers* of data as well as for *users* of data. Providers of highly structured data often find it useful to structure the data in a specialized format that domain-specific applications can easily process. However, this format may be useless to other users who are interested in the data but do not use those specific applications. For example, TIGER map data formats, HDF scientific data formats, and MARC bibliographic data formats are all highly detailed and readable by specialized applications. General-purpose applications, however, may be unable to parse these formats, even though a user of these applications may want to look at data in these formats. Hence, providers typically have to either use a specialized format that is only usable by a limited audience and application set, or use a lowest-common-denominator format that is widely accessible, but not well suited to specialized analysis.

TOM allows data providers to use complex data formats, while still making them accessible to a wide audience. Using TOM, providers can register information about new data types and formats with type brokers. Clients can use the data types abstractly through an object-oriented interface, as illustrated in the previous section, or they can convert the data to a form they can use more easily, as illustrated in section 2.1. In neither case do users have to import or install specialized application programs.

Once providers register data types, interfaces, and conversions with type brokers, anyone on the Internet can write a program to implement an operation on the type (such as an attribute fetch, a method, or a conversion), and register that implementation with a type broker. TOM servers provide a simple interface for incorporating new implementations, including off-the-shelf programs. In Unix, for instance, to make a conversion invokable by a TOM server, one simply wraps it so that the input comes from standard input and the output goes to standard output (if the program does



Type description: net:tomswiftdesc-012098@tom.cs.cmu.edu

Name

net:tomswiftdesc-012098@tom.cs.cmu.edu

Add a Name

Status: **Experimental**

Supertypes

net:bookdesc-012098@tom.cs.cmu.edu

Add a Supertype

Semantics

"This object gives information about a Tom Swift book."

Attributes

- e:text invention

  (derived)

  "A string naming the invention featured in this book."

  Add an agent

Figure 2.5: Registering a type via a Web form

not already work this way), and then adds a few lines to the server's configuration file.

Registration of new type information is also easy. Figure 2.5 shows a user registering information about a new type using a World Wide Web form. (In the figure, the user is registering a new type for information about Tom Swift books, as a subtype of a generic book-information type.) Once the user creates the type, and adds information about any necessary attributes, methods, supertypes, and encodings, the Web program sends the type information to a type broker. Type brokers can then share their registrations with other type brokers, so that users anywhere on the Internet can learn about and use new data types.

TOM's type brokers, then, allow new data types and formats to be added in a decentralized manner and be used right away, without having to wait for standards bodies to act or for a large user population to adopt a new data type. Just as the use of distributed naming and routing services has allowed the Internet to grow without bounds or bottlenecks, TOM's use of a distributed broker network allows the number of widely-usable formats and resources to grow in a similar fashion.

2.4 Summary

In this chapter, we have seen how TOM allows clients to use unfamiliar data formats through conversion and abstract interfaces. We have seen how TOM allows information providers to publish their data in specialized formats while still making them usable by a wider audience. We have seen how the system of type brokers and registrations allows TOM's users to take advantage of type knowledge and services all over the Internet. We have also seen that TOM can be used through a variety of interfaces, either directly from a program, such as simple command-line applications like `netconvert`, or through interactive user interfaces accessible via the World Wide Web. Finally, we have seen that TOM scales up to handle an unbounded set of data types and formats, including existing formats like HTML, and that TOM works with existing data repositories such as those maintained on Web servers. In the chapters to come, I will show how TOM's design make all these benefits possible.

Chapter 3

Related Work

Many systems have been designed to give access to data distributed across the Internet. In this chapter, I describe systems other than TOM that are designed to access and analyze a wide range of distributed data. I pay special attention to systems that are now used by a significant number of Internet users. This survey will identify design concepts that TOM has borrowed from related systems, and will also show where these systems fall short of solving the general problems that TOM is designed to address.

3.1 The architecture of Internet information systems

I begin with an overview of the architectural principles used by information systems on the Internet, in order to understand the environment in which TOM operates. There are two important design dimensions for such systems. The *component model* of systems explains how the computational parts of the systems are structured, connected, and arranged. The *data model* explains how the data in the system is structured. In this section, I introduce basic concepts used in the component model and the data model. This section will form a basis for the analysis of particular information systems, and their relation to TOM, in the sections that follow.

3.1.1 Component models

The field of software architecture has developed well-defined models for describing the computational structure of systems. The standard way of describing them, as seen in Shaw and Garlan [SG96] and elsewhere, is as a collection of *components* and *connectors*. Components represent loci of computation; connectors represent communication pathways between components. Informally, a software system architecture is often described as a box-and-line diagram, with boxes representing components, and lines representing connectors. Many collections of software systems share similar component and connector vocabularies, or exhibit characteristic connection patterns. Allen, Garlan, and others define the concept of *architectural style* to describe families of similar architectures (cf. [AAG93], [All97]).

Many distributed information systems can be described in a *layered* architectural style. A common layering is given in figure 3.1. The *repository* layer can be thought of as a passive data store. The *server* layer provides data to *clients*, possibly taking the data from one or more repositories. Between data servers and end-user clients, *mediators* can provide assistance such as caching, protocol translation, brokering, and other services. In many cases, mediators look just like ordinary clients to data servers, or like ordinary data servers to user-level clients. Hence, we see that the

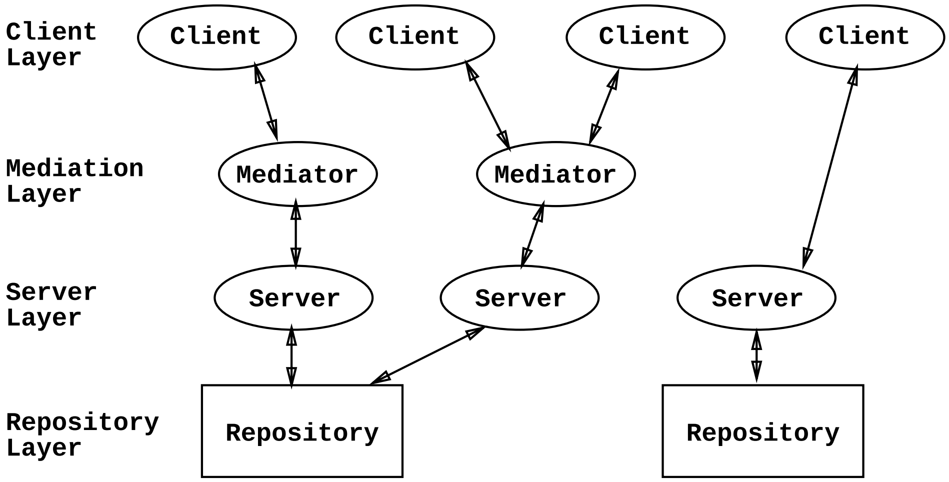


Figure 3.1: A common layering of components in information systems

same program may play a *client role* to some applications, and a *server role* to others. There may be multiple levels of mediators, or no mediators at all. Most of TOM's services live in the mediator layer.

Internet-based information systems have *dynamic* architectures. In many conventional software systems, a fixed number of component instances talk to each other using a fixed configuration of connections. In contrast, Internet information applications typically include a large, fluctuating number of components operating in parallel. Connections are made and broken dynamically between components that speak compatible protocols, or that have shared access privileges.

Also in contrast to many other applications, most Internet information applications are highly *decentralized* with respect to their components. Many Internet applications, such as the Web and Usenet news, depend on no central repository of data, and can still function if an arbitrary set of components is removed. Decentralization, in fact, is a key design principle of the Internet itself. The Internet's use of packet switching and dynamic routing allows messages to be communicated without problems even if large portions of the Internet become unreachable.

The World Wide Web is a well-known example illustrating the concepts described above. The Web includes HTTP servers at the server layer, which generally serve files stored in filesystems or databases (the repository layer). The files are viewed through Web browsers like Netscape or Lynx, at the client layer. The Web's components are decentralized, and any number of clients and servers can participate in the Web. Connections are dynamic, and clients can establish a connection with any Web server (except for those behind firewalls or that otherwise limit their access). However, some clients interact with the Web's HTTP servers indirectly, via a *proxy server*. Proxy servers act as go-betweens between Web browsers and HTTP servers, and are therefore in the mediator layer.

3.1.2 Data models

In order to fully understand an information system, one must study the design of its *data model* as well as its component model. In the simplest case, an Internet information system might treat all data as uninterpreted sequences of bytes. Simple Internet applications, like Telnet, work at this level. Usually, however, information systems include mechanisms for distinguishing different kinds of data from each other, either semantically, by distinguishing between different functions for data types, or syntactically, by distinguishing between different ways of representing data, or both.

Some systems may predefine the kinds of data they will handle. For example, DNS is defined to work specifically with “resource records” [Moc87]. Other systems may provide ways to define and describe new types, such as MIME with its extendable type registry [BF92]. Some information systems use well-known abstractions for defining data types, such as the the *database* (used by Z39.50 [Nat95] and SQL) and the *object* (used by CORBA, Java, and TOM.)

In many systems, there is a sharp distinction between ordinary data that clients retrieve from servers, and *metadata*, which the system itself uses to manage, refer to, and describe the ordinary data. Examples of metadata in common Internet information systems include URLs in the World Wide Web, headers in mail messages and Usenet articles, and indices in WAIS databases. Metadata is typically highly structured and precisely defined in information systems, since it is often automatically processed by information clients and servers. URLs in the Web, for instance, encode instructions to a Web client about what server it should contact, what protocol it should use, and what to send to the server, in order to fetch a particular object.

In contrast, information systems may say less about structure of the ordinary data that passes between clients and servers, because the structure of that data, unlike the structure of the metadata, is not essential to the system's basic operation. In place of rigid definitions, there may be an informal consensus about what kinds of data belong in a particular information system. For instance, in the World Wide Web, HTML, plain text, and GIF images are generally considered to be “standard” formats understood by all browsers, though GIFs might not be displayable in non-graphical browsers. Other kinds of formats, such as Adobe Acrobat (PDF) format and Portable Network Graphics (PNG), may also be transmitted over the Web. However, these formats are not as widely used, since many Web users cannot view them. Common conventions, then, tend to limit the formats used for most Web-based data.

As the next chapter will show, TOM in many ways treats metadata no differently from ordinary data. In TOM, both kinds of data can be highly structured, and in most cases can take on a variety of forms. For example, references to objects are not constrained to be in one particular form, like the URLs of the Web, but can be expressed in an unbounded number of formats. However, TOM's protocols do use fixed formats for a few kinds of commonly-used metadata, to improve efficiency.

In summary, the components of Internet information systems tend to be layered (often with mediators), dynamic, and decentralized. Information systems also have data models that distinguish data formats syntactically, semantically, or both, and that often distinguish between ordinary data and metadata.

3.2 Examples

Having covered some of the basic design principles of Internet information systems, I now describe the designs of some commonly-used information systems. I highlight the ideas that TOM has built on, and indicate the limitations of existing systems that TOM transcends.

3.2.1 Client-server systems

Most Internet information systems, TOM included, are based on client-server connections. This form of information retrieval has been supported by the Internet from its early days, in services like FTP [B$^+$71]. The amount of information retrieved from servers on the Internet, though, increased substantially after the introduction of Gopher [AML$^+$93] and the World Wide Web [BLCGP92]. Adie [Adi94] describes the state of information retrieval on the Internet shortly after the release of widely usable Web clients.

Why did client-server information retrieval become so much more heavily used at this point? Part of the increase in such traffic can be attributed to the general growth of the Internet already in progress at the time. Part of it can also be attributed to the existence of user-friendly clients, which were easier to use than the common interfaces for FTP.

However, the Web also has a key advantage over simple FTP in its basic design. The Web *provides a component architecture and data model* for the data on the Internet that makes it much easier for users to access a wide variety of data sources than they could using FTP alone. The advantages of the Web's data model, for example, can be seen in the URL, a form of metadata used for references to Web documents. ("Selector strings" in Gopher play an analogous role to URLs in the Web.) The URL allows references to many different kinds of servers, without being tied to one protocol like FTP; hence, it gives access to a wider variety of information sources. It also allows data on one server to reference data on another server, and supports automatic retrieval of the referenced data, through hyperlinks with embedded URLs. Because of this facility, users of the Web can easily "surf" through related data on a variety of sites, rather than having to deal with data one repository at a time. This ability to link data on diverse servers in turn gives rise to informal component architectures on the Web. For instance, when a Web user is looking for data on a particular subject, he often uses a "search engine" server as a mediator for data that is actually stored on a variety of other servers. The search engine enables the user to find the required information on a single server, without having to individually check each server that might have the data.

Just as it is difficult to find particular information on the Internet using FTP alone, so today it is often difficult to find particular data type definitions and services operating on data on the Internet today. TOM's architecture makes these definitions and services easier to locate and use, just as the Web makes data easier to locate and use.

The design of the Web and Gopher has limitations, however. First, its reference model for data on Web servers is server-specific. If data moves from one Web server to another, old references to the data will no longer work. Therefore, Web pages and Gopher menus become less useful over time, unless the hyperlinks on the pages are updated periodically. A server-specific reference convention also creates bottlenecks for popular data. A project or group that suddenly makes headlines, for example, may find its Web site quickly swamped. More flexible, server-independent reference schemes, such as URNs, have been proposed [SM94], but have not yet been widely adopted, in part because existing browsers cannot use them.

TOM, in contrast, uses more flexible reference schemes. TOM's type definitions and services are named independently of the server that supplies them. Hence, all TOM clients can choose the most convenient of multiple servers that contain this information. This reference model allows TOM's infrastructure to scale more gracefully than the Web. TOM also supports Web or Gopher references, when appropriate, along with many other kinds of references. These references can all be grouped under a common *reference* supertype, allowing new TOM clients to use new kinds of references via a well-known generic interface.

A second limitation of the Web model is that it provides no facilities to coordinate data that is best spread out over multiple servers. Instead, the data on each Web server is maintained completely independently from the data on other Web servers. Two common techniques other systems have developed to distribute and coordinate data services are *replication* and *partitioning.*

Replicating information on multiple servers avoids overload on any one server, and can also reduce the average latency for data retrieval. Popular Web servers or FTP collections, for example, are often replicated on multiple "mirror sites," which spread out the load and reduce the average distance between server and client. (However, there is no automated way to find mirror sites using

FTP alone.) Going further, Usenet news uses “flooding” to distribute its newsgroups [KL86]. In flooding, an article posted on one Usenet server propagates to other “peer” servers, until every Usenet server that carries the article’s newsgroups has its own copy of the article. Readers then simply read articles from their local news server. More complex replication strategies are required when the replicated information needs to be secure, as in Lotus Notes.

TOM also takes advantage of replication to propagate type information through a network of type brokers. Other kinds of data replication can also be handled by appropriate reference types defined in TOM, though TOM itself does not define these types.

Partitioning information spaces also distributes loads in a large system, and independent agents working in the same information space from conflicts or unnecessary redundancy. In a partitioned system, the architecture of the information system is organized so that data appears on a particular server determined by some aspect of the data. Clients may then look on different servers depending on the kind of information they seek. For example, a simple partitioned database may assign indexing information to one server, and full records on a second server. In this way, one server can handle searches and another retrievals, allowing the two servers to handle twice as many client requests without having to replicate data. Hierarchical information space partitioning is also common. For example, the Internet’s Domain Naming System [Pos94a] uses different name servers to manage different parts of the Internet’s naming hierarchy. If a client tries to resolve a domain name in the `cmu.edu` domain, the client might first consult a name server that manages top-level `edu` domains. That top-level name server might then refer the client to a name server at Carnegie Mellon that manages `cmu.edu` domain names. TOM uses a similar hierarchy to partition its own space of type names, and even uses the same data hierarchy as DNS for a portion of its type namespace.

In a partitioned information space, it is useful to have a *directory* to locate appropriate servers. For example, WAIS [Kah91] users seeking information can first consult a central directory of servers to find out which WAIS servers are likely to contain information they seek, and then run searches on the appropriate WAIS servers. Name services, such as the DNS example above, also provide essential metadata for directory services. More informally organized directories, such as catalog pages and Net-wide search engines, are used on the Web. Examples of such search engines include Alta Vista, Lycos, Yahoo, and Infoseek. Such directories can be considered simple examples of mediators between clients and the servers that ultimately contain the desired information. Mediators are at the heart of TOM’s design, and we will describe other mediator-based systems in more detail in the next subsection.

3.2.2 Systems with mediators or expert agents

Mediators and other “expert” agents make large information systems easier to use. Gio Wiederhold [Wie92] cites numerous uses for mediators, such as finding servers with information a client seeks, coordinating multiple operations to fulfill a user’s request, and making systems more fault-tolerant and reliable.

A mediator can be as simple as a directory, such as a WAIS directory of servers mentioned above, or a Web search engine. But many systems also put mediators in more active and central roles. Some use mediators as a single, easy-to-use front end for a collection of services. The University of Colorado’s Netfind system [SP94] is a essentially a mediator of this type used for locating people. Given a name or other personal information, Netfind sends queries to services on the Web that maintain personal data (including WHOIS databases, finger and SMTP servers, and Usenet archives) to locate a profile of the person or persons sought. Netfind does not have any special name database of its own, other than a cached, and it gets its information from existing, directly-accessible databases. But it can be significantly easier for users to make one query to Netfind about a person than it would be to query all of the databases Netfind consults. The same mediator design principle can be found in some other Internet search systems, such as MetaCrawler [SE97], which queries several major Web search engines at once and returns the combined results to a user. Still more advanced mediator architectures are used in the work of Katia Sycara *et al* [SPW$^+$96]. Sycara has designed an architecture of mediator agents that collect and integrate information from diverse sources, and has successfully applied this architecture to applications that include organizational decision-making and financial portfolio management.

Besides acting as a common front end for existing services, mediators can also add *abstractions* of data not given in the original servers. The Rufus [SLST93] system of Shoens *et al.*, for example, provides object-oriented interfaces to "semi-structured" data, such as those found in filesystems or email. In OEM [PGMW95], mediators translate data formats and queries into a common intermediate form, which makes it easier for disparate clients and servers to work together. Unfortunately, translating to and from the intermediate form sometimes results in a loss of information or of efficiency. TOM, as we will see in later chapters, supports translations of data formats into forms that diverse clients and servers can use, but does not require that the formats go through a single "standard" intermediate form.

Mediators can also work together to locate and synthesize new data. One suitable architecture for this task is a network of distributed *brokers* that guide clients to appropriate servers, and that also allow new information to be propagated and stored in appropriate servers. For example, the Indie system [DLO92] uses brokers to keep track of searchable databases on a variety of subjects. When a broker is sent a query, it determines what servers should be searched, and forwards the request to appropriate parties. When servers receive new information, they can inform brokers about it, who can then pass it along to other servers or brokers that might be interested in it. Hence, the brokers act as mediators both for clients and for servers, and help coordinate independently managed information sources.

As we will see in later chapters, TOM takes advantage of many of these mediator techniques. Its type brokers provide a uniform interface to a variety of services, as do Netfind and MetaCrawler. TOM brokers provide object-oriented abstractions of various data types, like Rufus. TOM brokers can also work together to exchange information, and refer clients to appropriate services, like Indie. TOM goes considerably farther than these other systems do, however, by combining a sophisticated component model with a powerful data model.

3.2.3 Data description mechanisms (non-object-oriented)

The data model of an information system is as important as the component model, since it determines what kinds of data can be used in an information system and what can be done with the data. However, the data models of most of the systems described in the previous section are severely limited. In the next portion of this chapter, I examine some of the data models used widely on the Internet, and explain their limitations in comparison to TOM.

Some information systems actually have no data model at all, or a very limited one. For instance, FTP, still a common method for distributing files, does not have any mechanism for describing data other than distinguishing ASCII text transmissions from binary transmissions. Programs that retrieve files via FTP can only infer the data format of a file from the contents of the file, the ASCII text indicator, or the suffixes used in the name of the file being retrieved. These methods are limited, ad-hoc, and error-prone.

Most modern information systems, in contrast, do attempt to describe their data in some way, so that clients can identify and work with them more easily. At this time, the most commonly used bases for data description are MIME and SGML. Both have their merits, but also are limited in important respects.

MIME

MIME [BF92] was originally designed to identify and encode structured data in electronic mail messages. It has since been used to identify data formats in other information systems, including the World Wide Web. MIME identifies data formats with "MIME types", an expandable set of identifiers grouped in eight categories, including text, image, and audio. The meaning of each "MIME type" and its format is described in a file in a centralized MIME registry. New formats can be added by writing a new type description and registering it with the central MIME authority. Users can also unilaterally declare their own "experimental" formats in a special namespace, but if they want a new format to be recognized by standard MIME tools, they need to register it under a non-experimental name. MIME is flexible enough to be applied to a wide range of popular data formats on the Internet, and is at present the standard method for identifying formats both in electronic mail and on the World Wide Web.

However, MIME has some important limitations, both in its scalability and its semantic model. Its longtime dependence on a centralized registry has meant that only a limited number of formats have been registered. A new data format can first be brought into use as an "experimental type", but there is no easy way to get information about this experimental type, and the experimental name becomes obsolete when the type is centrally registered.

In 1994, noting that the registration of new MIME types was slow, the MIME administration decided not to require a published specification of a format as a prerequisite for registration [Pos94b]. While the number of MIME-registered types increased as a result, many of the registrations have very little information about the formats. They allow a format to be identified, but give very little information about the syntax and semantics of the format.

Even if a semantic specification is given, MIME's semantic model is limited. For instance, it does not clearly distinguish between what information a type conveys and how this information is represented. It is often appropriate for the same information to be conveyed in different ways, but MIME requires there to be one standard representation of a particular type of information. Alternate representations are not supported, except through a small set of standard encodings, or by tagging alternate representations with extra parameters whose meaning is uninterpreted by MIME.

MIME's handling of multiple representations is also limited. MIME defines a small repertoire of encodings that can be applied to any MIME format, but this repertoire is small and fixed, and cannot be applied recursively. As a result, the standard GIF format, and a compressed version of the GIF format would be two different "MIME types," as the system exists now. A uuencoded compressed GIF would have to be yet another format. Widespread use of multiple encodings, then, leads to a combinatorial explosion in MIME types. Clearly this scheme does not scale well.

Very little of MIME's data format descriptions can be automatically parsed by a program. While MIME has conventions for extracting the name of the MIME type from a MIME-encoded file, and defines some standard semantics for its fixed encoding repertoire, and for multipart MIME files, MIME provides no further automatable support. The descriptions of a MIME format's syntax and semantics are not provided in a machine-parsable form. Therefore, while a new MIME format may be parsable by a human reader, or by a code specially written for that format by a human reader, MIME provides no mechanism for a program to automatically make use of an unfamiliar MIME type. In contrast, SGML does provide such a mechanism, as described in the next section; and TOM does as well, through its data model.

Despite MIME's limitations, it is used widely enough that it must be accommodated by any information system that takes full advantage of existing data. TOM accordingly allows all MIME formats to also be defined as TOM types, and it also allows these formats to be recursively encoded. TOM also accommodates many other data types, and permits them to be defined in greater detail than MIME does. We will see how this is done in later chapters.

SGML

SGML [Pub86] takes a different approach to describing data from MIME's. SGML defines a standard for data formats that are mechanically parsable, extendable, and self-describing. SGML is expressive enough to encode many kinds of structured documents, and has therefore been the basis of many popular data formats. The best-known SGML format at present is HTML, the standard format for Web pages. Other significant SGML formats include HyTime [NKN91] and TEI [Ini94]. XML [Fla97] is a simplified version of the SGML standard, designed to make it easier to define and parse formats.

The main body of a SGML document consists of a sequence of nestable *elements*, delimited by *tags* that can include *attributes*. For example, in HTML the "anchor" element is used to define hyperlinks. An anchor element starts with a tag named "A". This tag may have attributes such as HREF, which gives the location of the document referenced by the hyperlink. Following the initial tag is text that will appear highlighted in the hyperlink. Finally, the element ends with a closing tag (`</A>`, in this case).

SGML data formats conform to standard syntactic conventions for the tags and elements of the document. SGML formats are defined by a coded syntactic description, called a "data type definition" or DTD, that lists the tags and elements that can appear in the document, and how they are parsed. (DTDs can even override some of the standard SGML syntactic conventions.) The DTD can be enclosed as part of an object in an SGML format.

DTDs allow SGML-aware applications to parse any SGML format they encounter, even if they have not seen it before. Known SGML formats can also be "extended" with new tags. The extended formats are often still usable by programs written for the older formats; typically those programs simply ignore the extra tags or elements.

While SGML is well-designed for expressiveness and extensibility, it has some serious limitations as a general data model. SGML can only handle data formats that follow the general SGML syntactic conventions, such as the use of tags delimited by angle brackets to structure data. Much information available on the Internet does not use these conventions, and is therefore not usable by SGML-only tools. Also, while SGML allows for extensive support of the *syntax* of structured documents, it does not describe the *semantics* of these documents. For example, although the SGML DTD for HTML, tells a program that there is an an element named "A" with an attribute "HREF", the DTD does not say in any machine-understandable for what the HREF attribute is meant to do (namely, refer to the location of a Web document).

The structure of SGML documents, and of SGML itself, can be modeled by appropriate TOM data types. TOM can also go further than SGML, since TOM accommodates a wider range of representation formats, and has facilities for describing the semantic structure, as well as the syntactic structure, of networked documents.

3.2.4 Object-oriented data description mechanisms

Some data models now in use on the Internet do provide semantic support for data. Specifically, *networked object-oriented systems* allow remote data to be manipulated through an object-oriented framework, by calling methods or getting and setting attributes.[1] The interfaces provided for these objects describe how they can be manipulated.

Numerous networked object systems have been proposed and built as research prototypes. Three systems that have been widely adopted in practice are the Object Management Group's CORBA [Gro92], Microsoft's COM (used as the basis for OLE and ActiveX), and Sun Microsystems's Java.

In COM, applications and data are encapsulated as objects, with one or more "interfaces". Through COM, one can select a particular interface to an object, and then call methods on that interface to read or manipulate the object. In the basic COM implementation, these calls are made on the same machine, but an extension to COM (known as DCOM) allows such calls to be made over a network. COM objects tend to be large-grain, and may wrap entire applications like Word. Method calls can modify the objects and produce side effects. There is no transparent support for "migrating" objects between machines.

CORBA was designed from the start to be networked. In CORBA, objects live on a particular server, and are managed by an "object request broker", through which clients call methods or get attributes. The objects are modifiable, and generally do not move from the server machine to other machines. One can add routines to make remote copies of objects, but this is not supported in the basic CORBA model.

In both CORBA and COM, determining what can be done with an unfamiliar object can be difficult. COM allows clients to query whether a particular object has a particular interface known to the client, but it does not allow clients to simply ask for all the interfaces a COM object has. The CORBA model includes a proposal for optional "interface repositories" that include information about commonly used object types, but it does not have any mechanism for keeping track of object types at a global, Internet-wide scale. Hence, in practice clients may have a difficult time using objects that are not of well-known types (or that do not have well-known COM interfaces.)

The representation of CORBA and COM objects is also inflexible. The representation of CORBA objects is normally determined by the server on which the CORBA objects reside, and is not made available to the client. COM objects are typically represented in a binary form defined by Microsoft. Neither system can handle information represented in other forms (such as common MIME formats or SGML) without special code or gateways.

Objects in Java [GM95] are more mobile than those in CORBA or COM. Java objects move between machines, as does the code that implements Java methods. However, the representations of Java objects, like CORBA and COM objects, are limited. Java objects can be transmitted in "serialized" form, either in a default representation or in programmer-defined representations, but there is no way to name or declaratively describe different representations, or relate them to each other. Implementation of operations on Java objects also need to be written in Java. Therefore, Java itself can only easily accommodate a limited subset of existing document representations and operation implementations. However, Java's design is simple and open enough that gateways between Java-native code and data and other applications can be built more easily than in CORBA or COM.

[1] Logically, manipulating attributes can be thought of as a special case of calling methods. Some systems implement fetching attributes through a "get-method", and setting attributes through a "set-method". In this thesis, I will use the term "operation" as a generic term covering method calls and attribute fetches, as well as covering the "conversions" that are described in the next chapter.

By themselves, all of these object-oriented systems are insufficiently flexible to handle the large corpus of data that has been defined independently of these systems, since they each require that objects (or code) be represented in their preferred format. They are ill-equipped to handle legacy data, and other systems already in common use. While these systems allow new data types to be defined, the formats of these data types are constrained by the system implementations. Moreover, they lack adequate mechanisms to make a new data type widely usable once it is defined, since there are few facilities for disseminating new data type information, or for relating new types to existing types, short of going through the bottleneck of a recognized standards body.

None of the data models described in this section, then, can adequately handle the wide variety of data structures (including data structures and formats already in use) that TOM is designed to handle. In the chapters to come, I will examine in more detail how TOM's design allows clients to take full advantage of the syntax and semantics of an ever-growing set of data formats, including both brand-new and decades-old formats.

Chapter 4

The Typed Object Model

The next two chapters present the basic design of the Typed Object Model (TOM for short), the model and system that are the basis of this thesis. The design has two main aspects: an object-oriented data model that describes both the abstract structure and the concrete representations of data, and a distributed broker architecture that allows this model to be applied and used in actual systems.[1]

In this chapter, I describe the data model. I give definitions of TOM objects, their types, their object-oriented interfaces, and their representations. I introduce the concept of *conversion* between data formats. I describe what information goes into a type definition, the motivations for including this information, and the similarities and differences between TOM objects and objects in standard object-oriented models. I close with examples of TOM types, to illustrate the use of the concepts described in this chapter.

4.1 Overview of the data model

TOM models data as typed, immutable objects. These objects can be queried via an *interface* much like the class interfaces of object-oriented languages such as C++ or Java. Objects can also be *encoded* as simpler objects (including simple byte sequences), allowing them to be transmitted over the Internet or stored in retrieval systems like the World Wide Web. As with Smalltalk-80 [GR83], TOM's data model is reflective: that is, type definitions, references to objects, and other metadata are also represented as TOM objects.

TOM's data model uses nine basic constructs for describing and interacting with data. Each is described in detail in this chapter. To summarize: *Objects* are the basic unit of data. *Types* classify an object, and give clients a way of finding out what can be done with an object. *Method* and *attribute/* specifications allow objects to be used through abstract, object-oriented interfaces. A *subtyping* model allows programs to use unfamiliar types through the familiar interfaces of their supertypes. *Encodings* specify how objects are represented in more concrete forms, allowing applications to use information at varying levels of abstraction. *Formats* specify a representation of objects as byte sequences, allowing objects to be stored in filesystems and shipped from one application to another as *shipped objects*. *Conversion* allows an unfamiliar data format to be transformed into a familiar one, while controlling information loss. Table 4.1 summarizes these data constructs.

[1] As I mentioned earlier, I will use the term "TOM" in two senses: a strict sense, denoting the data model itself, and a looser sense, denoting the overall architecture that uses the data model. I will clarify my usage when it might be ambiguous.

Concept	**Purpose**	**Modeled as**	**Notes**
Object	To encapsulate data	A typed value	Objects are immutable information, not variables.
Type	To classify data	An element of a basic set	Types are named (possibly with multiple names).
Encoding	To represent data	A partial, one-to-many relation between objects of an encoded type and objects of an encoding type	Corresponds to "representation" or "refinement relation" in other OO models.
Format	To represent data as transmittable sequences of bytes	A type and a composable sequence of encodings	MIME "types" are examples of formats
Shipped object	To transmit and store data	An object and a format	Corresponds to a "marshalled" or "serialized" object in other networked OO models.
Attribute	To extract information from an object	A specification of functions on objects	Implemented by any function satisfying the specification. Not dependent on context or additional parameters.
Method	To derive information from an object	A specification of functions on objects	As above, but can take parameters and be context-dependent. Cannot mutate the object (since objects are immutable).
Subtype	To allow objects to be used through familiar supertype interfaces	A relation between types	Subtypes must be "substitutable" for supertypes.
Conversion	To transform objects in unfamiliar formats to corresponding objects in familiar formats	A specification of functions on shipped objects	Information retention can be specified through "intersubstitutability."

Table 4.1: Major concepts of TOM's data model.

As we will see, this diversity of data constructs is both necessary and sufficient to accommodate the range of applications TOM is required to handle. Each construct is kept as simple as possible, and I have avoided adding unnecessary features or constructs. For example, TOM does not support user-defined exceptions, default or keyworded method arguments, or complex metaclass protocols. It also does not support operations that mutate objects, as I noted in the introduction. In some cases, these features were omitted because they were not necessary to satisfy the requirements of the thesis. In other cases, however, extra features would make it more difficult for TOM to satisfy some of its requirements, or make it more difficult to integrate TOM with other systems that do not support these features.

In the sections that follow, I describe each basic construct of the TOM data model, giving its rationale, its definition, and its function, both in prose and in more precise formal notation. For the formal descriptions, I use a variation of the Z specification language [PST91], augmented for higher-order logic. The basic concepts of the data model should be understandable independently of the formal notation.

4.2 Objects, values, and types

The basic unit of data in TOM is the *object*. An object consists of a *type* and a *value*. For example, for an object representing the contents of a text file, the value of the object is the text in the file, and the type is *text*. An integer object might have the value 3, and the type *integer*.[2]

In TOM, objects are *not* variables, as they are in most object-oriented models. They are not references to data, or containers of data, but the data itself. For example, for a text document consisting of the words "I am Sam," TOM considers the *text*, not the file where the text is stored, as the object. An exact copy of this text on another machine would be the *same* object, from TOM's point of view. On the other hand, if someone then edited the file so that it said "Sam I am" instead, the new contents of the file would be considered a *different* object.

The value of an object need not be a byte sequence. The color green, the sound of Martin Luther King's "I Have a Dream" oration, and the temperature at the Pittsburgh airport at 9 AM on October 28, 1995, are all legitimate TOM object values. However, values that are byte sequences are particularly useful to TOM, since computers can easily store and communicate such values. Other kinds of values can also be used and transmitted by TOM programs if they can be represented in digital form. TOM handles such values through *encodings*, described later in this chapter.

The *type* of an object describes *how to interpret* the object. Objects of a given type may be constrained to a certain range of values. The type may (but is not required to) specify an *interface* that describes attributes of an object, methods that can be called on the object, and the semantics of these attributes and methods. For example, the *integer* type has the usual arithmetic semantics, and an interface that includes addition, multiplication, equality testing, and other arithmetic operations. The *integer* type also constrains its values to be numbers, and prohibits non-integral numeric values like 3.5 or π.

Formally, an object is a pair consisting of a *value* and a *type*. Values and types come from the primitive domains *Value* and *Type*. For any object o, $value(o)$ is the value part of the pair, and $type(o)$ is the type part of the pair. Equality operators exist for both types and values. As we saw when discussing integers, object types may restrict the set of their legal values. Hence, there

[2] To eliminate ambiguity in classifying objects, TOM agents explicitly tag an object with an explicit type name when they transmit it. We will see later how this is done. Types may be implicit for data not directly managed by TOM agents.

exists a total function called *typevalues* that gives the set of values associated with a particular type. This function can then be used to define *Object*, the set of *all* valid objects:

$$\begin{array}{l} \mathit{typevalues} : \mathit{Type} \longrightarrow \mathit{set\ of\ Value} \\ \mathit{Object} == \{t : \mathit{Type}; v : \mathit{Value} \mid v \in \mathit{typevalues}(t)\} \end{array}$$

The *extent* function, related to *typevalues*, defines all the valid objects in a type:

$$\begin{array}{l} \mathit{extent} : \mathit{Type} \longrightarrow \mathit{set\ of\ Object} \\ \quad \forall\, t : \mathit{Type} \bullet \mathit{extent}(t) = \{o : \mathit{Object} \mid \mathit{type}(o) = t\} \end{array}$$

Note that the extents of different types are disjoint, since the type is part of the object.

An alternate way to model the extent of a type is to define a function *constraint* from types to predicates. For any type T, the predicate $\mathit{constraint}(T)$ must be satisfied by all objects of the type. Formally, if *Pred* is the set of all possible predicates, *constraint* can be defined as follows:

$$\begin{array}{l} \mathit{constraint} : \mathit{Type} \longrightarrow \mathit{Pred} \\ \quad \forall\, o : \mathit{Object} \bullet \mathit{constraint}(\mathit{type}(o))(o) \Leftrightarrow \mathit{value}(o) \in \mathit{typevalues}(\mathit{type}(o)) \end{array}$$

(Note that the definition above uses higher-order logic, not supported in Z.)

The constraint function is useful for defining the extent of types with an infinite number of legal values. It can also be used to formally express semantic restrictions on a type's value. TOM itself does not specify the syntactic structure or semantics for the predicates in *Pred*, but the predicates can be interpreted using any appropriate formal system.

4.3 Encodings, decodings, and formats

Objects must be represented in a concrete form accessible to programs, In order for programs to transmit and operate on them.

If the value of an object is already a finite sequence of bytes, then it can be directly represented in computers and transmitted across a network. Types of objects with such values are particularly important types in TOM. We will call the set of such types *ByteSeqType*, the set of their values *ByteSeq*, and the set of their objects *ByteSeqObject*. A *ByteSeqObject* is transmitted over a network by sending the byte sequence that is the object's value, along with the name of its type and certain other meta-information. (We will see later in this chapter how this meta-information is transmitted.)

What about objects with other kinds of values? Objects whose values are not sequences of bytes can be transmitted if one can *encode* the object as a *ByteSeqObject*. An *encoding* is a refinement relation between a set of objects of one type (the *encoded type*), and a set of objects of an another type (the *encoding type*) that *represent* objects of the encoded type. A refinement relation is a mapping that matches objects of one "abstract" type with corresponding objects of another, more "concrete" type, that represent the abstract objects.[3]

If an object O can be encoded as *ByteSeqObject* B, then O can be transmitted across the network by transmitting the value of the object B, along with enough meta-data about types and encodings to allow the recipient to reconstruct the original object O. We will see shortly how to do this.

[3] Encodings can also map objects of one type to other objects of the *same* type. For example, a *ByteSeqObject* may be encoded as another *ByteSeqObject* with a shorter value, in an encoding meant for data compression.

Encodings are one-to-many relationships between objects of the encoded type and objects of the encoding type, but encodings need not be total. For example, an encoding of integers as 32-bit sign-extension byte sequences allows two different byte sequences (one with the sign bit set to 1, the other with it set to 0) to represent zero. Hence, this encoding is not one-to-one, but one-to-many. The encoding is also not total, because it is incapable of representing large integers. In its restricted domain, however, it is unambiguous. No byte sequence in the encoding type encodes more than one integer. There is no byte sequence in the encoding, for instance, that could be interpreted as either 4 or -4.

Because encodings can be partial, and because many types (such as integers) are often represented in different ways, TOM allows a variety of encodings to be defined for the same type. For example, one encoding of integers may use 32-bit sign-extension, while another encoding uses a twos complement, little-endian byte sequence. A third kind of encoding might use an ASCII encoding, representing the integer 300 with the string "300".

The ASCII encoding of integers illustrates how one-to-many encodings support robust parsing and interpretation of objects. For example, to parse ASCII integers robustly, whitespace and extraneous punctuation should be considered irrelevant. A tolerant ASCII encoding, then, would treat "300", " 300.0 ", and "+300" as all encoding the same integer.

Based on the discussion above, we formally define *Encoding* as the set of all legal encodings; that is, partial one-to-many relations between objects of an encoded type and objects of an encoding type. We can also define the functions *encodingtype* and *encodedtype* that return the encoding type and encoded type of any encoding. (Because extents of types are disjoint, *encodingtype* and *encodedtype* are functions).

$$\begin{aligned}
&\mathit{Encoding} == \{r : \mathit{Object} \longleftrightarrow \mathit{Object} \mid \\
&\qquad (\exists\, t_1, t_2 : \mathit{Type} \bullet r \in (\mathit{extent}(t_1) \longleftrightarrow \mathit{extent}(t_2))) \land \\
&\qquad (\forall\, x_1, x_2, y : \mathit{Object} \bullet ((x_1, y) \in r \land (x_2, y) \in r \Rightarrow x_1 = x_2))\} \\
&\mathit{encodingtype} : \mathit{Encoding} \longrightarrow \mathit{Type} \\
&\qquad \forall\, e : \mathit{Encoding} \bullet \operatorname{ran} e \subseteq \mathit{extent}(\mathit{encodingtype}(e)) \\
&\mathit{encodedtype} : \mathit{Encoding} \longrightarrow \mathit{Type} \\
&\qquad \forall\, e : \mathit{Encoding} \bullet \operatorname{dom} e \subseteq \mathit{extent}(\mathit{encodedtype}(e))
\end{aligned}$$

The inverse of an encoding relation is a *decoding function*, an abstraction function from objects of the encoding type to objects of the encoded type. Decoding functions are partial, and many-to-one. In our ASCII-encoded integer example, our decoding function might be the equivalent of the C library's `atoi()` function.

Formally, we can define a function *dfun* that returns the decoding function of any encoding:

$$\begin{aligned}
&\mathit{dfun} : \mathit{Encoding} \longrightarrow (\mathit{Object} \nrightarrow \mathit{Object}) \\
&\qquad \forall\, e : \mathit{Encoding};\, o_1, o_2 : \mathit{Object} \bullet (o_1, o_2) \in e \Leftrightarrow \mathit{dfun}(e)(o_2) = o_1
\end{aligned}$$

Or, more succinctly, if o is an object, E is an encoding that includes that object in its domain, and D is E's decoding function, then

$$D(\!|E(\!|\{o\}|\!)|\!) = \{o\}$$

Since objects are typically transmitted and stored as sequences of bytes, we are particularly interested in ways to represent objects as *ByteSeqObject*s. A *format* specifies such a representation. Formats provide a concrete syntax for objects of a particular type that allows them to be stored in filesystems and transmitted across networks.

Informally, formats can be considered to be a special kind of encoding that maps between abstract objects and objects whose values are sequences of bytes. However, a more flexible definition for formats is a *sequence* of already-defined encodings that, when composed, encode an object of the desired type as a *ByteSeqObject*. If the format's type is itself a *ByteSeqType*, there may be no encodings needed. Some formats, however, may involve multiple encodings. For example, suppose that a *directory* type is encoded as a *list of strings* type, and the *list of strings* type is encoded as a sequence of bytes. In this case, a format for the *directory* type would include both encodings: first the encoding that maps from *directory* to *list of strings*, and then the encoding that maps from *list of strings* to *byte sequence*.

There are distinct advantages to defining formats as sequences of encodings. Doing so allows data to be interpreted at different levels of abstraction. In the directory case, clients can either operate on the object as a directory, or as a list of strings, or as an uninterpreted byte sequence. Composing encodings also allows a large number of formats to be specified using a relatively small set of encodings. For example, the encoding that maps from *list of strings* to byte sequences may be reused in formats for many other types besides directories.

A format, then, consists of a type combined with a sequence of encodings that, when composed, represent an object of that type as a *ByteSeqObject*. Formally:

$$
\begin{array}{l}
\mathit{Format} : \mathit{set\ of}\ (\mathit{Type} \times \mathrm{seq}\ \mathit{Encoding}) \\
\quad \forall\, t : \mathit{Type} \bullet (t, \langle\rangle) \in \mathit{Format} \Leftrightarrow t \in \mathit{ByteSeqType} \\
\quad \forall\, t : \mathit{Type};\ \langle e_1 ... e_n \rangle : \mathrm{seq}\ \mathit{Encoding} \bullet (t, \langle e_1 ... e_n \rangle) \in \mathit{Format} \Leftrightarrow \\
\qquad (\exists\, o_1 : \mathit{extent}(t), o_2 : \mathit{ByteSeqObject} \bullet (o_1, o_2) \in e_1 \fatsemi e_2 \fatsemi ... e_n)
\end{array}
$$

Note that while TOM formats specify a representation of an object as a sequence of bytes, TOM types (except for *ByteSeqType*s) do not. Therefore, the equivalent of a "MIME type" in TOM would be a format, not a type, because "MIME types" include a specification about how documents are represented as byte sequences. For example, the MIME type specification for a GIF file specifies both the graphical information given in a GIF, and the syntax used for that graphical information. Similarly, the "file types" of many filesystems would also correspond to TOM formats, and not to TOM types.

Object-oriented languages typically allow the implementation of an object to be defined separately from its interface, and hide the underlying representation of an object from the user of an object. TOM, like these other OO systems, allows implementations to be fully separated from an object's abstract definition. Other OO systems, however, typically do not provide the general facility for defining the representation of an object down to the level of byte sequences, as TOM does. Instead, most OO languages represent objects in a "standard" internal form defined by the language implementation. In contrast, TOM's distinction between types, encodings, and formats also allows the representations of objects (at multiple levels) to be fully specified by the type definer, and cleanly separated from the object's abstract definition. This separation between an object's syntax and its semantics gives greater flexibility in using and specifying information.

For instance, in the upcoming discussion of subtyping, we will see how objects of different types and formats can substitute for each other, and be used by the same programs, even if they are represented in completely different ways. (This fundamental feature of objects is not present in systems like MIME, which have "subtyping" but not abstraction.)

4.4 Getting concrete: shipped objects

Shipped objects are the means by which TOM objects are transmitted over a data connection, such as an Internet connection. A shipped object includes all the information needed to define a particular object, including its value (encoded in some format), and the specification of the format used to encode the object for transmission. Some object-oriented systems call these units of data "marshaled" or "serialized" objects.

This section gives a formal model of shipped objects. It explains how *metadata* such as type names, encoding names, and format designations, are represented and communicated in TOM.

We have already seen that certain object values (*ByteSeq*s) are sequences of bytes, and are therefore directly transmittable. Similarly, type information can be transmitted by sending an additional byte sequence that names the type. Legal type names are taken from the set *TypeName*, a subset of *ByteSeq* that uses a restricted alphabet of ASCII characters. Every type that is declared in TOM has one or more type names, used only by that type. We can therefore define a partial function *namedtype* that maps from a type name to a type:

$$namedtype : TypeName \rightarrowtail\!\!\!\!\!+ \; Type$$

where *namedtype*(*name*) returns the type named *name*.

To transmit a *ByteSeqObject* o, then, one transmits the name of the object's type (that is, the *TypeName* n for which $namedtype(n)$ is $type(o)$), along with the sequence of bytes that is $value(o)$. Such objects are known as *directly shippable objects*, since the object's value is already in the form of a byte sequence, and can be transmitted along with the type name without extra transformation. Receivers of the shipped object can use the type name to look up complete information on a type. (Chapter 5 describes how this lookup works.)

Encodings, like types, are also named by byte sequences, taken from a restricted subset of *ByteSeq* named *EncName*. Thus, information about encodings can also be transmitted in the same way as type names. TOM requires that each encoding of a type have a different name from other encodings of the same type, so that an encoding is uniquely identified by the type of object it encodes and the name of the encoding.[4] Hence, the naming function for encodings looks like this:

$$\begin{array}{l} namedencoding : (Type \times EncName) \rightarrowtail\!\!\!\!\!+ \; Encoding \\ \quad \forall\, t : Type;\, n : EncName \bullet (\mathrm{dom}(namedencoding(t, n))) \subseteq extent(t) \end{array}$$

From what has been shown so far, we can show that any object with a named type can be transmitted across the network if one of the following holds:

- the object is *directly shippable* (that is, it is a *ByteSeqObject*); or
- there exists a sequence of encodings which, when composed, map from the object to a *ByteSeqObject*.

The first case is obvious: for a *ByteSeqObject* O, one transmits the name N of O's type, and the byte sequence V that is O's value. Then O can be derived from N and V as $(namedtype(N), V)$.

In the second case, let B be the *ByteSeqObject* to which O maps. Let T be the name of the type of O, and let E be the sequence of encoding names. (B, T, and E can all can be transmitted as byte sequences, as described above.) Let S be the sequence of encodings applied to O that map it to B. We can determine S recursively, using T, and E: S_1 is $namedencoding(namedtype(T), E_1)$,

[4] We could have made *all* encodings have unique names, like types, but relaxing the rule to allow reuse of encoding names for different types permits shorter, more mnemonic encoding names. For more discussion of naming issues, see Chapter 6.

and S_{n+1} is $\mathit{namedencoding}(\mathit{encodingtype}(S_n, E_{n+1}))$. We can then reconstruct the original object O from B, T, and E by applying all the corresponding decoding functions to B in reverse order.

An object that can be transmitted this way, either directly or through a series of encodings as above, is called a *shippable object*. The form used for transmission is called a *shipped object*, and consists of a directly shippable object plus the names of the encodings required to map the original object into the directly shippable object. This is equivalent to the directly shippable object combined with the specification of a *format*:

$$
\begin{array}{l}
\mathit{ShippedObject} : \mathit{set\ of}\ (\mathit{ByteSeq} \times \mathit{Format}) \\
\quad \forall\, b : \mathit{ByteSeq};\ t : \mathit{Type} \bullet (b, (t, \langle\rangle)) \in \mathit{ShippedObject} \Leftrightarrow t \in \mathit{ByteSeqType} \wedge (t, b) \in \mathit{Object} \\
\quad \forall\, b : \mathit{ByteSeq};\ (t, \langle e_1 ... e_n\rangle) : \mathit{Format} \bullet (b, (t, \langle e_1 ... e_n\rangle)) \in \mathit{ShippedObject} \Leftrightarrow \\
\qquad (\exists\, o : \mathit{extent}(t) \bullet (o, (\mathit{encodingtype}(e_n), b)) \in e_1 \fatsemi e_2 \fatsemi ... e_n)
\end{array}
$$

For any *ShippedObject* s, I define $\mathit{bytes}(s)$ as the byte-sequence value used in s, $\mathit{format}(s)$ as the format used in s, and $\mathit{obj}(s)$ as the original object transmitted by s (after all decoding functions specified in $\mathit{format}(s)$ have been applied).

4.5 Subtypes

The discussion above shows how TOM models objects of various types, and their transmission. There is no limit to the possible number of types; the two developers of TOM at Carnegie Mellon have themselves defined over 100, and many thousands could be easily be defined by a larger group. Yet most programs can handle only a relatively small number of types and formats. How can these programs work with other formats they might encounter?

TOM's approach to solving this problem is to define *relations* between types. If an program encounters an object of a type it was not programmed to deal with, but discovers that the unfamiliar type is related to a familiar type, then it may be able to work with the object using its knowledge of the more familiar type.

We have already seen one such relation in this chapter. The *encoding* relation allows an program to operate on a representation of an object, rather than on its abstract type. For example, suppose that an program receives an object of type *catalog-record* encoded as ASCII text. The program may know nothing about the *catalog-record* type. But it may suffice to work with the object's more familiar encoded *text* type. For example, using the text encoding of the *catalog-record* object, the program might be able to display it in a document, save it to a file, or perform any other applicable text-based operation on the object.

Object-oriented languages introduce another important kind of relation between types: the *subtype* relation. The interface of subtypes includes the interface of their parent types (also known as their *supertypes*). Thus, a program that encounters an object with a type that is a subtype of a known type can use all the attributes and methods defined in the known supertype.

Programs usually expect the subtype to behave like the supertype. In practice, though, for most object-oriented languages there is no guarantee or requirement that the behavior of the subtype match the behavior of the supertype, or even have any predictable relationship to the behavior of the supertype. For example, in C++ subtypes inherit method names and signatures from their supertypes, but the method implementations can be arbitrarily overridden. Therefore, objects in subtypes cannot always be used in the same way that objects of the supertypes would be. In a non-distributed system, this may not be a problem, because programmers usually know about the complete type hierarchy and know when and how a particular subtype can be used in place of a supertype, or at least have access to code that includes this information.

For a distributed system like TOM, however, this ad-hoc approach will not work. The developers of types may be thousands of miles apart from each other, and have no knowledge of what type designers had in mind when defining a type, except for what is actually stated in the type definition. Hence, without clear subtyping rules, one may not be able to make any predictions about the behavior of subtypes. TOM therefore uses a *substitutability* model of subtyping, which is stricter than ad-hoc subtyping. If type S is a subtype of type T, then an object of type S can *substitute* for an object of type T. This means that when a client program uses an object of type S through T's interface, the object will behave exactly as the client would expect an object of type T to behave.

With substitutable subtypes, clients can safely use objects of types they have never seen before. As long as an unfamiliar type is a subtype of a known type, a client can use the object through the known type's interface, and it will behave in accordance with the semantics of the known type. (We say that the object *conforms* to the known type.) Substitutability thus lets a client work with many more types and formats than it otherwise could. A client can even work with newly introduced data types without any code changes. TOM permits a type to have multiple supertypes, increasing the possibility that a new type will be usable through a familiar supertype.

Having informally introduced the principle of subtyping in TOM, I now give the basic formal model of the relation. TOM's subtyping model is essentially the one defined by Liskov and Wing [LW94], except that since TOM objects are immutable, "history properties" need not be considered.

We define the relation $subtype(S, T)$ to denote that type S is a subtype of type T. Formally, if this relation holds, there must be an abstraction mapping from *all* objects in type S to objects in type T. (An abstraction mapping is a relation between objects of one type and corresponding objects of another, more "abstract" type, in this case the supertype.) We will call this mapping $typeabs(S, T)$. Objects in type S are *substitutable* for objects in type T via the abstraction mapping. We write $conforms(o, t)$ to say that object O conforms to type T; that is, its type is T or one of the subtypes of T:

$$conforms(o, t) \Leftrightarrow subtype(type(o), t)$$

Subtyping is transitive and reflexive. Transitivity follows as well, since if there exists $typeabs(S, T)$ and $typeabs(T, U)$, then one need only compose those functions to get $typeabs(S, U)$. The composition is always well-defined, if the individual functions are, because $typeabs(T, U)$ is constrained to be defined for all objects of type T. Reflexivity ($subtype(T, T)$) also follows, by using the identity function as the abstraction mapping.[5]

Figure 4.1 shows a small portion of TOM's type hierarchy. The actual names TOM assigns to types are in `typescript` font (as they are throughout this thesis), and their informal names appear below the actual TOM names. TOM's type hierarchy is rooted in a top-level *Object* type, as this figure shows. *Object* (or `e:obj`) is a supertype of all other types, and therefore all objects conform to this type. Types can have multiple subtypes and multiple supertypes. For example, the MIME multipart message type, which includes information defined in the *sequence* type and information defined in the *package* type, is a subtype of both of those types.

4.6 Type interfaces: methods and attributes

So far, I have described the types, values, and encodings of objects, and shown how objects can be transmitted between agents. How does an agent use the information contained in objects? For some

[5] I have purposely not included anti-symmetry; hence, I have not ruled out the possibility of equivalence classes of types. The current implementation of TOM does not provide any explicit support for them, though, and I will not consider them in this thesis. Equivalence classes would pose new problems for implementations and naming, however.

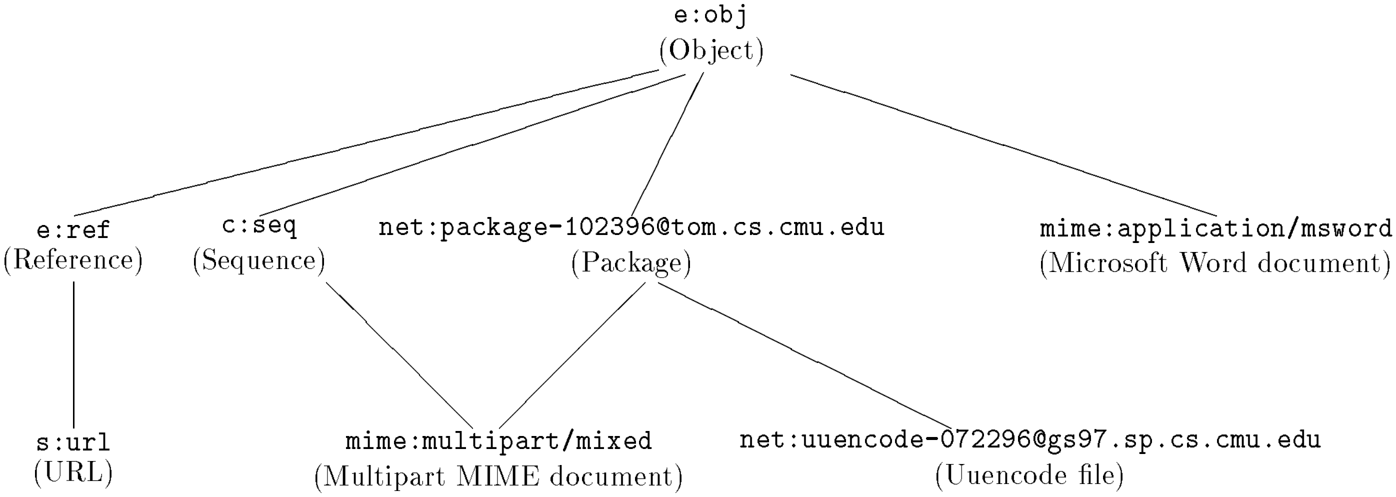

e:obj
(Object)
e:ref
(Reference)
c:seq
(Sequence)
net:package-102396@tom.cs.cmu.edu
(Package)
mime:application/msword
(Microsoft Word document)
s:url
(URL)
mime:multipart/mixed
(Multipart MIME document)
net:uuencode-072296@gs97.sp.cs.cmu.edu
(Uuencode file)

Figure 4.1: A portion of TOM's type graph. Lines in the figure represent subtype relations.

formats, an agent is already equipped to interpret the value of an object, or perform computations on it. But in other cases, an agent may know *what* data needs to be extracted or computed from an object, but not know *how* to get this data. Object-oriented systems solve this problem by defining an *interface* for an object's type. The interface defines *attributes* that can be retrieved, and *methods* that can be computed on an object to yield new data. Many object-oriented languages also allow attributes to be set to new values, and allow methods to change the internal state of an object. TOM does not, since its objects are immutable, but it does allow attributes and methods to create new objects.

The sections that follow discuss how attributes and methods are defined for TOM object types. I will discuss methods first, since in TOM attributes are essentially a special kind of method. In the section on attributes, I explain how and why attributes are defined in this manner.

4.6.1 Methods: functions and specifications

A *method* is a function which, given an object of a specified input type, possibly some additional objects as parameters, and possibly some implicit context, returns either an object conforming to some specified output type, or an exception. For example, an `born-before` method for biographical record objects, called on Thomas Aquinas' biographical record, and given Tom Hanks's biographical record as an input parameter, returns a boolean object with value `true`, since Thomas Aquinas was born before Tom Hanks.

The interface of a TOM type includes *method specifications* for methods that can be called on objects of that type. A method specification includes a *name* used for calling methods of that specification, and a *signature*, which states what additional input parameters the methods take, the types the input parameters must conform to, and the type the return value will conform to (if an exception is not returned). Method specifications can also include constraints on what objects will be returned for various inputs and contexts, and can specify when an exception can or cannot occur. For example, the specification of `born-before` might indicate that it always returns the value `true` (and not an exception) when the `birth-year` attribute of both inputs is defined, and the `birth-year` of the first object is less than that of the second.

Note that, strictly speaking, method specifications are distinct from methods, since methods are defined to be functions. Informally, however, we often use the term "method" to describe either methods themselves or method specifications (such as when saying "the interface defines two methods"). When the meaning of "method" is not clear from context, I will use the term "method specifications" when appropriate, and will use the term "method function" for methods themselves.

For any given method specification, there may be more than one method function that satisfies the specification. Consider a method specification called `pick` on *sequence* objects that returns an element of the sequence. A method that always returns the first element in the sequence satisfies this specification, but so does a method that always returns the last element in the sequence. If a client calls `pick` on a *sequence* object, the result may be computed by either of these methods, or any other method satisfying the specification. Some method specifications, in fact, work best when they are not completely constrained. For example, when specifying an `OCR` method to recognize text displayed in a graphical image, one may wish to accommodate any algorithm that produces text above a certain accuracy threshold, rather than spell out in minute (and possibly inaccurate) detail exactly how all OCR methods must determine an image's textual content. A given method specification can be further constrained in subtypes, as we will see later in this chapter.

The value returned by a method call may be dependent on an implicit *context* distinct from the explicit parameters supplied to a method. For example, the result of a `fetch` method on a *URL* object depends on the state of an HTTP server somewhere in the World Wide Web, but this state is not explicitly provided in the `fetch` method call. (Since the full specification of this state may require multiple gigabytes, it is fortunate that the state is *not* so provided.) The state of the server, then, is part of the *context* of the method call, and the method specification can stipulate that the result of `fetch` is dependent on it.

Contexts model all external phenomena (such as the time of day, or the state of a filesystem) that could affect the results of a computation, and that are not given by explicit parameters. It is possible to obtain information about a given context, but it is not possible in general to determine whether two contexts are identical, since that would constrain the possible phenomena of contexts to those considered in an equality test. Defining contexts this way allows us to model processes that do not have a predictable outcome, but do have a known pattern or distribution of behaviors, such as a random number generator. Method specifications do not usually note contexts except when a method's result is specified to be independent of context, or when the result depends only on certain aspects of context (such as the state of an HTTP server in the *URL* example above).

Exceptions are results returned when a method call cannot return a normal object as a result. TOM does not specify anything about exceptions except that they can be recognized as exceptions. Some programming languages have more elaborate exception models, and spell out different types of exceptions, or allow exceptions to have the same structure as ordinary objects. TOM does not provide this level of detail for exceptions, since it is unnecessary to satisfy TOM's basic requirements.[6]

Given the discussion above, I can provide a formal definition of methods. A method function takes an object, a context, and a sequence of parameters (which are also objects), and returns either an object or an exception, if it returns. Method functions are partial, since method computations may not always terminate. *Object* was defined earlier; *Context* is a primitive set with no predefined structure, as is *Exception*. Hence, the set of method functions, *Methodfun*, is defined thus:

[6] Adding this feature to TOM, however, could be easily done. In Chapter 9, I briefly discuss what would be involved.

$$\mathit{Methodfun} : \mathit{set\ of}\ ((\mathit{Object} \times \mathit{Context} \times \mathrm{seq}\ \mathit{Object}) \mathrel{+\!\!\!\!\rightarrow} (\mathit{Object} \cup \mathit{Exception}))$$

A method specification consists of a name, a type for which the method specification is defined, and a set of method functions that match the specification. The name is from the set *MethodName*, a subset of *ByteSeq*.

$$\begin{array}{l} \mathit{Methodspec} : \mathit{set\ of}\ (\mathit{Type} \times \mathit{MethodName} \times \mathit{set\ of\ Methodfun}) \\ \quad \forall (t, n, s) : \mathit{Methodspec} \bullet \forall f : s \bullet \forall (o, c, p) : \mathrm{dom} f \bullet \mathit{conforms}(o, t) \end{array}$$

We define the functions *mtype*, *mname*, and *mfuns* to extract the elements of a *MethodSec* tuple. We also require that the names of method specifications be unique for a given type, so that method calls are unambiguous:

$$\begin{array}{l} \mathit{mtype} : \mathit{Methodspec} \longrightarrow \mathit{Type} \\ \mathit{mname} : \mathit{Methodspec} \longrightarrow \mathit{MethodName} \\ \mathit{mfuns} : \mathit{Methodspec} \longrightarrow \mathit{set\ of\ Methodfun} \\ \quad \forall (t, n, f) : \mathit{Methodspec} \bullet \\ \quad\quad \mathit{mtype}((t, n, f)) = t \wedge \mathit{mname}((t, n, f)) = n \wedge \mathit{mfuns}((t, n, f)) = f \\ \quad \forall\, m_1, m_2 : \mathit{Methodspec} \bullet \\ \quad\quad (\mathit{mtype}(m_1) = \mathit{mtype}(m_2) \wedge \mathit{mname}(m_1) = \mathit{mname}(m_2)) \Leftrightarrow m_1 = m_2 \end{array}$$

We also define $\mathit{mresulttype}(m)$ to be the return type of the method's signature, and $\mathit{margtype}(m, i)$ to be the type of input parameter i in the method's signature, if there are at least i parameters. For any method specification m, $\mathit{mtype}(m)$ is known as "the method's type".

$$\begin{array}{l} \mathit{mresulttype} : \mathit{Methodspec} \longrightarrow \mathit{Type} \\ \mathit{margtype} : (\mathit{Methodspec} \times \mathbb{N}) \mathrel{+\!\!\!\!\rightarrow} \mathit{Type} \\ \quad \forall\, m : \mathit{Methodspec} \bullet \forall f : \mathit{mfuns}(m) \bullet \\ \quad\quad (\forall\, r : \mathrm{ran} f \bullet r \in \mathit{Exception} \vee \mathit{conforms}(r, (\mathit{mresulttype}(m)))) \wedge \\ \quad\quad (\forall (o, c, s) : \mathrm{dom} f;\ i : \mathbb{N} \bullet \mathit{conforms}(s_i, \mathit{margtype}(m, i))) \end{array}$$

When discussing the extent of a type, I defined a *constraint* function that, given a type, returned a predicate that was true of all the objects of a type, and no others. Similarly, it is also useful to define a constraint function that determines whether a given method function satisfies a method specification. This constraint function, *mconstraint*, is defined as follows:

$$\begin{array}{l} \mathit{mconstraint} : \mathit{Methodspec} \longrightarrow \mathit{Pred} \\ \quad \forall f : \mathit{Methodfun}, m : \mathit{Methodspec} \bullet \mathit{mconstraint}(m)(f) \Leftrightarrow f \in \mathit{mfuns}(m) \end{array}$$

Another useful function, *mrange*, gives the set of all objects that could be the result of a particular method call:

$$\begin{array}{l} \mathit{mrange} : (\mathit{Methodspec} \times \mathit{Object} \times \mathit{Context} \times \mathrm{seq}\ \mathit{Object}) \mathrel{+\!\!\!\!\rightarrow} \mathit{set\ of}\ (\mathit{Object} \cup \mathit{Exception}) \\ \quad \forall\, m : \mathit{Methodspec};\ o : \mathit{Object};\ c : \mathit{Context};\ s : \mathrm{seq}\ \mathit{Object};\ r : (\mathit{Object} \cup \mathit{Exception}) \bullet \\ \quad\quad r \in \mathit{mrange}(m, o, c, s) \Leftrightarrow (\exists f : \mathit{mfuns}(m) \bullet f(o, c, s) = r) \end{array}$$

4.6.2 Class methods

Along with the normal methods described above, TOM also allows "class methods" to be associated with types. They are defined and specified as above, except that they are not invoked on a particular object (and hence only take a context and a parameter list). Class methods are most commonly used to return new objects of the type for which they are defined.[7] For example, for a *cartesian-point* type, one could define a class method `newpoint` that takes integers `x` and `y` and returns a *cartesian-point* object with `x` and `y` attribute values equal to the supplied parameters. Class methods are also sometimes used when a method takes several objects of the same type, when a method designer prefers not to give one of those objects special status over the other objects in the method call.

4.6.3 Attributes: functions and specifications

Attributes are used to extract information contained in an object. The only information used to calculate an attribute is already present in the object and the type definition. Attributes, like methods, are essentially functions, but they take no additional parameters, and are not context-dependent. In addition, attributes do not return exceptions, and can always be computed in finite time.

The namespace for attributes is separate from the namespace for methods; that is, a type can have an attribute specification named `part` and an unrelated method also named `part`. Programmers should try to avoid using the same name for an attribute and a method, due to possible confusion; however, such overlap sometimes occurs when operations from different supertypes are redeclared.

TOM's definition of attributes as functions differs from many other object-oriented systems. Languages like C++, for example, simply treat attributes as variables (if the object is mutable), or symbolic constants (if the object is immutable). This approach is feasible in these systems because all objects share the same underlying representation, and attribute access is a built-in feature of the language or run-time environment shared by all components in the system.

TOM's distributed environment demands a different approach. Not only are objects, and therefore attributes, represented in any number of formats, but the objects themselves may not be in the same address space as that of the program that needs to extract attributes. Therefore, attribute extraction needs to be done through an external program or subroutine, and hence is best defined as a function call.

Modeling attributes as functions rather than as constants or variables also allows attributes to be defined that are not explicitly stored in an object, but computed from other information. For example, the *cartesian-point* type mentioned above may also have a *magnitude* attribute that measures the point's distance from the origin. This attribute can be computed from the `x` and `y` attributes, and need not appear explicitly in the object's encodings. (This ability to compute attributes as needed is also present in certain Lisp-based object models that use IF-NEEDED constructs.)

The formal definition of attributes is based on the formal definition of methods, with the simplifications described above:

[7] This does not conflict with our earlier principle that methods do not mutate objects. Class methods can create new objects, but do not mutate existing ones.

$$\begin{array}{l}
\mathit{Attributefun} : \mathit{set\ of}\ (\mathit{Object} \longrightarrow \mathit{Object}) \\
\mathit{Attributespec} : \mathit{set\ of}\ (\mathit{Type} \times \mathit{AttributeName} \times \mathit{set\ of\ Attributefun}) \\
\quad \forall (t, n, s) : \mathit{Attributespec} \bullet \forall f : s \bullet \forall o : \operatorname{dom} f \bullet \mathit{conforms}(o, t) \\
\mathit{atype} : \mathit{Attributespec} \longrightarrow \mathit{Type} \\
\mathit{aname} : \mathit{Attributespec} \longrightarrow \mathit{AttributeName} \\
\mathit{afuns} : \mathit{Attributespec} \longrightarrow \mathit{set\ of\ Attributefun} \\
\quad \forall (t, n, s) : \mathit{Attributespec} \bullet \\
\qquad \mathit{atype}((t, n, s)) = t \wedge \mathit{aname}((t, n, s)) = n \wedge \mathit{afuns}((t, n, s)) = s \\
\quad \forall a_1, a_2 : \mathit{Attributespec} \bullet \\
\qquad (\mathit{atype}(a_1) = \mathit{atype}(a_2) \wedge \mathit{aname}(a_1) = \mathit{aname}(a_2)) \Leftrightarrow a_1 = a_2 \\
\mathit{aresulttype} : \mathit{Attributespec} \longrightarrow \mathit{Type} \\
\quad \forall a : \mathit{Attributespec} \bullet \forall f : \mathit{afuns}(a) \bullet \forall r : \operatorname{ran} f \bullet \mathit{conforms}(r, \mathit{aresulttype}(a))
\end{array}$$

Note that the attribute specifications, like method specifications, can allow multiple distinct functions for the same attribute. This is useful when an attribute's value is not well-defined for certain objects. The default assumption for attributes is that they are "well-defined"; that is, all functions of an attribute specification will return the same attribute value for a given object. However, for added flexibility, TOM allows the attribute's value to be declared undefined in certain circumstances. For example, if an *integer* type includes a `sign` attribute that is either `+` or `-`, it may specify that the sign of 0 is undefined, since 0 is neither positive or negative. When an attribute value is undefined, an attribute function may return any value consistent with the attribute signature, or no value at all. While undefined attributes give implementations more flexibility, they should be used with care. Clients need to be able to determine when an attribute is not well-defined. For example, in the *integer* case above, the `0` value of the integer may be a tipoff that sign is unimportant. In other cases, one might define auxiliary attributes or methods that indicate whether certain other attributes are well-defined.

4.6.4 Attributes and methods in subtypes

In my previous discussion of subtypes, I noted that there must be an abstraction mapping between objects of a subtype and objects of the supertype, and that the subtype must be substitutable for the supertype. Therefore, every attribute and method specification of a supertype has a corresponding specification in a subtype, so that the attribute or method can be used on objects of the subtype. Furthermore, the semantics of the subtype's method or attribute specification must imply the semantics of the supertype specification. That is, the subtype specification must guarantee all the behavior that the supertype specification guarantees, when called with the same inputs and context.

A method specification m in type T can be declared to specialize a corresponding specification in supertype S. We call that specification $\mathit{superspec}(m, S)$. Similarly, an attribute specification in a subtype can be declared to specialize a corresponding specification in a supertype. In many object-oriented languages, specialization is implicit when a method or attribute name from a supertype is reused in a subtype. In TOM, the specialization must be declared explicitly, for reasons that will be explained in Chapter 6.

We can then state the following rules for specializing methods (taken from or based on those given by Liskov and Wing [LW94]):

1. Method results must *covary.* That is, the return type of the subtype's method must be a subtype of the return type of the supertype's method. (Since the subtype relation is reflexive,

the return type can be the same type for both methods.) Hence for all types S that have a *superspec* defined for method specification m,

$$\mathit{subtype}(\mathit{mresulttype}(m), \mathit{mresulttype}(\mathit{superspec}(m, S)))$$

2. Method parameters must *contravary.* That is, the parameters of the supertype's method must be subtypes of the parameters of the subtype's method. Hence for S as above, and i a valid argument index for method specification m,

$$\mathit{subtype}(\mathit{margtype}(\mathit{superspec}(m, S), i), \mathit{margtype}(m, i))$$

3. Semantic constraints on the subtype's method specifications must imply the constraints on the supertype's method specifications, when applied to the supertype abstraction of a given object in the subtype. That is, the results allowed by the subtype method must also be allowed in the supertype method. Hence, with T, S, and m as above, and p a valid parameter list for the supertype method:

$$\begin{array}{l} \forall\, o : \mathit{extent}(T);\, c : \mathit{Context};\, p : \mathrm{seq}\, \mathit{Object};\, r : \mathit{Object} \bullet \\ \quad r \in \mathit{mrange}(m, o, c, p) \Rightarrow \\ \qquad \mathit{typeabs}(\mathit{type}(r), S)(r) \in \mathit{mrange}(\mathit{superspec}(m, S), \mathit{typeabs}(T, S)(o), c, p) \end{array}$$

Corresponding rules apply for attributes, and for class methods.

Note that it is permissible for a subtype method to signal an exception in a case when a supertype method might not have, so long as the supertype's method semantics do not *prohibit* an exception in that case. A subtype method must return without an exception in any case where the supertype method is required to return without an exception.

We can illustrate these relations with an example. Consider a *list of objects* type that has a method `elem` that takes an integer i and returns the ith item in the list, if it exists. Consider a subtype of this type, *list of string objects*, whose `elem` method returns the ith string in the list, and an exception if there is no ith string. Is this a valid subtype method? We apply the three rules above to find out.

1. The return type of the subtype method is *string.* The return type of the supertype method is *object.* The *string* type is a subtype of the generic *object* type, so the requirement for a covarying result is satisfied.

2. The parameter of both the supertype method and the subtype method is of type *integer.* Thus, the supertype's method parameter is trivially a subtype of the subtype's method parameter, since subtyping is reflexive. Thus, the requirement for contravarying parameters is satisfied.

3. The semantic constraints for the subtype are the same as those for the supertype, except that the return value is constrained to be a string, and an exception is specified if the parameter is out of range. (The supertype's method semantics did not specify what happened if the input parameter was out of range, and therefore permitted any value, or an exception, to be returned in this case.) For both of these differences, the requirements imposed by the subtype's stricter semantics imply the requirements imposed by the supertype's looser semantics. Thus, the third requirement is satisfied.

We therefore conclude that the `elem` method for the *list of strings* type is a legitimate specialization of the `elem` method for the *list of objects* type.

For a similar example that uses nontrivially contravarying input parameters, suppose that the `elem` method for *list of strings* took any real number (and not just an integer), and in nonintegral cases returned a tail of a given string. For instance, if given 1.7 as a parameter, the routine would return the tail of string 1, omitting the first 70 percent of the string. This routine would also be a legitimate specialization. The return type still covaries; the parameters contravary (assuming integers are subtypes of reals); and the semantics of the subtype method still imply those of the supertype method, since for integral parameters the entire string is returned, as before.

4.7 Conversions

In many cases, it is better for a program to operate on data directly, rather than going through an object-oriented interface. Programs can operate directly on the value of a shipped object if it is in a known format, but most programs are not designed to operate on more than a few formats. To handle more formats, TOM programs can invoke *conversions* to transform shipped objects into familiar formats. Conversions can map between formats of different types, or between formats of the same type. The results of conversions can also be given to programs that have no knowledge of TOM. Conversions, then, allow a wide range of programs to share data in mutually understandable forms.

For example, consider an image-display program that can display images in the standard GIF format. If that the program retrieves an image in the PNG image format, a format it does not know how to parse, it may not be able to display the image. In theory, the program might be able to make a long sequence of method calls defined on the PNG type to determine pixel by pixel what should be displayed on the screen. In practice, though, it is far easier and quicker to have the PNG object converted to a GIF object, and then display it just like any other GIF image.

TOM associates *conversion specifications* with types, just as it associates attributes and method specifications with types. However, because conversions work with *representations* of objects, a conversion function takes as input *shipped objects* in a given format. (Attributes and methods, in contrast, take *objects* conforming to a given *type*.) A conversion takes no additional arguments. It returns a shipped object that corresponds to the original object, encoded in a new format. For example, the image conversion above would be defined on the `standard` *byte-sequence* encoding of the *PNG image* type, and return the `standard` *byte-sequence* encoding of the *GIF image* type. Because TOM's subtypes need not represent objects the same way that supertypes do, subtypes do not "inherit" conversions.

Conversions map from shipped objects of one format to shipped objects of another format. Conversions may be partial. Given these requirements, we can define *Conversion*, the set of all possible conversions:

$$\begin{aligned}
&\mathit{Conversion} == \{f : \mathit{ShippedObject} \rightarrow\!\!\!\!\!\!+\; \mathit{ShippedObject} \mid \\
&\quad \forall\, o_1, o_2 : \operatorname{dom} f \bullet \mathit{format}(o_1) = \mathit{format}(o_2) \wedge \\
&\quad \forall\, o_1, o_2 : \operatorname{ran} f \bullet \mathit{format}(o_1) = \mathit{format}(o_2)\}
\end{aligned}$$

We then define *Conversionspec* analogously to *Methodspec* and *Attributespec*. We use the same sort of naming scheme for conversions as we used for attributes and methods. Specifically, a conversion can be uniquely identified by its name (taken from *ConversionName*, a subset of *ByteSeq*) and the type of the shipped objects the conversion takes as input. The functions *ctype*, *cname*, and *cfuns* are defined analogously to their counterparts for methods and attributes. We also add the functions *cfromfmt* and *ctofmt* to specify the input and output format of the conversion specification:

$$\begin{array}{l}
\mathit{Conversionspec} : \mathit{set\ of}\ (\mathit{Type} \times \mathit{ConversionName} \times \mathit{set\ of\ Conversion}) \\
\quad \forall (t, n, c) : \mathit{Conversionspec} \bullet \forall f : c \bullet \forall s : \operatorname{dom} f \bullet \mathit{conforms}(\mathit{obj}(s), t) \\
\mathit{ctype} : \mathit{Conversionspec} \longrightarrow \mathit{Type} \\
\mathit{cname} : \mathit{Conversionspec} \longrightarrow \mathit{ConversionName} \\
\mathit{cfuns} : \mathit{Conversionspec} \longrightarrow \mathit{set\ of\ Conversion} \\
\mathit{cfromfmt} : \mathit{Conversionspec} \longrightarrow \mathit{Format} \\
\mathit{ctofmt} : \mathit{Conversionspec} \longrightarrow \mathit{Format} \\
\quad \forall (t, n, s) : \mathit{Conversionspec} \bullet \\
\qquad \mathit{ctype}((t, n, s)) = t \wedge \mathit{cname}((t, n, s)) = n \wedge \mathit{cfuns}((t, n, s)) = s \\
\quad \forall c_1, c_2 : \mathit{Conversionspec} \bullet \\
\qquad (\mathit{ctype}(c_1) = \mathit{ctype}(c_2) \wedge \mathit{cname}(c_1) = \mathit{cname}(c_2)) \Leftrightarrow c_1 = c_2 \\
\quad \forall c : \mathit{Conversionspec} \bullet \forall f : \mathit{cfuns}(c) \bullet \\
\qquad (\mathit{format}(\!|\operatorname{dom} f|\!) = \{\mathit{cfromfmt}(c)\} \wedge \mathit{format}(\!|\operatorname{ran} f|\!) = \{\mathit{ctofmt}(c)\})
\end{array}$$

The function *crange*, given a conversion specification and a shipped object, returns the set of all shipped objects that could result from a conversion allowed by that conversion specification:

$$\begin{array}{l}
\mathit{crange} : (\mathit{Conversionspec} \times \mathit{ShippedObject}) \mathrel{+\!\!\!\!\!\rightarrow} \mathit{set\ of\ ShippedObject} \\
\quad \forall c : \mathit{Conversionspec}; s_1, s_2 : \mathit{ShippedObject} \bullet \\
\qquad s_2 \in \mathit{crange}(c, s_1) \Leftrightarrow (\exists f : \mathit{cfuns}(c) \bullet f(s_1) = s_2)
\end{array}$$

The complete *Conversion* set includes arbitrary mappings between formats. Intuitively, though, a conversion is not just an arbitrary mapping, but one in which the input and the output of a conversion are somehow "equivalent", even if they are in different formats. By "equivalent", we mean that the output of a conversion conveys approximately the same information as the input of a conversion. For example, one would expect that when the PNG image in the example above was converted to the GIF format, the resulting GIF would look either identical, or at least very similar, to the original PNG image when displayed.

In practice, conversions do not always preserve all the information given in the original object. Sometimes they cannot preserve all the information, such as when one converts from a complex format to a simple format. Therefore, one must ask: *What information* does a given conversion actually preserve?

In TOM, the concept of *substitutability*, which we defined in our introduction to subtypes, can be adapted to express how "equivalent" the result of a conversion is to the original object. For any conversion specification c, there exists a type T such that for any shippable object o, o and all the elements of $\mathit{crange}(c, o)$ are substitutable for each other through the interface of T. That is, the interface of T cannot discriminate between the objects. (Of course, o and the elements of $\mathit{crange}(c, o)$ must all conform to type T.) The proof that such a type always exists is trivial, since the topmost type in TOM's type hierarchy cannot distinguish between any two objects. In TOM, type T is known as an *intersubstitutable type* of conversion specification c, and we assert $\mathit{intersub}(c, T)$. (Since "intersubstitutable" is a rather long word, the term "top type" is sometimes used in place of "intersubstitutable type".)

Type T is an intersubstitutable type for conversion specification c if the abstraction mapping to type T of both the input and the output of the conversion resolve to the same object. Formally:

$$\begin{aligned}&\forall\, T : \mathit{Type};\ c : \mathit{Conversionspec} \bullet\\ &\quad (\forall\, s_1 : \{s : \mathit{ShippedObject} \mid \mathit{format}(s) = \mathit{cfromfmt}(c)\} \bullet \forall\, s_2 : \mathit{crange}(c, s_1) \bullet\\ &\qquad \mathit{typeabs}(\mathit{type}(\mathit{obj}(s_1)), T)(\mathit{obj}(s_1)) = \mathit{typeabs}(\mathit{type}(\mathit{obj}(s_2)), T)(\mathit{obj}(s_2)))\\ &\quad \Rightarrow \mathit{intersub}(c, T)\end{aligned}$$

The requirement that all results of a conversion must map to the same object in T as the input of a conversion is actually stronger than necessary. (We therefore call this requirement *strong intersubstitutability.*) For intersubstitutability, it suffices for the interface of T to be unable to distinguish between $\mathit{obj}(s)$ and $\mathit{obj}(c(s))$. (Or, to put it another way, no method or attribute specification will mandate different results when invoked on $\mathit{obj}(s)$ than when invoked on $\mathit{obj}(c(s))$, if inputs and context is otherwise the same.) This condition is obviously satisfied if the abstraction of the inputs and outputs of a conversion are equal, as stated above, but can also be satisfied if they are sufficiently similar.

Formally, we can express this weaker intersubstitutability requirement as follows: Let c be a conversion specification and T a type. Suppose that for all methods of T, the inputs of conversion c can produce the same results as the outputs, that is:

$$\begin{aligned}&\forall\, m : \{m : \mathit{Methodspec} \mid \mathit{mtype}(m) = T\};\ s_1 : \{s : \mathit{ShippedObject} \mid \mathit{format}(s) = \mathit{cfromfmt}(c)\} \bullet\\ &\quad \forall\, s_2 : \mathit{crange}(c, s_1);\ x : \mathit{Context};\ p : \mathrm{seq}\ \mathit{Object} \bullet\\ &\qquad \mathit{mrange}(m, \mathit{obj}(s_1), x, p) = \mathit{mrange}(m, \mathit{obj}(s_2), x, p)\end{aligned}$$

If the statement above is true, and corresponding guarantees hold for class methods and attributes, then T is an intersubstitutable type for c.

Our image conversion example can be used to illustrate intersubstitutability. Suppose that the *GIF image* and *PNG image* types are both subtypes of a generic *pixel map* type that specifies the colors of pixels in a rectangular region. GIF images are limited to 256 distinct colors, unlike PNG images. If the generic *pixel map* type does not have such a color limit, and a conversion accepts PNG images with more than 256 colors, then the generic *pixel map* type is not an intersubstitutable type for the conversion, since it is possible to use the *pixel map* interface to distinguish a PNG image with thousands of colors from its conversion to a GIF image with only hundreds of colors. On the other hand, suppose that the *pixel map* type is in turn a subtype of a more generic *bitmap* type, that simply records whether a graphical element is set or clear. Let us further suppose that the abstraction mapping between the *pixel map* type and the *bitmap* type is that bits are considered set if they are not black, and clear if they are black. As long as the PNG to GIF conversion does not change any non-black color to black (or black to non-black), and otherwise preserves the pixel layout, there is no way for the *bitmap* interface to distinguish the PNG image from the GIF image that results from the conversion. The *bitmap* type, then, would be an intersubstitutable type for this conversion.

When a conversion specification is declared in TOM, it can include the assertion that a type T is an intersubstitutable type for the conversion, and that the conversion therefore preserves the information in T. The further down in the type hierarchy T is, the stronger is this guarantee about information preservation. Because subtypes are required to be substitutable for supertypes, if type T is an intersubstitutable type for a conversion c, then all of T's supertypes are also intersubstitutable types for that conversion.

Conversion specifications can also specify whether they are `trivial`. A trivial conversion is one that is total on all shipped objects of the input format, and maps to shipped objects with the same byte sequence value, tagged with the output format. (In other words, the conversion is accomplished simply by relabeling the format of the shipped object.) Formally, if a conversion specification c is trivial, then

$$\forall\, s : \mathit{ShippedObject} \bullet \mathit{format}(s) = \mathit{cfromfmt}(c) \Rightarrow \mathit{crange}(c, s) = \{(\mathit{bytes}(s), \mathit{ctofmt}(c))\}$$

4.8 Conversions: notes on their design and use

Conversions are often invoked automatically. For instance, a method or attribute implementation designed to work with one format might need to convert its inputs to that format, if they are supplied in a different format. The design of conversions in TOM is meant to make these automatic conversions as straightforward as possible.

The declaration of intersubstitutable types allows methods to safely convert input arguments to formats they recognize. Suppose, for instance, that a program computes an attribute or method for an object of type T, but expects to receive shipped objects in a particular format F of T. If the program receives a shipped object in another format of type T (or its subtypes), the object can be converted to format F via any conversions whose specifications declare type T (or a subtype of T) as an intersubstitutable type. These declarations, if accurate, guarantee that the results of the computation with the converted object are consistent with the computation allowed for the original object.

Because intersubstitutable type declarations can be useful for making these guarantees, it may sometimes be useful to define a type used only as an intersubstitutable type, and not as a type with instances. For example, when defining a conversion from LaTeX to HTML, one may find it useful to define a virtual supertype of both types that includes information about section titles. If one declares this type as an intersubstitutable type of the conversion, one effectively declares a constraint that the conversion preserves section title information.

Because conversions take no additional arguments, they can easily be automatically composed. This property makes it feasible for a program to ask for a conversion of an arbitrary object to another format, without having to specify by name which conversions to use. A type broker or other program can determine what conversions need to be invoked, and compose them, without any additional inputs, to produce the desired result. If T_1 is an intersubstitutable type for conversion c_1, and T_2 is an intersubstitutable type for conversion c_2, then any supertype of both T_1 and T_2 is an intersubstitutable type for the composition of these conversions.

More details about how type brokers plan conversion strategies appear in Chapter 5.

4.9 Semantic specifications: notes on implementation

At various points in the previous discussion, I noted that semantic specifications can be associated with a type, or with various aspects of a type, such as its attributes and methods. But I have said little about how these semantic specifications are defined in practice.

The signatures of methods, attributes, and conversions, the intersubstitutable types of conversions, the supertypes of types, and the names of types and encodings, can be directly declared in a type definition. A conversion specification can also be declared to be `trivial` (as defined above) in a type definition. Other semantic constraints or interpretations discussed in this chapter, such as the *constraint* predicate defined for types, can be expressed in whatever form is clear and convenient to the definers and implementors of types, whether this form is "formal" (e.g., a Z declaration) or informal (e.g., an English-language description).

TOM's conventions do constrain what these open-ended declarations can assert. In particular, statements about *representations* of objects cannot appear in the semantic constraints of *types* (except for the special case of *ByteSeqTypes*), but only in the semantic constraints of *encodings*. This rule preserves the separation of an object's abstract interface and its concrete representation.

Semantic constraints, like other aspects of a type, are themselves represented as objects. The objects may simply be text strings, in the case of an informally-specified type or operation, or they may have more highly structured types (such as those that express formal constraints). TOM itself does not automatically check or enforce the statements made in these open-ended semantic objects, but simply records the specifications and makes them available to clients and implementors.

One advantage to this approach to constraints is that it gives type definers flexibility. They are not locked into a constraint notation that may be cumbersome, inadequately expressive for certain constraints, and difficult to learn to use properly. Type definers can also make specifications as tight or as loose as is appropriate for the type. (There are, in fact, many kinds of data already on the Internet that are only loosely or informally specified, but are common enough to be worth including in TOM's type graph.) In many cases, aspects of the type's behavior that are open to variation or interpretation can be specified more precisely in subtypes. We will see an example of this practice in the next section, when discussing Microsoft Word document types.

One problem with this approach, though, is that when semantic constraints are not expressed formally, programs cannot automatically enforce them. Hence, while TOM can *require* certain behavior of types and methods, in practice it cannot actually *guarantee* this behavior. For example, it is possible for someone to define a type S that is declared to be a subtype of T, where the informal semantic declarations for S break the substitutability requirement TOM demands of subtypes.

On the other hand, certain semantic constraints may not translate easily into a particular formal notation. Even if they are easily expressible in formal notation, it might not be practical for the system to automatically enforce the formal constraints, since enforcement may require excessive overhead, and is in general undecidable.

When semantics are stated using a structured type, it may be possible for programs to check them. For example, a precondition stated for a method might be automatically checked before the method is invoked. The current implementation of TOM provides no built-in facilities for making these checks, but implementors of clients and servers may add them if they wish.

In Chapter 8, I discuss in further detail the tradeoffs involved in this approach to semantic constraints, and discuss how well the approach has worked in practice.

4.10 Examples of TOM types

Having described the basic aspects of the type model, I now illustrate its use through examples.

4.10.1 Example 1: Uniform Resource Locators

The uniform resource locator, or URL, is the basic *reference* type for the World Wide Web, and was described in chapter 3. A TOM type for an absolute URL is defined as follows:

- Like all types, the type has a **name**. Its name in TOM is `s:url`. (Chapter 6 discusses how names are assigned to types.)

- The type has a **supertype**, which is the generic *reference* type, named `e:ref`. The generic *reference* type has a method called `fetch` that returns the object the reference refers to, or an exception if the reference cannot be dereferenced. The generic reference type attaches no special semantics to this dereferencing. In particular, it does not specify whether a fetch will always succeed, how an object is retrieved, whether different invocations of `fetch` retrieve the same object, or what the retrieved object will be. The semantics of the URL type specialize the minimal semantics of the generic reference type.

- The type has an **encoding** named `standard` which encodes the URL as a *ByteSeqObj*: specifically, as a Latin-1 text string consistent with the standard grammar for URLs. It also has another encoding, named `strict`, that restricts the character set to a small, portable subset of ASCII characters, encoding any other necessary characters through a percent-escape convention defined for URLs in RFC 1738 [BLMM94].

- The type has one **method** relevant to this example. This method, called `fetch`, either retrieves the object that this URL points to, or returns an exception if the object cannot be retrieved. It is a specialization of the `fetch` method of `e:ref`, its supertype. This method's return type, like that of the supertype method, is the generic object type (`e:obj`), since one cannot predict what kind of object a URL might point to. We can see that the semantics of `s:url`'s fetch method imply the semantics of `e:ref`'s `fetch`, confirming that `s:url` is correctly considered a subtype of `e:ref`. A program can therefore dereference a URL in the same way that it dereferences any other subtype of `e:ref`, by invoking the `fetch` method.

- The type has several **attributes:**

  - The `protocol` attribute is a text string (type `e:text`), as are all the attributes listed here with the exception of `port`. The protocol attribute indicates the technique that should be used to dereference the URL. Example values are "http", ftp", and "news". (We could declare that the legal values of this attribute are limited to the URL protocols that appear in Internet RFCs at this time, but we will refrain from doing so for this example. Omitting such a declaration allows this TOM type to be used for future protocols that might arise.)
  - The `host` attribute is another text string, indicating the name or IP address of the host to be contacted for resolving this URL. If no such host is specified, as is the case for certain kinds of URLs, the value is undefined (meaning that this type does not prescribe or guarantee any particular value for the attribute, and fetches of the attribute at different times may yield different values). For example, if the program has already determined that the `protocol` attribute is "news", a protocol that does not specify a particular server, a client can ignore the `host` attribute. However, if client programs are not always this smart, it makes sense to define this attribute differently. For instance, one might specify that this attribute value should be an empty string when the URL does not specify a host.
  - The `port` attribute is an integer indicating the port number to use when contacting a host. Some URLs do not specify a port; in such cases, the value is undefined. By the rules given for attributes earlier, a fetch of this attribute in such cases may yield any integer value, or no value at all.
  - The `username` and `password` are text attributes specifying a username and password to use when connecting to a host to resolve a URL. These attribute values are defined for certain URLs, in particular, some FTP URLs. If a password is not required, the password is defined to be an empty string. Both values are text strings.
  - The `path` attribute specifies the instructions to resolve the URL from the appropriate server. It may include both a directory path and a query. (Basically, it includes the

remainder of the URL, as normally encoded, up to the # sign, or to the end if there is no # sign.)

  - Finally, the `fragment` attribute, if present, is a text string that specifies a portion of the object that is being referred to by this URL. If the URL is meant to refer to the entire object, the value of this attribute is an empty string.

- The URL type also has **conversions** associated with it. Some of these are between different formats of the same type. For instance, we noted that URLs have both a `standard` and a `strict` encoding. There exists a conversion from the `standard` encoding to the `strict` encoding. (The conversion from `strict` to `standard` also exists, and is trivial.) Both conversions have `s:url` as their intersubstitutable type, since these conversions do not change any of the information in a URL, but only the representation. Conversions to other types are also possible: for example, some URLs are convertible to objects of the *Gopher menu item* type.

4.10.2 Example 2: Sequences

Many forms of data on the Net represent sequences of objects. Sequences come in many forms. Archive files include a sequence of smaller files, mail spools include a sequence of letters, directory listings include a sequence of file names, and even ordinary files are sequences of bytes. It is useful to have a generic type for sequences, since clients can use it to access objects in any of these sequences through the interface defined in the generic sequence type.

In TOM, we can define a sequence type as follows:

- The type has a **name**, `c:seq` in this case.

- The generic *sequence* type does not need any **encodings**, since we expect that most sequence objects will belong to a more specific subtype of the generic *sequence* type (such as *mail spool* or *directory listing*.) However, it is possible to define an encoding even in the generic case, such as a concatenation of shipped objects for every element in the sequence.

- The type has an **attribute** called `length`. This is an integer that indicates the number of objects in the sequence.

- The type has a **method** called `elem`, which, given an integer argument n, returns the nth object in the sequence.

This information suffices to define the basic sequence type. Subtypes of this type, such as *tar file*, redeclare `length` and `elem`. Those methods are then implemented in whatever manner is appropriate for the subtype. Clients navigating through objects of these types, or other subtypes of the sequence type, can use the interface defined here.

4.10.3 Example 3: Microsoft Word documents

The two previous examples involved types that could be fully specified. But it is also possible to define TOM types for data for which a full specification is not known. In this example, I explain how a type is defined for Microsoft Word documents.

Microsoft does not publish the specification for the Microsoft Word format. Furthermore, the Word format and its capabilities change with each major release of Microsoft Word. One might argue, then, that little or nothing can be said that is guaranteed to stay consistent for all Word formats, past, present, and future.

At the same time, many documents on the Net are simply identified as Microsoft Word documents, with no further information. Furthermore, a nontrivial set of operations can be carried out on these documents. For instance, they can be converted to other formats, such as plain text, HTML, or Postscript. (Such conversions are among the most popular features of existing TOM-based services, as we will see in Chapter 7.)

It is therefore useful to have a TOM type for Microsoft Word documents. Such a type can easily be defined, and remain robust over time, if it is defined using only enough information to distinguish the type, and nothing more. We therefore define the *Microsoft Word* type as follows:

- The type is given a **name**, `mime:application/msword`.

- The type has no **supertypes** except for the generic `e:obj` type. It also has no **attributes** or **methods**. In the semantic constraint of the type's values, though, we specify that the values of this type must include the information present in some Microsoft Word document, for some version of Microsoft Word. In the actual definition, one can specify this in simple English, as in the previous sentence, since we allow semantic constraints to be expressed in any notation.

- We can also define a **`standard` encoding**, which encodes this information in one of the standard Word formats defined by Microsoft. (This encoding does assume that different versions of Microsoft Word can be unambiguously distinguished from one another. This assumption has been valid to date, and it would be to Microsoft's advantage to continue to make their formats distinguishable.) We do not have to specify exactly *how* the information in the document is encoded, but simply note that it follows a standard Microsoft format.[8]

- With this information in hand, we can then define **conversions** from Microsoft Word formats to other formats, like plain text, and HTML. The implementation of these conversion routines will change over time, as Microsoft defines additional Word formats, but the basic interface definition can remain the same.

If we want to carry out more specific operations on Word documents, but do not wish to assert that they apply to all Word documents for all time, we can define more specialized Word subtypes. For example, the specification and capabilities of Microsoft Word 5 documents are now fixed (since Microsoft has released Word 5 and gone on to define new formats for later releases.) One can therefore define a *Word 5 document* type, as a subtype of the generic Word type. Such a type can have a variety of operations defined for it, such as counting words, extracting author information, or producing a document with different fonts and spacing. One can also define conversions between this specific type and the more generic Word type. (The upward conversion is trivial; the downward conversion can be as simple as checking whether the document is already in a valid Word 5 format, and raising an exception if it is not.)

One can also define intermediate types between *generic Word document* and *Word 5 document*, such as types that promise that certain specific information (such as font specifications) are included in the document. These types may be useful for controlling information loss in Word conversions. Alternatively, if each successive Word format has all the information of the earlier Word formats,

[8] Note that it is only in the encoding, and not in the basic semantics of the type, that we specify that the data appears in a particular *format*. Making it a condition of the type that the document be in "Microsoft Word format" would confuse abstract interface with concrete representation.

and simply adds extra information, then each new Word format can be a subtype of the previous one. In that case, a good conversion between two different Word formats would have the earlier Word format declared as its intersubstitutable type.

This example shows how TOM can work with documents with types whose full specification may be unknown, or subject to change. An increasingly specific hierarchy of types lets us take advantage of however much we happen to know about a particular kind of document. The conversions and their intersubstitutability properties allow us to move from general to specific types, or vice versa, as needed. Even if there are many types that represent Word documents, a client application need not know about all of them, but can simply use the Word type it knows about.

4.11 Summary

In this chapter, I described the information defined in TOM types, and showed how TOM allows data to be used in a variety of ways: through their native low-level representations (via encodings and formats), through abstract interfaces (via attributes, methods, and subtyping), and through controlled transformations into objects with familiar types and formats (via conversions, with the help of intersubstitutability declarations). The next chapter will describe the system that puts this type information to work.

Chapter 5

Type Brokers

In the last chapter, I defined the data abstractions that TOM uses to describe information in a distributed system. This chapter, in turn, describes the infrastructure that allows programs to take advantage of these abstractions. The key elements in the infrastructure are mediator programs that inform clients about data types, find appropriate servers to perform services on objects, and coordinate the conversion of data when needed. These mediator programs are known as *type brokers*.

This chapter shows how type brokers can be used to access attributes and call methods on objects, to convert information from unfamiliar formats to familiar ones, and to use information and services from systems not designed with TOM in mind. We will also see how ordinary application programs can use the services TOM provides.

I begin by giving a brief overview of TOM's architecture, and the central role that type brokers play. I then describe in detail each of the major roles type brokers play, and how they support the constructs defined in Chapter 4. This description will include detailed discussion of how type brokers operate, information on the type descriptions that type brokers maintain, and details on how type brokers use these descriptions to plan conversion strategies. I will also show how type brokers and other TOM agents communicate.

5.1 Overview of TOM's component architecture

The basic architectural model for TOM consists of three kinds of components: clients, servers, and mediators called *type brokers*. Servers provide data, and perform operations on the data, such as conversions, method executions, and attribute fetches. Clients retrieve data, and request operations on the data. Type brokers take client requests, and find servers that return data and operation results that clients seek. Type brokers also maintain information about types and formats. This information is represented as objects that can be examined, operated on, and converted just like any other TOM object. TOM components (clients, servers, and brokers) are also known generically as *agents*.

It is possible for a TOM agent to play multiple roles. For example, an agent might act as a server for certain kinds of operations, but when computing those operations it might be a client of other agents that do intermediate computations on its behalf. Likewise, a type broker can also act as a server or as a client for certain operations.

Figure 5.1 shows the basic architecture of TOM. The architecture diagram shown here is similar to the generic "layered" architecture shown in Chapter 3, except that the repository layer is not in the scope of TOM's architecture.

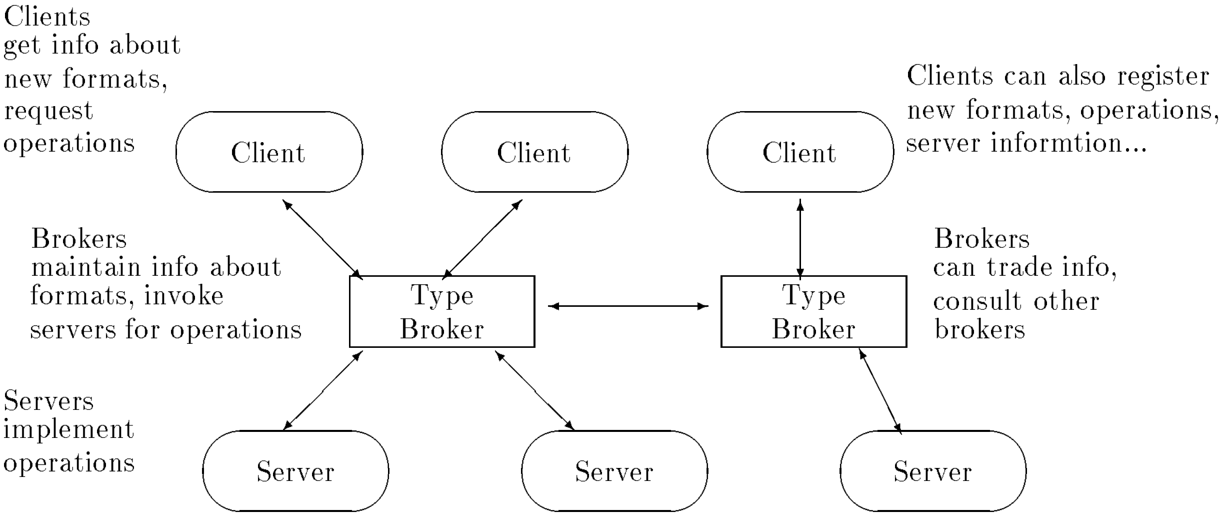


Figure 5.1: The basic components of TOM

The key elements of the architecture are the mediating type brokers. They have three main functions:

1. They act as mediators for operation requests by clients, invoking servers that implement the operations.
2. They maintain information about data types, formats, operations, and servers that implement operations, and can provide this information to other agents.
3. They allow new information about data types to be registered, and they propagate this information to other type brokers, allowing TOM's resources and expertise to scale up gracefully.

In the remainder of this chapter, I describe each of these functions in detail. After that, I describe the protocol that TOM-aware agents use to communicate, and conclude with an explanation of how TOM agents can work with servers and clients not designed with TOM in mind.

5.2 Invoking operations on TOM objects

A key task of type brokers is to invoke operations on behalf of a client. Operations that type brokers support include attribute fetches, method invocations, and conversions.

5.2.1 Basic invocation: rationale and description

In traditional object-oriented systems, the run-time system transparently handles the details of operation invocation. When a client calls a method (or any other kind of operation) on an object, the run-time system looks up the location of the procedure implementing the method in a dispatch table. The inputs (or pointers to the inputs) are then passed to the procedure, and the thread of program control branches to the procedure. The calling routine, the method implementation, and the data passed back and forth all typically operate in the same address space, using a common programming language, data representation, and run-time system.

The distributed environment in which TOM operates is radically different from the traditional object-oriented environment, and requires different mechanisms. In a distributed environment, callers of methods and other operations are often application programs that run on completely different network sites from the implementations of the operations. The data to be operated on may also come from anywhere in the network, and may be structured in a variety of forms. The code that implements an operation may require an object to be in a different form than that used by the code that calls the operation. Clearly, the traditional object-oriented execution model described above is not sufficient for this environment.

TOM's type brokers handle many of the extra details involved in invoking operations in a distributed environment. To call a method, for instance, a client sends a request to a type broker. The request specifies the method to be invoked, the object on which the method should be invoked (or a reference to it), and any additional arguments required by the method. The additional arguments can either be passed directly, or by reference, using reference objects.

When a type broker receives the request, it finds a server that implements the method and forwards the request to that server. The set of servers implementing a given operation may vary over time, and different brokers may know about different sets of servers, but this is not a problem as long as all implementations satisfy the specification of the operation. The broker ensures that the input and output data of a method appear in forms that the client and server recognize. The server selected by the broker then executes the method, and returns the result (or a reference to the result) to the type broker. The broker finally returns the result to the program that requested the method. Fetching an attribute or performing a conversion is carried out in much the same way as calling a method.

The requests, the replies, and their accompanying data are communicated via a *protocol*, the Typed Object Protocol (TOP). Many type brokers operate simultaneously[1] on the network, each handling requests for a particular set of clients or services. The type brokers share information about types, operations, and servers with each other, so that over time they provide referrals to an ever-growing variety of services. Client programs can thus take advantage of a wide variety of services distributed throughout the larger network, thanks to the type brokers' ever-growing knowledge base.

5.2.2 Variations on basic invocation

While the procedure described above makes it possible for operations to be invoked in a distributed system, it does not by itself solve all of the problems present in a heterogeneous, distributed environment. There are four key problems yet to be addressed. First of all, data on the Internet may appear in multiple forms, and the data formats used by one agent may not be the same as the data format used by another. Second, data on the Internet often comes from "legacy systems". Such data is not served by TOM-aware agents, and is not specifically packaged or tagged as TOM objects. It may not include explicit metadata identifying its type or format, like TOM's shipped objects do. Third, some programs may require a more efficient protocol than type brokers allow. Fourth, a client may be unwilling to release data to outside parties, due to security or privacy concerns.

TOM therefore allows four kinds of variations and elaborations on the above approach to address

[1] At present, there are two type brokers usually in operation at Carnegie Mellon. TOM allows any number of brokers to run in parallel, however.

these problems. TOM also allows data and services that were not designed with TOM in mind to be used through TOM's facilities.

The variations on the basic procedure above include the following:

1. **Bypassing the broker.** If the client already knows of a server that implements an operation, it can contact the server directly (using TOP), instead of going through a type broker. This option can speed up a method call by eliminating the mediator, and also gives a client more control over where an object is sent. Type brokers can also, instead of contacting a server themselves to carry out an operation, tell a client what server the client should contact. This latter variation, while not implemented on current type brokers, may help reduce the load on a heavily-used broker, and also reduce further mediation overhead.

2. **Converting the data.** The client or the server can request that the data be *converted* to another format either before or after an operation is invoked. If the conversion is intersubstitutable with respect to the data type in the operation's signature (that is, no essential information is lost or changed), servers will compute correct results using the converted data, even if the server and client use different data representations. (Intersubstitutability was described in detail in Chapter 4.) Since conversion itself is an operation, conversions can be carried out via a type broker using the basic invocation procedure described above, or using any of the variations described here.

3. **Directly operating on the data.** A TOM client can also examine the type and format designation given to a particular object, to see if the object is in a format compatible with its own operation implementations. If the format is compatible, the client can operate directly on the bytes that encode the object's value. If not, it can have the object converted into a familiar format (as above) and then operate directly on the resulting object. This approach allows familiar operations to execute quickly, since there is little or no overhead introduced by remote calls.

4. **Sending instructions for operating on the data.** A fourth possibility, mostly unimplemented in current TOM agents, is for a type broker to send instructions on how to carry out an operation to a client, rather than the client sending the data to a broker for computation on a remote server. These instructions might be in the form of code (such as a Java routine) or other information sufficient to tell the client what to do. One type of instruction supported in current TOM type brokers is a specification that a particular conversion is *trivial* (in the sense defined in the previous chapter). This information allows a client to convert a shipped object by retagging its value with the output format of the conversion.

   Informing the client how to carry out an operation can be more efficient than having the operation carried out on a remote server, if it takes less work to send code to data than to send data to code, and if the client has sufficient computing resources. This approach may also be advisable when confidentiality is important, since the client would not send object data to any other agent. On the other hand, there are a number of well-known security concerns involved in executing code received over the network. Such code would therefore need either to be executed in an environment that prevents them from accessing any sensitive resources, or would need to be certified by a trusted source. Java, SafeTCL, and Perl with its "Safe" module [WCS96], are current examples of languages that support execution in a securely sealed environment. While current implementation of TOM do not support sending Java or other portable code to clients, it would not be difficult to add this facility to the implementation of TOM types and brokers.

We can compare and contrast these approaches with an example. Suppose a client program needs to extract text from an image via optical character recognition. Suppose there is some `OCR` method on the image type that will accomplish this. The client can send a request to a broker to execute that method (the default approach). Or, it can contact a server that implements that method directly (alternative 1). This would cut down on the communication required for the routine, by cutting out the "middleman." Contacting a server directly would also ensure that no other server is sent the image, which may be useful if the image includes confidential information. If the implementation of the `OCR` method assumes a GIF format, and the image is in JPEG format, the data may need to be converted from GIF to JPEG first (alternative 2). Or, the client itself may implement an appropriate OCR algorithm, and operate on the image directly (alternative 3), without sending it to anyone else. Finally, if a Java routine to recognize text is available, a type broker that maintained implementation code might send the routine to the client so that the client could execute it on the image directly (alternative 4). With these options, TOM's operation invocation over the network can be efficient, secure, and compatible with a wide range of data formats.

5.2.3 Server implementations

A server implementing an operation is free to implement it in any way that is consistent with the operation's specification. It needs to parse the request and deliver the reply using TOP. The prototype Unix-based server implemented for this thesis can execute operations by invoking programs in any language using a simple interface similar to the standard CGI interface [Nat97] used by World Wide Web servers. A configuration file for scripts specifies the program to run to implement a particular operation, and the formats that should be used for the inputs and outputs to the program. The server gives the inputs to the specified programs via standard input and command-line arguments, and then ships the standard output of the programs back to the client, tagged with the specified return format. If the inputs are not already in the formats specified in the configuration file, the server converts the inputs to the appropriate formats before running the program.

Figure 5.2 shows three sample entries in a server configuration file, all describing operations on GIF images. The entries for each operation tell the broker what program to run or what script to call, and how the inputs and outputs are formatted. The first operation implementation shown is a conversion named `giftoppm`, from the `standard` encoding of GIF images to the `any` encoding of PPM (portable pixel map) images. It runs the program `/usr/local/bin/giftopnm` to perform the conversion. The second operation implementation is for a method called `turn90left`, which, given a standard GIF image, returns the same image rotated 90 degrees, by running the specified script. The third operation implementation fetches the `height` attribute from a GIF image, running a Perl script to calculate the height, and returning the result as an integer encoded as an ASCII string.

5.2.4 Error recovery: when invocations fail

Occasionally, an operation invocation does not return an object. There are several reasons an operation may fail. The input values might not be available, or might not be convertable into a form that an implementation can use. The operation may return an exception instead of a return value. Or a server may be unavailable, crash, or drop off the network in the middle of executing an operation.

TOM's type brokers are designed to handle failures gracefully. TOP specifies different replies for servers to return depending on the reasons for failure. Type brokers interpret these replies to

```
CNVT giftoppm
EXPECT mime:image/gif standard
RETURN net:ppm121494@gs1.sp.cs.cmu.edu any
CMD /usr/local/bin/giftopnm

OPER turn90left
EXPECT mime:image/gif standard
RETURN mime:image/gif standard
SCRIPT
giftopnm | pnmflip -rotate90 | ppmtogif
.

ATTR height
EXPECT mime:image/gif standard
RETURN e:int ascii-rep
SCRIPT
#!/usr/local/bin/perl
read(STDIN, $buf, 10);
@vals = unpack("x8 C2", $buf);
print $vals[0] + $vals[1] * 256;
# consume the rest of the file, to make parent happy
while (<STDIN>) { };
.
```

Figure 5.2: Part of the "scripts" configuration file for a type broker at Carnegie Mellon.

compensate, if possible, for different kinds of failure. For instance, if a broker contacts a server to carry out an operation, and that operation fails due to implementation limitations, server failures, or network problems, the broker will try to find another server to carry out the operation. If, however, the failure is an exception based on invalid input values, the broker will give up on that operation. If the broker invoked the operation as part of a multi-step plan, such as a complex conversion, the broker will try to find an alternate plan.

5.3 Maintaining information about types

Type brokers maintain information about data types, formats, operations, and servers that implement the operations. This information allows them to find appropriate servers to execute operations, as described in the previous section. This information can also be passed along to clients and to other type brokers.

Type information is represented in the form of objects, which, like any other TOM object, can be can be examined, operated on, and converted. A *type description object* contains all the information used to define a type (as described in Chapter 4), information on servers that implement operations on the type, and other metadata concerning the type and its formats.

Type brokers typically store information about many related types. The complete collection of type description objects is called the broker's *type graph*, since the overall collection can be modeled as a graph with nodes consisting of type description objects, and edges consisting of relations between types, such as subtyping, encoding, and conversion relations. Conceptually, the type graphs of individual brokers can combine to form a global type graph for TOM as a whole. Type brokers can take advantage of their type graph to carry out complex tasks. For example, if a broker receives a request to convert an object from format A to format B, it can invoke a direct conversion if one is available, and if not, search its graph for a chain of conversions starting at format A and ending at format B.

5.3.1 Type description objects

A type description object contains the following information:

- Names of the type.
- The names of supertypes.
- Encoding specifications for the type. This includes the name of the encoding, the name of the encoding type, and an optional specification of how the information is encoded.
- A specification of the type's overall semantics, detailing what the type does or describes. As I noted in the previous chapter, this semantic specification can be made in any notation. However, it should the notation should be one that does not overspecify behavior. Implementation code, for example, usually implies more constraints on the type's behavior than actually warranted, so it is generally not appropriate in semantic specifications.
- The definition of methods and attributes of the type, including their names, signatures and specification.
- Conversions from formats of the type. This includes the conversion name, the source and destination formats, and an optional semantic specification. The specification can include an intersubstitutable type for the conversion, and can also note whether the conversion is `trivial`, as defined in the previous chapter.
- Administrative information about the type, such as its creator and its publication status.
- For each method, attribute, or conversion, a list of agents that implement the operation. When a broker receives a request for the operation, it can contact any of the agents on this list to carry it out. All else being equal, agents listed earlier on the list will be tried before agents listed later. (However, if a type broker is also a server implementing a particular operation, it may try to execute the operation locally before contacting any other agents.)

Type brokers also record the date and time a type description object was registered or updated. This information is not part of the type description object itself, but is associated with the object in the broker's internal database. This information enables a broker to inform clients of newly registered or changed type descriptions.

With this information, type brokers are able to supply type definitions to any client. They can identify the encodings and formats available for a type, give details on the interface and conversions available for a type, and contact an appropriate server to carry out an operation requested by a client. They can also inform other brokers of new type information, as described later in this chapter.

Figure 5.3 shows part of a type description object for the `s:url` type, one of the example types from Chapter 4, with some of the attributes deleted to save space in the figure. At the top of the figure, we see that the type is named `s:url`, that it has `e:ref` as a supertype, and that it is meant to model URLs in accordance with Internet standards. The `STATUS X` line is an administrative line indicating its experimental status. (Status of types will be discussed in the next chapter.) Two attributes of the type are shown here, with their names, the types of their return values, and an informal statement of their semantics. Following this is the `fetch` method, shown to be a specialization of `e:ref`'s fetch method and to return an object. After that come specifications for two encodings, `standard` and `strict`, both of which encode a URL object as text. Finally,

```
NAME s:url
STATUS X
SUPER e:ref
SEM
"This is a type for absolute URLs, as defined in RFC 1738 et al."

ATTR e:text protocol {
SEM
"A string indicating the protocol that should be used (e.g. 'http',
'gopher', 'ftp'"
}

ATTR e:text user {
SEM
"A string indicating the username to use for connecting to a server.  Not
 defined for most URLs (but permitted for some, like telnet and ftp).
 Escape sequences may appear (e.g. %20 for space)"
}

OPER e:obj fetch {
SUPER e:ref
SEM
"Returns the object referred to by the URL.  This routine may assign a
type to the object based on information in the URL itself or obtained
while fetching the object; or it may simply return it as an uninterpreted
byte sequence."
}

ENC e:text standard {
SEM
"The standard character representation of URLs specified in the RFCs,
 but allowing 'unsafe' ASCII characters to be represented literally (as they
 often are in practice)"
}

ENC e:text strict {
SEM
"The standard character representation of URLs specified in the RFCs.
 All characters not exempted in RFC1738's rules must be represented with
 the percent-escape convention."
}

CNVT stricttostd {
TYPE s:url
ENC e:text strict
TYPE s:url
ENC e:text standard
TOP s:url
TRIVIAL
}

CNVT stdtostrict {
TYPE s:url
ENC e:text standard
TYPE s:url
ENC e:text strict
TOP s:url
AGENT {
HOST tom.cs.cmu.edu
PORT 4617
}
}
```

Figure 5.3: The standard encoding of the type description object for Uniform Resource Locators, `s:url`.

the figure shows the description of two conversions, one going from the `strict` encoding to the `standard` encoding, and the other going the other way. The `TOP` line on these conversions indicates that `s:url` is an intersubstitutable type for both of these conversions. The `stricttostd` conversion is marked as `TRIVIAL`, meaning that it can be implemented simply by relabeling values with the new format. The `stdtostrict` conversion description lists one server that implements the operation, on port 4617 of `tom.cs.cmu.edu`.

5.3.2 Reasoning with the type graph

Type brokers are more than just query and referral services. Their database of type descriptions and relations allows them to find non-obvious solutions to problems, and to plan and execute complex tasks.

Finding operation implementations

Knowledge of the type graph can help brokers find alternate operations to invoke when the operation specifically requested is not available. For example, if an operation on type T is requested, and no implementations are available that can operate directly on objects of type T, a type broker might be able to find a suitable implementation in a subtype of T that will accept the input arguments. Recalling the generic sequence example from Chapter 4, for instance, if a client tries to fetch a sequence element using the `elem` method, and no implementation is available for the type of sequence provided by the client, a broker can look for any other subtype of "sequence" for which an implementation is available. The object can then be converted to the appropriate new subtype (using any conversion for which the generic sequence is an intersubstitutable type) and the method carried out.

Conversion searches

One crucial task often requiring extensive type information is to plan the conversion of an object from one format to another. Conversion may be specifically requested by a user, or it may be required by a server that implements an operation, in order to get an input argument into a format the implementation can work with.

Often a conversion request does not specify a particular conversion routine to use. Instead, it requests the conversion of a particular object to any of one or more desired formats, through whatever means are available. The client may also require certain information to be preserved in the conversion, by specifying an intersubstitutable type for the conversion. Even if an intersubstitutable type is not explicitly mentioned, clients generally prefer conversions that do not lose information unnecessarily, and do not require excessive time.

Type brokers find plans for these conversions through graph search, where the nodes of the graph are formats known to the broker and the edges of the graph are known conversions. If the client specifies an intersubstitutable type, only edges whose conversions have that type (or a subtype) as an intersubstitutable type are considered. Once a broker finds a suitable path through the graph, it invokes the individual conversions in the path to yield the final answer.

Different brokers may choose to search the graph in different ways, but should find a path satisfying the client's request if one exists in the broker's type graph. The prototype type brokers implemented for this thesis use bounded breadth-first search to find the shortest known path from the starting format to one of the requested formats, ignoring conversions that do not meet the client's intersubstitutability requirements. If there are two paths of the same length to different formats, the brokers will prefer the path leading to a format that appears earlier in the list of desired formats requested by the client. In this way, clients can express a preference for one format over another.

Breadth-first search has three important strengths. Short paths are likely to have shorter execution times than long paths. Conversion steps may also have unspecified amounts of information loss; if so, a small number of conversion steps is likely to minimize this information loss. Breadth-first searches also have well-defined run-time bounds to find the shortest path, and to search a subgraph given a maximum path length. In practice, I have found that successful conversion paths tend to be short, so cutting off the search after half a dozen steps generally will not rule out viable conversion strategies. (Chapter 7 discusses experiences with conversion services in more detail; in the case studies described there, nearly all viable conversions were handled in two steps or less.)

Type brokers must also be able to make alternate plans if a conversion fails. Failure in a complex conversion plan can occur for many reasons, but two important categories are worth distinguishing. The first category is operation failure, when a particular conversion is inherently not possible for a given object. For example, a conversion might be well-defined for only *some* of the objects of a particular format, and not the object at hand. The second category is server failure, where a conversion that could in theory be carried out cannot be executed due to the lack of working, reachable servers. Server failure is to blame, for instance, if the only agents known to implement a particular conversion are unreachable, buggy, or overloaded.

Different strategies are called for to recover from the two kinds of failure. After an operation failure, trying the same conversion on the same object using a different server is likely to be fruitless, but that might be a useful course of action if the problem was simply a server failure. On the other hand, if a server failure occurs for all servers known to implement a particular conversion, it is probably fruitless to try the same conversion on other objects.

The brokers that are currently implemented continue to go through a bounded breadth-first search after a failure, trying the next shortest path they find. Intermediate results of previous conversions are saved, so that old subpaths recurring in the new path do not have to be recalculated. The new search uses the experience of the old strategy to discard conversion edges known not to work. If there was a server failure for all agents implementing a particular conversion, the conversion is not tried again. If the failure was an operational one, such as might occur with an object not in the domain of a partial conversion function, the conversion can be be tried again for a different input, but not for the same input that failed before.

There are several possible enhancements to this basic conversion search strategy. For example, it may be possible to further speed up conversion searches by caching conversion paths found for a particular request, and reusing them when the same kind of request comes in later. However, the time spent in conversion searches is currently low enough that I have not found it worthwhile to implement such a cache for the prototype type brokers. (Such a cache would also require its own maintenance overhead, which might minimize its benefits.)

It is also possible in theory to weight conversions, so that some conversions are preferred over others depending on criteria specified by the client. Brokers could then search their type graph using weighted breadth-first search. I have not yet seen compelling reasons to include such weights in TOM, and current type broker implementations do not support them. However, weights could be supported in future subtypes of the current *conversion description* type, and would still be backward-compatible with the current unweighted conversion descriptions due to substitutability.

5.4 Registering and propagating type description objects

Type brokers accept registrations of new type information, such as new types, encodings, operations, or information about agents that implement a particular operation. Once registered on one type broker, type information can then propagate to other type brokers, until it has spread throughout the network.

To register a new type, its creator registers a type description object with a type broker. When registering, the creator gives the definition of the type, and may also include information about encodings, agents, and other aspects of the type description. Brokers also allow a registrant to claim "ownership" over the type, which reserves rights to make certain major changes to the type description.

Type brokers are configurable to either accept new type information automatically, or forward it to a broker maintainer for manual review. Current implementations of type brokers have configuration files where a maintainer can specify which kinds of registrations will be automatically accepted, which will be automatically rejected, and which should be forwarded for manual review. Broker maintainers may configure their brokers in different ways, depending on how concerned they are about upholding the quality of their type graph and how much time they have to review type descriptions.

Once a type description object is registered, it is still possible to augment or add to it in certain ways. For example, more encodings or conversions can be added, and information about agents implementing a particular operation can be added or removed. However, users should normally be prevented from adding information that would fundamentally redefine or constrain the type in a way that would make it inconsistent with existing definitions. The next chapter will discuss these issues in more detail.

Once a type description object is registered with a broker, it becomes visible to any other agent querying the broker. TOP includes commands to get the full type description object of any type known to a broker. TOP also includes commands to retrieve the names of all types known to a broker, or the type names of all type description objects that have been added or changed in some way since a particular date.

These TOP commands make it possible for a type broker to learn everything about types that a peer broker knows. All it has to do is ask a peer broker for all the type description objects that have been changed since the last time it queried the peer, and then register the new information in its own database (possibly sending some of it to a broker maintainer for review first). If the client broker records the time it started the query, it can then repeat the process at a later date to update its own database.[2] This update mechanism is similar to that used by Usenet news servers to propagate newsgroups [KL86]. If the network of such peer relations is transitively closed, and covers all public type brokers, information registered with one public type broker will eventually propagate to all the others.

5.5 TOP: The communication protocol for TOM

TOM components communicate using a protocol specially designed for TOM called the Typed Object Protocol (TOP). TOP is a stream-oriented, request-response protocol that lets clients invoke operations, query for type information, register new type information, and make other requests that

[2] I have not yet implemented automatic client polling for type information, which would allow new type information to move between brokers with no human intervention as described. But current type broker implementations do fully support the TOP commands that supply this information.

Command	**Function**	**Common parameters**
Invoking operations on TOM objects		
OPER	Call a method	Name and type of method; object; arguments (passed by value or by reference); preferred format(s) for return value
ATTR	Fetch an attribute	As above, but without arguments
CNVT	Convert an object	Name and type of conversion, and/or preferred format(s) for return value; intersubstitutable type
Maintaining information about types		
TYPQ	Get information about a type	Name of the type
TYPL	List types that have been registered or changed recently	Time span of interest
REGI	Register new type information	Description object for type, encoding, name, supertype, operation, or agent
Managing interactions		
PROTO	Negotiate protocol version	Version to use
AUTH	Request authorization	Authentication information
NOOP	Synchronize	Acknowledgement pattern
QUIT	Terminate a session	None

Table 5.1: Requests supported by TOP

aid in these tasks. In this section, I will describe some of the key features of TOP, and show how the protocol is well-suited for distributed object systems like TOM.

TOP supports ten kinds of requests, which are summarized in table 5.1, and described in full detail in Appendix A. Many clients need not use all of these requests. For instance, some may only use the three operation invocation requests (ATTR, OPER, and CNVT), along with TYPQ and QUIT.

The design of TOP has four main requirements. First, it has to be expressive enough to support the constructs and services defined in TOM and provided by type brokers and servers. Second, it has to be simple enough to be easily and compactly implemented. Third, it has to be reasonably efficient, comparable to similar protocols in widespread use on the Internet. Finally, TOP has to be flexible enough so that TOP-speaking agents can interoperate smoothly with agents that incorporate extensions of TOP, or even entirely different protocols.

5.5.1 Expressiveness

TOP's basic request vocabulary is expressive enough to support the functions of type brokers, and is further enhanced with the objects and metadata provided by TOM. Basic TOP requests provide all the elementary requests required for clients to interact with servers and brokers, and for brokers to acquire and communicate type information. As shown in table 5.1, TOP includes the following types of operations:

- Carrying out operations on objects, such as attribute fetching, method invocation, and conversions.
- Getting information about types known to a broker, and retrieving their descriptions.
- Registering new type information with a broker.
- Negotiating the ground rules for the interactions above. Under this category are commands to negotiate the protocol used to communicate, to synchronize between client and server, and to authenticate a client to be authorized to carry out certain privileged operations. The authentication features, while not developed at length in this thesis, have potential use in security and electronic commerce.

The object model itself can be used to further enhance the expressiveness of TOP, if needed. For example, TOP itself supports only elementary queries about types and a broker's type graph. But it is also possible to model the broker's type graph as an object.[3] Operations can then be defined on this object that allow more complex queries. For example, the *type-graph* type could have a method that lists all known subtypes of a particular type, or that returns a list of conversion steps that could be used to convert between two specified formats. Clients could then retrieve this information by invoking these methods on the broker's type graph. They would not need to know the details of how this graph was implemented.

TOP also supports tagging objects with metadata. Recent research in information systems has highlighted the importance of metadata associated with information. Michael Schwartz [SEKN92] has surveyed several Internet information systems that use metadata. The Stanford Digital Library [BCGP97] is an example of the extended use of metadata in one particular application domain. Metadata can be used to annotate as well as organize data, and can take many forms. For example, in one system, it might be appropriate to annotate information with a citation of its source, so that its reliability can be assessed. Metadata describing when data was last updated might be important in a caching information system or an information system that relies on the latest information.

TOP allows arbitrary metadata about objects to be passed between agents along with the objects. This metadata is itself passed in the form of TOM objects. Metadata can also have its own metadata. TOM itself does not specify anything about how this metadata should be used. However, the inclusion of metadata in the basic TOP protocol enables agents that need this extra information to pass it along in a manner completely compatible with basic TOM agents.

5.5.2 Simplicity

TOP is designed to be relatively simple to speak and understand. We see this in three ways: First of all, the full protocol is small enough that it can be implemented and recognized in a small amount of code. The protocol parsing code in currently implemented type brokers uses less than 1000 lines of C. As measured by the number of commands supported, and the length of the protocol specification, TOP is no more complex than the standard Internet protocols for mail and news. SMTP, the standard Internet mail delivery protocol, defines 14 requests [Pos82], and NNTP, the standard Usenet news protocol, defines 15 [KL86].

[3] Current type broker implementations do not do this.

Second, TOM agents do not have to support the entire protocol. For example, servers that are not type brokers might only recognize the three TOP requests required to carry out operations, and QUIT.

Third, many useful extensions to the protocol can be made on top of the basic commands through TOM's object model (and can thus be backward compatible with agents that handle only the basic commands), as we saw in the *type-graph* example above. Hence, new, incompatible additions to the protocol are not required for these extensions. Thus, client and server handling of TOP can stay simple.

5.5.3 Efficiency

TOP is also designed to be reasonably efficient, comparable to the efficiency of the basic World Wide Web protocol, HTTP/1.0 [BLFF96]. Protocol efficiency can be measured in terms of latency (the amount of time required to complete a particular request) and throughput (the amount of work that can be completed in a particular time period). Exact latency and throughput figures will vary depending on the bandwidth of Net connections, the amount of data involved in a request, and the computation time required for a request. TOP is also designed for a wider variety of tasks than HTTP/1.0 is. However, it is still possible to compare TOP and HTTP, assuming that the two protocols are used in the same environment, and for tasks supported by both protocols (such as document fetching). The protocol used by TOP for document fetching is the same as that used by TOP for other operations, so it is reasonable to generalize the analysis of TOP in this area to TOP in other areas.

When considering document fetches, TOP has an important latency advantage over HTTP/1.0 when fetching multiple documents. HTTP/1.0 requires a new connection for every retrieval of a Web page or image. It takes significant time to establish the underlying TCP/IP connection and reestablish context for every request, so latency in HTTP/1.0 is relatively high. TOP, in contrast, allows a sequence of related requests and replies in a single connection. This "session-oriented" design reduces the number of connections required for multiple fetches, or other requests.

In terms of throughput, TOP and HTTP/1.0 perform roughly equally well for document requests. In a normal document fetch, HTTP/1.0 requests and replies actually use slightly fewer bytes than the equivalent TOP requests. Since document fetching is a fundamental part of HTTP/1.0, and one of many possible operations in TOP, HTTP/1.0 requires fewer bytes to tell a server it wants to fetch a document. Also, because HTTP/1.0 indicates the end of a document by the close of a connection, it does not need to use any quoting or length metadata to delimit a document, as TOP needs to. However, the extra overhead involved in the start of a TOP fetch request consists of only about 40 bytes, and quoting overhead can be kept to less than 1 percent of the total length of a document.

Furthermore, TOP has some extra ways of conserving bandwidth not available in HTTP/1.0. TOP clients can request that a value be returned in a particular format. If this format is a compressed format, a document fetch can require much less bandwidth than a corresponding HTTP/1.0 fetch would. (HTTP/1.0 allows clients to specify preferred MIME types for a return value, but as we saw in Chapter 3, MIME types are inadequate for naming arbitrary compressed formats.) Also, TOP allows arguments and return values to be passed either as complete objects or as references to objects. Passing by reference dramatically reduces the bandwidth required for certain operations.[4]

[4] In traditional programming languages, "call-by-reference" often implies that the operation mutates the referenced object in place. Since TOM does not mutate objects, TOM uses it simply to improve performance by avoiding unnecessary data transmission.

A session-oriented protocol like TOP gives ample opportunity for saving bandwidth through passing references. For example, a client may call a method on a particular object, get back a reference to the result, and then call a method on that result of the previous method. The result itself does not need to be passed back and forth between client and server.

The efficiency of HTTP has improved in recent revisions. In 1997, a specification for a revised HTTP protocol was published as RFC 2068[FGM$^+$97]. HTTP/1.1 adds a number of features that were absent in HTTP/1.0 (and present in TOP), to improve efficiency. Like TOP, it is session-oriented and supports compressed formats. Clients that need early access to particular parts of an object, such as the height and width of a GIF image, may be able to get them more quickly via HTTP/1.1 than via TOP, due to the support for range fetching in HTTP/1.1 (which allows portions of multiple documents to be fetched). While range-fetching reduces latency for critical data, it does make HTTP/1.1 significantly more complicated than HTTP/1.0, and many clients and servers do not yet implement it. TOP remains comparably efficient to the HTTP protocol in use for years, and its support of call-by-reference can still give it an advantage in throughput. There is also room in TOP for further optimization if needed.

The efficiency of TOM as a whole depends not only on the efficiency of TOP itself, but also on the efficiency of the overall pattern of TOP requests required to carry out an operation. As we saw earlier, TOM clients can send requests to a type broker, which then forwards them to appropriate servers. The use of a mediator agent (the type broker in this case) necessarily increases latency, since clients could have requests fulfilled more quickly if they contacted the appropriate servers directly. However, type brokers also allow clients to take advantage of a large number of servers that they might not otherwise know about; hence, the additional functionality that type brokers provide may be worth the extra run-time inefficiency. We also saw earlier that a client can take advantage of more efficient interaction styles, including interacting with servers directly to eliminate mediator overhead.

5.5.4 Interoperability and gateways

TOP directly supports interoperability through protocol negotiation. The PROTO request in TOP allows clients to propose and accept (or reject) variations and extensions of TOP, or even to switch to entirely different protocols. A more common way for TOM agents to interact with agents that use completely different protocols, however, is through *gateways*.

Gateways are mediators between TOM agents and non-TOM agents. They can mediate on the server side or the client side, as shown in Figure 5.4. Server-side gateways receive TOP requests and translate them into requests to non-TOM programs via appropriate protocols or APIs. In Figure 5.4, the server-side gateway translates TOP requests into SQL requests to a database. Client-side gateways receive requests from non-TOM agents through some well-known interface and translate them into TOM requests. In Figure 5.4, the client-side gateway receives HTTP requests from a Web browser and translates them to TOP requests. Gateways also append or strip away TOM metadata, as necessary, to the data they transmit. From the perspective of a type broker, client-side gateways and server-side gateways appear no different from ordinary clients and servers. Chapter 7 includes further examples of gateways used in the thesis case studies.

5.6 Summary

In this chapter, I have described the computational infrastructure that allows clients to take advantage of a wide variety of data formats, and also disseminate information about new data formats.

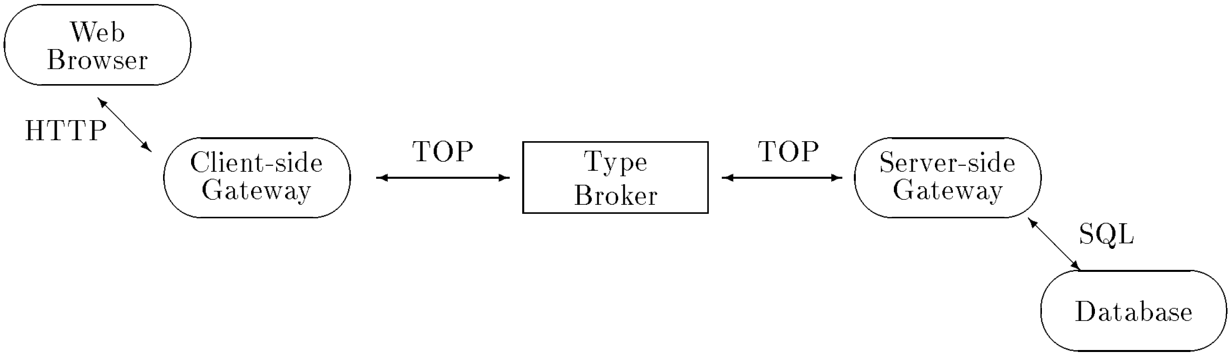


Figure 5.4: An example of gateways in TOM

In particular, I have shown how type brokers maintain constructs to describe types and formats as defined in the previous chapter. I have shown how brokers mediate between clients and servers to carry out operations on data, and how brokers plan complex conversions from one format to another. Type brokers thus allow clients to use a variety of object formats through interfaces and conversion.

The services that type brokers provide fulfill the expressiveness requirements described in Chapter 1. Specifically, brokers describe a wide range of data through their type description objects; brokers can be queried to find out what can be done with a given piece of data; and brokers support a wide variety of execution models through the techniques and variations for invoking operations described in this chapter.

Type brokers also support the data and computational heterogeneity requirements specified in Chapter 1, through their abilities to convert between different data formats, and to refer clients to a servers that can operate on different kinds of data. The techniques and variations described in section 5.2 for computing operations, the ability to configure servers to run arbitrary scripts, and the client-side and server-side gateways between TOM and other systems all provide rich support for computational heterogeneity and composability.

The components of TOM support incremental integration as well. Agents can communicate in TOP with a relatively small investment in code, and existing protocols are supported through object implementations (such as the support for HTTP through the URL type) and through gateways (such as the script interface provided by servers, and the gateways to and from TOM).

I also demonstrated the workability of the system, by showing that TOP's efficiency is roughly equivalent to other commonly used Internet protocols, and by showing that type brokers can handle new types and services through augmenting their database of type description objects. The ability of type brokers to update their type and agent information also shows that the system accommodates evolution in the universe of types, formats, and agents. In the chapters to come, I will turn to the remaining issue of scalability, and also further verify workability through case studies.

Chapter 6

Scaling Up: The Growth of the Type Graph

We have now examined the data model and computational infrastructure for TOM. We have seen how types are defined, and how type brokers manage type information and give access to type operations. But in order to reap the full advantages of a distributed type system, the universe of known types must be able to grow in a decentralized fashion. Expandable type systems that depend on a central registry, such as MIME's, do not keep up with the development of new types and services that occurs constantly on the Internet. It is much better to allow *anyone* to add new types and services on *any* type broker, so that the type graph can grow in many directions simultaneously. While decentralized growth can produce a large and powerful type graph quickly, it also raises questions about order and scalability. New type information must not be allowed to invalidate previous type guarantees, or make the repertoire of types chaotic and unmanageable. At the same time, new type information needs to be able to build on old type information, so that clients can continue to use the installed base of data types and services.

In this chapter, I analyze three key growth and scalability issues for distributed type graphs. First I discuss how new types are added to the graph, treating issues of global naming, coexistence with old types, and propagation. Second, I show how existing types can be updated and extended, and how the type graph can be further connected with new subtype relations. Finally, I examine the problem of ensuring correct method dispatch in a type graph where new methods and subtype relations can be added at any time. In each of these cases, I first look at the problem abstractly, and then show what techniques TOM uses to handle these problems, and keeps a complex, ever-growing type graph manageable.

6.1 Adding new types

It is important that new types in a type graph not conflict with existing types, either by being ambiguously named or by semantically interfering with other types. There must also be a way for information about types to be made widely known. In this section, I show how new types can be added gracefully to a distributed type graph, by examining the techniques used by TOM.

6.1.1 Naming a type

The usual way to refer to a type in an object-oriented system is to name it. In conventional object-oriented programs, programmers choose any type name they wish that is not already in use.

Since programmers know all the types used in the program, they can easily detect and avoid name conflicts. In programs that incorporate types from a variety of independently-developed modules, type references are sometimes qualified with the name of the module, to avert type name conflict between modules. This approach works as long as a program does not use two distinct modules with the same name.

In a distributed type system, where anyone can create new types, and no one broker has information on all the types in existence at a given time, name conflict cannot be averted if users are allowed free rein in choosing names for types. To avoid name conflict, a distributed system must impose some control over type naming.

Desirable type naming schemes do more than just ensure uniqueness. Useful names are also easy to copy and remember, so that human users and programmers use the right name in programs, and can easily type or copy the name in messages. It is also desirable for a name to help users find the named entity, or give hints about the nature of the entity. Compactness is also useful, for efficiency and ease of remembering. For example, the name "Dr. Marcus Welby" is short, relatively unambiguous (since there are not very many doctors named Marcus Welby), easy to look up (in an appropriate phone book or medical directory under the last name), and gives some hints about the person's status (namely, that the person is a doctor, probably male, and likely to be related to other people who share the Welby surname). A given naming scheme will balance these desiderata based on the needs of its users. For example, the American system of personal names favors easily remembered names over unique names. In contexts where uniqueness is more important than mnemonics, another personal naming system might be used, such as Social Security numbers.

Different type naming schemes balance these requirements differently. Some schemes concentrate on uniqueness, without much attention to other virtues. Microsoft's COM, for instance, generates a unique 128-bit identifier for new types, using (in part) a machine's IP address and the current time. Such identifiers are both unique and reasonably compact, though they are also not easily copied either by human typists or by ASCII-based messages and protocols, and they give no indication of the function of the type.

Generally, the technique used by most distributed systems to generate unique names is to divide the namespace hierarchically into progressively smaller subspaces, controlled by different naming authorities. Each naming authority has control over a particular subspace that does not overlap with the subspaces of other authorities, except possibly for subordinate authorities. The authority then simply needs to make sure it does not assign the same name in its subspace twice. COM's hierarchy, for instance, uses a different subspace for each Net-connected computer, based on its IP address. Hierarchical namespaces are also used in many other naming schemes, including the Domain Naming System of the Internet, international telephone numbers, ISBNs, and postal codes.

TOM's also uses a hierarchical naming scheme. It also requires that type names use only printable ASCII characters, to ensure that names are relatively easy to read, type, and transmit. Parts of many TOM type names can also be used as hints (though not always infallible hints) of brokers that are likely to include a description for that type.

TOM's namespace is first divided into initial subspaces with a word followed by a colon that starts the name of a type. Examples of these words include `net`, `mime`, or even individual letters like `s`, `e`, or `c`. The division into initial subspaces is managed by a central naming authority. (I chose initial subspaces to reflect a few common mechanisms for assigning type names. We will see examples of some of these subspaces, and their purpose, later in this section. Table 6.1 gives a summary of the five top-level namespaces currently used by TOM.) Since new top-level subspaces do not need to be created on a regular basis, centralization at this level does not create a scaling bottleneck.

Namespace	Purpose	Controlled by	Example names
`e:`	"Essential" types every broker should know about	Central authority	`e:byteseq`
`s:`	"Standard", common, well-defined types	Central authority	`s:url`
`c:`	"Computable" types: description can be computed from a template	Algorithms set by central authority	`c:seq:s:url`
`mime:`	Types corresponding to the official MIME format set	IANA (official MIME authority)	`mime:multipart/mixed`
`net:`	New types defined by an individual person or project	Each broker	`net:pbm-012396@tom.cs.cmu.edu`

Table 6.1: Major top-level namespaces for TOM types.

Each subspace has its own policy for assigning names within the subspace, and for subdividing the namespace. Names in the `mime:` subspace, for example, are determined by a centralized body, according to the MIME standards. (In TOM, the `mime:` subspace only includes the officially sanctioned MIME types, and not experimental MIME types, where name collision may occur.) Some subspaces, like the `c:seq:` subspace, may even be laid out by an algorithm rather than by explicit human action. In the `c:seq:` subspace, the type name "c:seq:N" refers to a sequence of objects conforming to the type named N, where N is any type name.

TOM's `net:` subspace uses a different naming strategy from those described above. The `net:` subspace lets type brokers on the Internet assign unique names without having to coordinate with any agency outside the broker's own machine. The `net:` subspace is subdivided using Internet host names, giving each Internet host its own naming subspace. Names starting with `net:` end with "@(host)" where (host) is an Internet domain name. A type broker running on a machine named `wheel.compose.cs.cmu.edu`, then, can assign type names starting with `net:` and ending with `@wheel.compose.cs.cmu.edu`. It is up to each individual type broker to make sure it does not assign the same name twice within its namespace. One common way to avoid repetition is to include a timestamp in the name. This technique is also commonly used by email and news servers to generate Message-IDs.

The initial name generated for a new type may be long and unwieldy, particularly in namespaces like `net:`. Types can, however, be given additional names, known as "aliases", after their creation. To assign an alias name to a type, one simply registers the new name with a type broker. The new name, like other names, must not conflict with existing names. A type might start out with a long name in the `net:` namespace when it is first defined, but once it becomes widely used, it might be assigned a shorter, more mnemonic alias in some other subspace.

TOM's naming scheme thus has many desirable features for distributed object systems. Its hierarchical decomposition makes it easy to ensure unique type names. Because the names use a subset of the ASCII character set, they can be easily copied and propagated in a text buffer or mail message. The subspace names can provide hints on where to find the definition of a type. For example, there may be a few brokers known to keep track of all types in the `mime:` namespace. Or, in the `net:` namespace, the type named `net:obscuretype-012897@foo.bar.edu` may well be known at the type broker at `foo.bar.edu`. Because subspaces can have different policies for assigning names, types can be assigned names in spaces where new names are easily assigned (such as the `net:` subspace), or in subspaces that specialize in particularly useful and popular types. (Type brokers can be programmed to give priority to particularly useful subspaces when propagating new type information.) Finally, because types can have multiple names in multiple namespaces, creators and users of TOM types can take advantage of a variety of names, including those that are relatively short and easy to remember.

6.1.2 Coexistence of new types and old types

In some systems, adding new types, even if they have unique names, may cause problems for existing types. For example, if someone can register an arbitrary type T and declare it to be a supertype of an existing type U, and U is not actually a proper semantic subtype of T, the registration can invalidate the client's expectations of U. Furthermore, not only U itself, but all of U's subtypes, are affected due to the transitivity of subtype relations. Hence, the addition of one type can invalidate an arbitrary large portion of the distributed type graph. This vulnerability is not acceptable.

TOM prevents new type declarations from interfering with existing type declarations by requiring that type declarations cannot be introduced with declared subtypes. (Or, to put it another way, new types must be introduced at the "bottom" of the subtype graph.) This policy prevents a new type from affecting the semantics of other types through the subtype relation. A new type can still be initially registered as a subtype of an existing type, but even if this declaration is incorrect, the error affects only the new type and objects of that type, not objects of any other type.

Once a type is added to the type graph, however, other types (both new types and previously existing types) can declare the new type as a supertype. These declarations are made in separate registrations. The addition of such subtype relations needs to be carefully controlled, and will be discussed in Section 6.2.5.

Since newly introduced types are not referenced by earlier types, there is no way in TOM for a new type to interfere with earlier types, except through the subtype relation. Since new types are (initially) not supertypes of existing types, and have unique names, anyone can add new types without worrying about interfering with old types.

6.1.3 Propagating a new type

In a traditional object-oriented program, all types are declared somewhere in the program's source code, or programmer's manual. Anyone can therefore find the definition of any type used by the system by examining the manual or the source code. In a distributed, expanding type graph, there is no longer a single place where all type definitions can be found. So in order for a new type to become useful, people using the system need help finding it.

In TOM, three techniques help clients find the type information they need. The first and most important technique is propagation between brokers. Chapter 5 described how new type information registered on one type broker can become known to the entire Internet. The second technique involves using type names as hints to find brokers with information on the type, as I described in the previous subsection. The third technique is referral: type brokers with a limited repertoire of types may refer to brokers known to collect information about as many types as possible.

6.2 Augmenting existing TOM types

Once a type has been added to the type graph, it may be useful to extend it in various ways. For instance, people might develop new encodings of the type, new conversions involving the type, new servers that can work with the type, or even new methods and attributes for an existing data type.

Type descriptions, therefore, need to be extendable. If a type description could not evolve after it was added to the type graph, it would become less and less useful as services on the Internet developed further, just as a Web page whose links never change becomes less useful as old links go stale, and potentially useful new links are not added.

A system in which all type descriptions were static could still support innovation to a certain extent if a new type were defined for every extension. Clients and programs would quickly fall out of date with such a policy, however, since types that those programs used would be regularly superseded by newer, extended types. Furthermore, the global type graph would become littered with obsolete types,

At the other extreme, it would be a mistake to allow types to be extended or changed without restriction. If arbitrary change is allowed, types will also eventually become less and less useful, because there would be no guaranteed consistency in the behavior of a type over time. For example, in an unconstrained system a method specification might change to something incompatible with the original method specification, causing problems for clients relying on the old specification. Furthermore, in such an environment, different type brokers might have incompatible modifications of the same type. For example, one person might change a type to require attribute A to always be less than attribute B, and another person might change the type to require the opposite.

Current practice is rife with examples of types that have changed in unexpected ways, making programs written for the old version obsolete or even unworkable for the new version. Common examples include many "vendor-enhanced" versions of HTML, and the formats of successive versions of Microsoft Word. Many programs will by necessity change to handle the latest version of particularly popular types, but the costs involved can be substantial. Programs generally will not attempt to keep up with more obscure types. Clearly, in order for a large-scale distributed type system to remain widely usable, it needs to restrict how a type is extended or otherwise changed.

Fortunately, there are policies that allow types to be both consistent and extendable over time. Two key insights govern how TOM allows type descriptions to be extended. First: If the *semantics* and *representations* of a type do not change, programs that rely on the type will continue to work even as other parts of the type description change. Specifically, if nothing is added to or subtracted from the explicit guarantees about the type's behavior, and the operation specifications originally defined for a type are also left unchanged, the semantics will stay the same. Second: It is possible to make many changes to a type description (such as adding new encodings, new conversions, new agents, and even in some cases new methods, attributes, and supertype relations) without making the new type description incompatible with earlier type descriptions. These sorts of extensions (which I call *conservative extensions* due to their preservation of earlier characteristics of the type) can make a type much more useful than it it would be if its description were static, while still being compatible with old programs that used the type as it was originally defined. Conservative extensions are not only compatible with earlier versions of the type description, but are also compatible with other conservative extensions.

Robust extension mechanisms also need to control who can make changes and when changes are allowed, as well as controlling the kinds of changes that can be made to type descriptions. In some contexts, for instance, consistency concerns may be more important than they would be in other contexts. For example, if a type is published and used by many servers, it is more important that it stay consistent than it would be if it was labeled as an experimental type and used by only one broker. While certain kinds of changes do not pose potential problems, others may severely impair the utility of a type if made incorrectly. The ability to make riskier changes may need to be reserved to certain trustworthy people, such as the registered maintainer of a type.

In this section, I discuss specifically what aspects of a TOM type description must remain the same, what can be changed or extended, and how these changes should be made. First I describe what aspects of a type may not change once a type is published, in order for extensions to remain conservative. Then, I briefly discuss how an initial, nonconservative, "experimental" period can be useful in the initial development of a type. In the remaining sections, I address the issues raised by different kinds of extensions: new methods and attributes, new conversions and encodings, new subtype relations, and new agent information. I explain what consistency and usability issues are raised by each extension, and show how TOM's policies preserve consistency in the type graph while allowing a type's functionality to be maintained, and extended, over time.

6.2.1 Fixed aspects of a type

I have already argued that a type must have a persistent and unambiguous name in order for it to stay accessible as the type graph develops. To stay useful, its semantics need to remain consistent as well, so that existing applications continue to work in the face of changes.

Therefore, after a type is published, TOM requires that the type's initial name(s), the type's stated semantics, and the signature and semantics of its initial operations cannot change. The existing names cannot change because clients depend on type names remaining consistent. The static semantics of the type cannot change because changes in these semantics may violate previous guarantees of the type's behavior. If the semantic constraints narrow, they may make a formerly valid implementation no longer reflect the published semantics of the type. On the other hand, if they widen, constraints that clients depended on might no longer be guaranteed. For similar reasons, the signatures and the stated semantics of previously defined operations cannot change.

While these semantic aspects of a type may not change, other aspects of a type can still be added over time. For instance, new names (aliases) for a type, may be added as described in Section 6.1.1. Also, certain new methods and attributes can be added, as can encodings, conversions, and subtype relations, subject to restrictions described later in this chapter.

While the interface and semantics of operations are not allowed to change, *implementations* may vary over time, and from server to server, as long as the implementations stay consistent with the published semantics of the methods.

6.2.2 Handling change in the basic definitions of types

Once a type is declared published, certain fundamental aspects cannot change. However, determining the most useful definition of a type may require an initial period of planning and experimentation, How can TOM leave room for early experimentation with a type, while still being able to guarantee consistent type behavior to clients?

One simple but strict policy would be to require a new type every time the basic semantics were revised, even in early experimental stages. Creating a new type for each revision, though, would lead to a cluttered type graph. Moreover, in the initial experimental phase a type is generally used only locally, so consistency concerns are less of an issue that they are later on.

Therefore, rather than require a new type to be defined every time one changes the type definition in this experimental period, TOM allows a type to be tagged as "experimental" when registered with a type broker. (TOM's experimental types, however, are unlike MIME's experimental types, in that they can still be registered with brokers, and do not need to change their name once published.) The definition and semantics of experimental types are subject to change, until the author of the type upgrades it to "published" status. The "experimental" status designation on the type's description object warns clients not to depend on the type's semantics, and type brokers might decide not to propagate experimental types to other brokers. This "experimental" stage supports flexibility and experimentation while designing a type, without forcing a large number of obsolete types to litter the type graph. At the same time, once a type is declared "published," its behavior can then be guaranteed, and clients and implementations can use it without worrying that the definitions will become incompatible with earlier published definitions. Definers of types have strong incentives to mark a type as "published" once it is stable, if they want their types to be used widely.

In the remainder of this section, I will discuss only changes and extensions that are allowed after a type is declared "published."

6.2.3 New methods and attributes

The initial set of methods and attributes defined for a type may not be sufficient for practical use of the type. In particular, if a commonly performed task requires multiple method calls, it may be easier (and much quicker) to define a new method that performs the entire task in a single call. Operations of this sort are known as "convenience functions" in object-oriented languages. In TOM, they are called "derived" methods or attributes, because their behavior is derived from that of the original methods and attributes defined for the type. (Those original methods and attributes are known as "basic" methods and attributes.)

Certain kinds of operations can be added to a type without changing its semantics. In particular, operations that can be defined as functions of existing, "basic" operations (combined, perhaps with additional information independent of the object itself) are safe to add. Derived methods in TOM are equivalent to the methods defined in Liskov and Wing's "extension maps," [LW94], except that TOM's derived methods are defined on the same type as the methods they "extend", rather than on a subtype.

To see why derivability is important, consider the following example. Suppose we define a simple type named *bitmap*, with a constructor function, `width` and `height` attributes, and a `bit` operation that, given appropriate x and y, will return whether the bit in row x and column y is set or clear. We define a number of encodings for it. One encoding, for instance, consists of a pair of integers representing width and height, followed by a binary representation of the bits in the bitmap.

Suppose that we wish to add more operations. For example, we may want a method called `firstsetrow` that returns the index of the first row that contains a set bit, or -1 if there are no set bits. (Such a method would help us crop or center a bitmapped image.) The semantics of this method can be derived from that of existing methods. In particular, it can be defined as the integer x_0 for which $bit(x, y)$ is clear for all x less than x_0. and such that either x_0 is -1, or there exists a y_0 such that $bit(x_0, y_0)$ is set.) This method is implementable through existing operations on the object, without any additional information from the object. Hence, it is well-defined for all valid objects of the *bitmap* type, in all valid encodings, and can be added to the type without violating the expectations of either clients or implementors of the type. Furthermore, a client that wants this information would to get it much more quickly by calling `firstbitrow` once than it could by calling `bit` many times.

Certain other desired operations, though, would cause problems for users and programs that rely on the type. Suppose that one wanted to add a `color` method (with parameters x and y), in the hope that a *bitmap* could then carry information about the color of various pixels. Unless the color is defined simply as a function of the bitmap (such as defining a 1 bit to be black, a 0 bit to be white, and allowing no other colors), or color is determined by a cell's location or neighbors, this operation *cannot* be defined as a function of existing operations. It therefore is not well-defined for all *bitmap* objects, unless the semantics of the *bitmap* type are extended so that every object carries whatever additional information is required to compute this operation.

Such an augmentation, however, would cause problems for existing constructs that reference the type, particularly for the type's encodings, subtypes, and any operations that return new objects of the *bitmap* type. We recall from Chapter 4 that objects are transmitted between machines in encoded formats. The encodings defined on a type are defined to carry enough information to completely represent an object of a given type, and need not carry any additional information. If a type gets a new operation that is completely derivable from old ones, the encodings of the type still work, since they do not need to carry any additional information. If, however, a new operation requires additional information from an object, the existing encodings, which might not carry this information, would no longer work. An object which reported certain `color` values on machine A might report different `color` values when transmitted to machine B via an old encoding, since when the encodings were devised, there was no requirement that they faithfully preserve color information.

Similar problems exist for operations that return the *bitmap* type. When originally defined, such operations had no obligation to produce any particular `color` values in their return values. Likewise, subtypes of the *bitmap* type may have been defined without taking color into account; if color is added to the *bitmap* type, then, the subtype relation between *bitmap* and color-less subtypes may break.

Derived operations, which are defined as functions of existing operations (where the functions do not use additional information from the object) do not have these problems. If basic operations can be legitimately called on an object in some particular encoded format, then so can the derived operations, since it is possible to calculate the derived operations from the basic operations. For the same reason, operations that return objects of a particular type will return objects that support both that type's basic operations and its derived operations. Likewise, derived operations can be called on the type's subtypes, and can be calculated based on the previously-defined basic operations. Therefore, one can always add derived operations to a type without violating any of the guarantees implied by a type's initial definition.

When a derived operation is added to a type, it can be registered with both a semantic specification and a derivation, so that readers can see how the new operation can be calculated from the basic operations. It can then be called in exactly the same way as a basic operation.

The derived operation need not need be *implemented* as a series of basic operation calls. For instance, while one can derive the semantics of integer multiplication from repeated integer addition, one would not normally want to implement $a * b$ with a loop that added a to a running total b times. Implementations of derived operations can take whatever shortcuts are appropriate, so long as their results are consistent with the operation's definition.

However, it can be advantageous to implement derived operations by invoking basic operations, if this is efficient, when one expects subtypes of the type for which the derived operation is defined. Doing so enables code reuse in subtypes. Consider a type T with basic attribute b, and derived attribute d, which can be calculated by fetching b and feeding the result into a simple mathematical function. Suppose one implements d by doing this fetch, and then applying the mathematical function. Then, if one later defines type U, a subtype of T with different encodings, one may need to re-implement the fetch of the b attribute, but the d attribute implementation from T can be reused once the b attribute fetch for type U has been implemented.

In Chapter 4, I noted that "class" methods can be associated with a type, usually for creating a new object of that type. Since class methods are not called on a specific object, they also do not specify any additional information carried by an object type. Therefore, they too can be added to a type at will, without disturbing the previous semantics of the object or violating the assumptions of any object encodings.

Because many new derived operations and class operations can be usefully added to a type's description, TOM allows derived operations to be registered in a distributed manner, just like types. New operation definitions propagate through the broker network in the same way that new type definitions propagate. Other new type information also propagates in the same way.

6.2.4 New encodings and conversions

As we have seen in Chapter 4, encodings define representations of an abstract object as a more concrete object. There can be any number of representations for a particular object type. They do not change the basic definition of a type, but rather are based on the information a type has already been defined to carry. Therefore, TOM allows new encodings to be added at any time.

Conversions are also useful to add, particularly as new types and encodings are defined. New conversions make it easier to move between different types and formats. Since conversions are defined with respect to a type's encodings, which are in turn based on the basic definition of a type, new conversions can also always be added without changing the basic definition of a type.

6.2.5 New subtype relations

The subtype relation is a particularly useful relation between types, allowing types with similar semantics to be grouped together. There are at least three important reasons for establishing subtype relations. The first has to do with reuse: if type T is a subtype of S, objects of type T can be used in contexts where objects of type S are expected, with only minimal implementation support for type T. As we saw in Chapter 4, one conversion from a format of T to a format of S, preserving the information of type S, allows all the operations implemented for S to be also invoked on objects of type T. Second, the subtype relation is a conceptual aid for people looking for closely related types. Third, the subtype relation allows conversions to be defined between related types that guarantee the preservation of essential information, by declaring a supertype as an intersubstitutable type for a conversion.

In traditional object-oriented programs, the subtype graph is determined in advance, or grows in a carefully controlled fashion, usually by adding new subtypes to the bottom of the type graph. In distributed, expanding object systems, however, to get the benefits described above, developers need to be able to add new subtype relations over time, either at the bottom of the graph or in the middle of the graph, as they see fit. There are three particularly useful ways to add subtype relations. First, a new type can be added as a subtype of an older type. Second, two existing types can be linked together in a supertype-subtype relation. Third, a new type can be created that represents an abstraction or generalization of some existing types, and those existing types can then be made subtypes of the new type.

Distributed object systems that allow the addition of new subtype relations need to make it clear what it means to add a subtype relation. Once this is established, two questions remain. First, when can a subtype relation be asserted? Second, who should be allowed to assert the relation?

Chapter 4 defined the semantics of a subtype relation to be essentially a constraint on the subtype. Specifically, the subtype relation asserts that the semantics of the subtype are a specialization of the semantics of the supertype, and that the subtype supports all of the operations that the supertype does. A subtype relation cannot change the definition of the supertype; at most, it asserts that the operations defined in the supertype can be safely invoked on objects of the subtype.

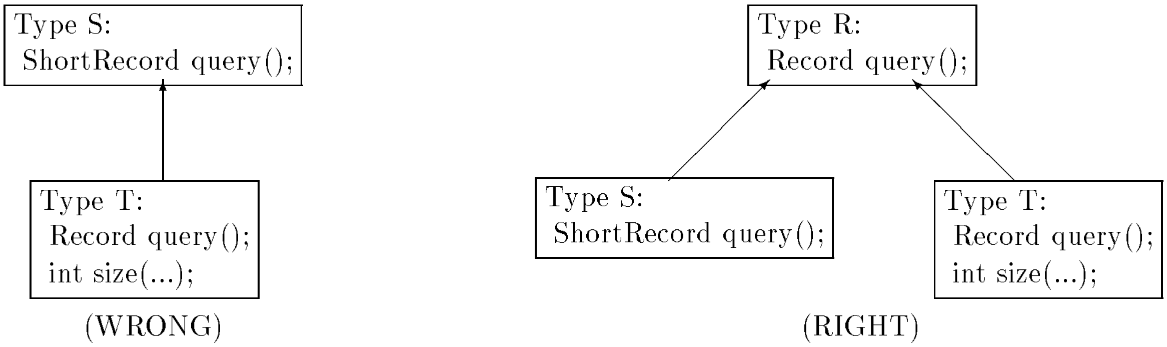


Figure 6.1: How to relate two previously defined types through subtype relations

Because subtype relations constrain the subtype, such relations should only be asserted when the subtype semantics actually do imply the supertype semantics. If asserting the subtype relation either contradicts or further restricts the initially-defined semantics of the "subtype", previous expectations about the "subtype" will be violated.

Suppose, for instance, that one asserts type T as a subtype of type S, but T is not actually a subtype of type S (either because T's semantics contradict S in some way, or the interface of S cannot be supported by the interface of T). In this case, a client's expectation of the behavior of type T (and any of T's own subtypes) will be violated. A bad subtype relation, then, can "break" the subtype.

To see how restriction also causes problems, suppose that that types T and S both describe databases, and have the same set of operations, but that type S requires that all database records be less than a certain length, and the original definition of type T does not. If type T is then declared to be a subtype of type S, it inherits this semantic constraint, placing a new restriction on the type. This restriction may suddenly invalidate objects that were previously legitimate instances of type T. It may also break other subtypes of type T that did not impose this requirement.

To avoid these problems, TOM requires that the subtype relation not impose additional semantic constraints on the subtype, or contradict semantic assertions made by the subtype.

If one wants to relate two similar types, neither of which strictly specializes the other type, one can create a third type that embodies information common to both types, and then make subtype relations from the first two types to the third. For example, in the database case above, one can make a new supertype R that has the operations and semantic constraints that are common to both S and T. Types S and T can then both be made subtypes of type R. Clients that wish to handle objects of type S and of type T can use the interface provided by R, as shown in Figure 6.1. Liskov and Wing call such newly defined supertypes "virtual supertypes," since they are often not meant to be instantiated directly.

Systems like TOM cannot automatically verify whether an asserted subtype relation is valid, or whether it improperly introduces additional constraints on the subtype. Instead, for the graph to stay consistent, the person who asserts the subtype relation must be trustworthy. Therefore, TOM requires that the maintainer of a type authorize any assertions of a subtype relation of that type with a supertype. The maintainer of the subtype is presumably knowledgeable enough to be able to verify whether the subtype assertion is correct. (The proof of correctness is usually straightforward.) Furthermore, if a subtype assertion is incorrect, the error will affect the subtype, and any subtypes of that subtype, but not any other types. The affected portion of the graph, then, is already being maintained by the person who authorizes the new subtype relation, or by people who trust that person when they declared that their types would be subtypes of his type.

In contrast, TOM does not require that the maintainer of a supertype authorize the establishment of subtypes of that type. As we have seen, a subtype relation gives no new information about the supertype. Also, there may be dozens of useful subtypes for a single supertype, and requiring them all to be approved by the supertype maintainer could unnecessarily hinder the introduction of useful subtypes.

TOM's subtype relations are recorded in the type description object for the subtype, and not in the type description object for the supertype. (This practice is similar to that of most object-oriented languages. For example, C++ type declarations explicitly list a type's supertypes, but not its subtypes.)

In many object-oriented systems, adding new subtype relations may also have unintended consequences for method dispatch, particularly when two related types define operations with the same name. In Section 6.3, we will see how TOM handles method dispatch so that appropriate operations continue to be invoked as the type graph evolves.

6.2.6 Agent information

As we saw in Chapter 5, a type description object includes a list of agents that implement the operations defined for the type. The listing of agents is completely independent of the type definition, and simply points to servers with valid implementations of the operation. As servers appear and disappear from the Internet, the set of agents that implement a particular operation can expand and contract.

A type broker should keep a sizable list of agents for each operation in its database. That way, when an operation is requested, the broker can pass the request on to any of multiple servers, and invoke a backup server if one server fails. It is also possible (though not currently implemented) for type brokers to occasionally poll agents themselves, requesting operations from time to time. If certain agents are chronically unreachable or unreliable, the broker might remove or downgrade their position in the agent list.

6.2.7 Naming conflicts

In the sections above, we have seen that TOM allows new operations, encodings, and conversions, to be added anywhere on the Internet. It is possible, in theory, for two people to choose the same name for two different encodings, conversions, or operations, and hence introduce a name conflict within a type description. (This is possible if they introduce the names at roughly the same time, before either of the names has had the chance to reach another type broker.) How can these conflicts be avoided or handled?

In Section 6.1.1, we saw that type name conflict could be avoided by dividing the type namespace hierarchically into separately administrated units. The type names generated using this approach, however, are often long and unwieldy.

When naming encodings, conversions, and operations defined within an existing type, a more optimistic approach to naming makes more sense. (In a pessimistic naming scheme like that used for types, names are generated so that they cannot conflict. In an optimistic scheme, one instead generates names that might conflict, but assumes that conflict is unlikely, and can be resolved when detected.)

Why does an optimistic approach make sense for names defined for aspects of a type? Since the initial encodings and operations defined for a type are propagated everywhere the type is, we know that the names of the most essential aspects of a type description will not conflict with other names, since brokers will have records of those names, preventing inadvertent reuse. Furthermore, a single type, unlike the type graph as a whole, usually develops at a slow pace, with only a few people augmenting the type at any given time. Therefore, the chances of two incompatible names within a type being generated at the same time is quite small; and most likely can be detected and repaired relatively quickly. In the rare case where two operations or encodings are assigned the same name at about the same time, the maintainer of the type can decide which one has the right to the name. TOM therefore allows definers of new encodings, operations, and conversions to pick whatever name they see fit, so that the names stay compact and mnemonic.

Applying TOM's policy to conversions and encodings, both of which are defined for one type only, is fairly straightforward. However, since subtypes can inherit and specialize attributes and methods, preventing naming conflicts for attributes and methods in subtypes is nontrivial. The next section describes how TOM defines and dispatches methods and attributes to avoid name conflict and other confusions.

6.3 Method dispatch in evolving type graphs

Because TOM clients and servers operate in a growing network of types and methods, method dispatch must be designed carefully to be unambiguous, scalable, and uncomplicated. In this section, I examine the general problem of method dispatch, show how it is handled in some conventional programming languages, highlight some of the problems in adapting method dispatch to distributed, expanding type systems, and then show how TOM's method dispatch handles these problems.

6.3.1 Introduction to method dispatch

A key feature of object-oriented programming models is dynamic method dispatch. Instead of calling a fixed procedure on a piece of data, a caller calls a method on an object, and an appropriate implementation is found and invoked at run-time, based at least in part on the type of object. The same method call may end up invoking different code, and causing different behavior, depending on the type of object used in the request.

Why have the same method name refer to different pieces of code, instead of always calling the same code? Object-oriented systems typically use ad-hoc polymorphism, where procedure calls are "overloaded" to apply to multiple types, to simplify the client's view of the world, while still allowing code specialized for particular data structures to be run when appropriate. We have seen in earlier chapters, for example, that the generic *reference* type is defined with a `fetch` method that returns the object indicated by the reference. Specific types of references, like URLs or Exodus database handles, can be dereferenced using specializations of this method for working with the Web or Exodus databases. The ad-hoc polymorphism of object-oriented systems allows the appropriate routine in any *reference* subtype to be invoked with the same generic `fetch` method call.

Dynamic method dispatch in object-oriented languages serves two basic functions. First, it supports semantic specialization, by allowing methods defined in a type T to be specialized in a subtype U. The *reference* case above is an example of this function. A client can thus call a method on an object conforming to type T, and if the object is actually of type U and has a more specialized method, that method will be invoked. This is an example of "downward dispatch". Second, it supports reuse, by permitting methods defined in supertypes to be reused in subtypes without explicit redeclaration. For example, if type T is a subtype of type S, which defines method *foo*, then the method *foo* can also be called on objects of type T, even if there is no definition of *foo* for T. This is an example of "upward dispatch".

The run-time dispatch mechanism must find an appropriate method implementation to invoke for a given method request. The usual mechanism is based on the assumption that name correspondence implies semantic correspondence. That is, if a method with name N is declared in type T, a redeclaration of the method in a subtype U_1 is treated as a specialization of the method in type T, and can be used in downward dispatch. Similarly, requests for the method named N in subtype U_2 of T, where U_2 does not redefine N, can be satisfied by an invocation of the method named N in type T: this is a case of upward dispatch. These two ideas lead to a basic dispatch algorithm: given an invocation of a method named N on an object with type T, the dispatch mechanism looks at T, and then searches upward in the subtype graph from T, until it finds a method with the name N in T or a supertype. It then invokes that method.

Some languages have more complex dispatch mechanisms than the algorithm described above. Those with multiple inheritance, for instance, may require the dispatch mechanism to decide between two supertype methods with the same name. Other languages allow the input arguments to a method to determine which method implementation is used; methods dispatched by such mechanisms are often known as "multi-methods". In some programming languages, such as CLOS, a programmer can even arbitrarily redefine how method dispatch works.

While a dispatch mechanism may assume that name correspondence implies semantic correspondence, most languages do not actually require this correspondence. It is possible for a subtype's method to have the same name, and even the same signature, as a supertype's method, but do something completely different. (If there is a complex subtyping graph involving multiple inheritance, this may be not only possible but probable.) In most languages (Eiffel [Mey88] being a notable exception) neither the compiler nor the run-time system will complain about this misleading correspondence. The definitions of many object-oriented languages and run-time systems do not even explicitly state any requirement that subtype methods and supertype methods act similarly.

Under such conditions, how can one rely on dynamic method dispatch to invoke methods that have the semantics a client expects? In practice, most programs and their type hierarchies are designed by a small set of developers who work together. Such groups of developers typically set up local conventions among themselves so that misleading name correspondences are not introduced, and so that subtype methods specialize supertype methods in well-understood ways. They also control the growth of the type hierarchy (or at least the top levels) so that additional subtypes do not violate these conventions.

6.3.2 Problems with method dispatch in distributed object systems

In a distributed, expanding type system like TOM, local conventions are not sufficient to ensure correct type dispatch. TOM's graph of types is distributed, constantly growing, and added to by many independent parties. The independent parties that are augmenting the type graph cannot all confer with each other. They also cannot see all of the type hierarchy; in particular, they cannot be sure of seeing all the types that are below the type being augmented.

Hence, it is easy to introduce new methods and subtype relations that cause name conflicts with existing methods in other types. Name conflicts can occur when methods are added with the same name as methods in supertypes or subtypes, but which have little or no semantic relationship to the other type's methods. A new subtype relation might also put two methods with the same name (but different semantics) in conflict with each other. Given this potential for introducing name conflicts, how can programmers calling a method be able to trust that the invocation will produce consistent behavior in an evolving type graph?

One way of solving this problem would be to severely restrict how the type graph can evolve. For example, one might decide to prohibit a type's methods from having names that match method names in supertypes, unless the methods specialize the supertype methods. However, a rule like this is both impractical to enforce and overly restrictive, in part because the enforcing mechanism may not know of all the possible conflicting methods in supertypes and subtypes. The enforcing mechanism has a reasonable, if still not perfect, chance at preventing name conflicts within the type itself, as I noted earlier, but if an arbitrary number of supertypes and subtypes have to be taken into account, name conflicts become much more likely, and harder to detect. Adding new subtype relations poses an even more difficult problem, since subtype relations may introduce multiple naming conflicts at once.

Since it is not practical to prevent naming coincidences across types, a workable dispatch system requires more advanced techniques than simple method name matching. The basic requirements of a such method dispatch mechanism are:

- **Consistency.** When a method request is made, a method implementation should be invoked that is consistent with the semantics of the method request, if any such method implementation is available.

- **Conservativeness.** The dispatch system should behave predictably even as more type information is added by diverse parties, without clients of existing methods having to know about new methods, and without introducing new conflicts as new types or methods are introduced. The growth of the type graph should not unnecessarily constrict the ability to define new types and methods.

- **Simplicity.** Type clients should not be burdened with unnecessarily complicated method names or inheritance rules. Type definers should not have to do increasingly more work to define a type or method as the type graph grows.

Solving the consistency problem requires that it be possible to determine whether a given type's method satisfies a client's method request. As I argued in Chapter 4, ensuring consistent behavior between subtype and supertype methods requires that there be well-defined guarantees about the relation of a subtype method to a "corresponding" supertype method. The substitutability requirements of TOM make it possible for clients to call any subtype method of a method defined in type T, and know that the method's behavior will be consistent with the definition in T.

While the substitutability requirement solves the consistency problem in a static type graph, it does not solve the conservativeness problem for an evolving type graph. Dynamic method dispatch allows the invocation of implementations for methods defined in types above or below the type that the caller had in mind, as I showed earlier in this chapter. Because these methods can be added at any time, their existence cannot be known in advance, but the dispatch mechanism must still be able to distinguish new methods that are consistent with a particular method request from those that are not.

Object-oriented languages designed for small-scale type graphs do not adequately guarantee conservativeness. Consider the dispatch mechanism of C++, for example. If subtype relations are added between types that coincidentally use the same name and signature for completely different methods, the client may end up invoking a method that has no relation to the intended method. Suppose that type T is a subtype of types A and B, and A has a method named N, which is not redefined in T. Then invoking N on an object of type T will invoke the method defined in type

A. But if a method also named N is added to type B, or to any supertype of B, invoking N on an object of type T might then invoke B's method instead. In a standalone program, programmers can avoid this problem by knowing not to add a type named N to type B. In a system like TOM, where new type relations and method definitions can be added at any time, these problems cannot be so easily avoided. If TOM were to use C++'s method dispatch semantics, a client program might suddenly break as a result of an addition to a distant part of the type hierarchy. This risk is unacceptable.

It is also unacceptable to simply prohibit subtype relations that might cause name conflicts. One cannot ensure that when a type is defined, no name conflicts with supertype methods can ever occur, unless the supertypes and their interfaces are already known and fixed in advance. If the type interfaces are not frozen, adding a method to a supertype may suddenly introduce a conflict with a subtype defined on another broker, not known to the person adding the method.

One way to avoid method name conflicts between types would be to require that every type's method names come from a different namespace. This approach would violate our simplicity requirement, both because it would require unwieldy names, and also because it would then be unclear which subtype methods were specializations of which supertype methods.

The distributed environment also rules out arbitrary redefinition of the dispatch function, such as occurs in CLOS, because methods may be dispatched and invoked on machines that the client does not know of or control. If the rules for method dispatch could be arbitrarily rewritten on any of these machines, a client would not be able to get reliable results from the type broker network. Similarly, most dispatch mechanisms that rely on the types of input parameters or other auxiliary data would be too complicated to be practical in an environment where dispatch and type definitions are distributed.

6.3.3 How method dispatch works in TOM

TOM removes ambiguity in method dispatch by including extra information in the definition of a type, and in a method call. In the type definition, TOM requires methods that specialize supertype methods to explicitly declare the specialization in their method definition. If a method is so declared, it must satisfy substitutability requirements as described in Chapter 4. If a method is not so declared, TOM will not treat it as a specialization of a supertype method, even if the supertype method has the same name. In the method call, a client calling a method can specify not only the name of the method but the name of the type that defines the method. The method that is invoked, in this case, will either be the method specifically defined for that type, or a specialization of that method.

To see how these rules work, suppose that type T has a method named N, which is a specialization of another method names N in a type R, a supertype of T. The definition of N in T notes explicitly that it specializes the method in R. This "link" allows T's method to be invoked by a client who is using either R's interface or T's interface. If S, another supertype of T, also has a method coincidentally named N, it will not be invoked if a client calls method N on an object of type T, unless the client specifically asks for the method N that is defined in type S.

TOM allows specializations of methods to have different names from the methods they specialize, (The "renaming maps" of Liskov and Wing serve the same function.) This renaming can be useful for linking methods previously defined with different names, or when a type needs to specialize two methods with the same name from different supertypes.

It may seem burdensome to the definer of a type to have to make explicit specialization declarations for each method. Without such declarations links, though, brokers cannot reliably determine which supertype methods with the same name as a subtype's method are being specialized by the subtype's method– or even if the subtype's method represents a specialization at all. Furthermore, type definers typically know which supertype methods they are specializing, so they can easily make the proper declarations. Such declarations also provide useful documentation for implementors.

TOM's dispatch mechanism uses three pieces of information from a client's method request to determine which method to invoke:

- The method name specified by the client.
- The name of the type that defines the method, as specified by the client. (The client may omit this information, but doing so may result in the invocation of a method with unexpected behavior.) I will refer to this type as the "method type" in the discussion below.
- The type of the object on which the method is being called. (If the client calls a class method, no object is provided, and the client must specify the name of the type defining the method.)

The client often needs to specify both method type and method name to make an unambiguous method to call. If the client requests a method named N with method type T_1 on an object of type T_2, then only the method N defined in T_1, or specializations of that method, can be invoked. If there is an implementation of a method defined on type T_2 that is declared to specialize method N of T_1 (either directly or through transitive closure), it can be invoked. If there is not, the method dispatcher can check for appropriate specializations in T_2's supertypes. Because of the strict substitutability requirements of TOM, any method that is a specialization of T_1's method will satisfy the semantic constraints for T_1's method. Furthermore, implementations defined for types near T_2 are most likely to be optimized for formats of that type.

Why not simply omit the method type, or just make it T_2, in this case? If the method type is omitted, the client may end up invoking a method defined in another supertype of T_2 that the client did not expect. After all, the method named N in T_2 might actually specialize a method in another supertype T_3, unknown to the client.

The correct behavior for calling a method with name N and type T_2, when T_2 does not have its own definition of N, is to report that no method was found, even if there is a method of the same name defined in a supertype T_1. Why? Because there may be a method named N in T_2 that specializes and constrains method N in T_1, but that is not known to a particular type broker. In that case, it would be erroneous to invoke T_1's implementation, since that would not give the client the same semantic guarantees that it might be expecting for the method defined in T_2.

In practice, sometimes a client may know little about the type graph, and not care about the exact semantics of the method being called. TOM therefore allows the client to omit the method type, for simplicity. If the type broker knows a definition and implementation for the method for the type of the object being passed in, that method will be invoked. Otherwise, it looks up the subtype graph until it finds some method with the same name and compatible signature that can be called. There is no guarantee as to what method will be called, or what its semantics might be, but in many cases this will give "good enough" results, and be likely to invoke some method instead of returning a "not found" error and requiring the client to search the type graph to find something else it can use.

6.3.4 Implementing the method search

TOM type brokers dispatch method requests by searching up from the type of the object on which the method is called, until they find a method that has specialization links up to the method

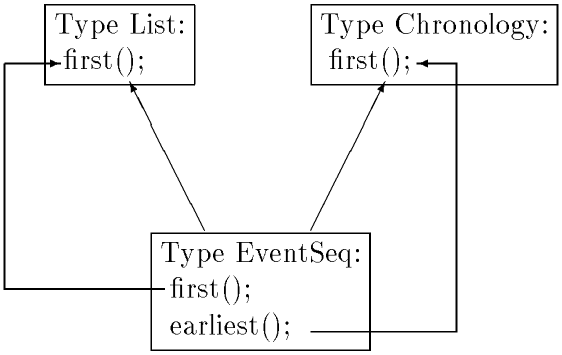


Figure 6.2: A type specializing methods from two supertypes

with the name and type given in the request. In the search, brokers may need to look at many methods, particularly if the object type and the method type are far apart, since the method may be arbitrarily renamed in subtypes. However, the broker does not need to search types that do not fall underneath the method type provided by the client. Also, the broker can ignore methods whose signature does not conform to the method type's signature, since the strict specialization requirement constrains the signature of subtype methods. Thus the search time required for method dispatch is kept down even in the presence of renaming.

The dispatch algorithm described above supports upward dispatch (up to the method type, but no further). A more complex dispatch algorithm could also look below the type of the object on which the method is called, to support downward dispatch, but our prototype type brokers do not implement such searches.

6.3.5 An example

To demonstrate how method specialization and dispatch works in practice, suppose we declare a type named *EventSeq*, which consists of a list of timestamped events, and we want it to be a subtype of the more general type *List*. The *List* type has a method name `first` that returns the object at the front of the list. We decide to specialize and implement the `first` method for the *EventSeq* subtype, so we redeclare it in that type, and say explicitly that it is a specialization of the `first` method found in *List*, as shown in Figure 6.2. We also define other attributes and methods for *EventSeq*s not shown in the figure.

Because of the specialization declaration, if a client asks to execute a method on an *EventSeq* object with the semantics of the `first` method in type *List*, the dispatcher knows that it can invoke the `first` method as implemented for *EventSeq*.

Since *EventSeq*s include timestamped events, one can think of them abstractly not just as an ordinary list, but as a chronology of times and observations. We may, then, decide that it would be useful to make *EventSeq* a subtype of another type, *Chronology*, which has methods dealing with temporal ordering. Like *List*, *Chronology* also has a method named `first`, which returns the observation with the earliest timestamp. Unfortunately, the `first` method in *EventSeq* is already constrained to return the first event in its list, which may or may not be the one with the earliest timestamp. So it cannot substitute for the `first` method in *Chronology*.

By using renaming, we can still allow the *EventSeq* type to be a subtype of both *List* and *Chronology*. We declare a new method called `earliest` (derivable from other methods in *EventSeq*), and declare that this specializes the `first` method in *Chronology*.

This linking and renaming allows *EventSeq* objects and their methods to be used through the interface of either supertype. If a method is called on an *EventSeq* object, with method name `first` and method type *List*, the `first` method on *EventSeq* can be called. If a method is called on the same object with method name `first` and method type *Chronology*, the `earliest` method on *EventSeq* can be called, and the `first` type on *EventSeq* will not be called (since it is not linked to the `first` method on *Chronology*). If an implementation cannot be found for the methods on *EventSeq*, the appropriate supertype method can be called. In all these cases, the client gets the behavior it expects, so long as it specifies a method type.

6.3.6 Summary of TOM method dispatch

From the preceding discussion, we see how TOM's method dispatch mechanism satisfies the requirements stated earlier:

- **Consistency.** An object's methods can always be accessed be specifying the object's type as the method type, and can also be invoked by specifying supertype methods that the object's methods specialize. If the methods are properly defined in accordance with TOM's subtyping rules, the method's behavior will always be consistent with the client's expectations.
- **Conservativeness.** Adding new types and methods does not change the behavior of old method calls (except possibly in the informal case where a caller does not specify a method type), due to the use of explicit specialization declarations, and method types in method calls.
- **Simplicity.** While TOM's method dispatch involves constructs like method links that are not present in other object dispatch systems, the explicit links make it easy to determine what methods can be called in response to any given request. They also guarantee that some method satisfying the request will be called if possible, if the appropriate links were declared. The amount of work required to define new types and methods does not grow with the type graph. Furthermore, the definer of a method typically already has the information required to make a specialization link, so these links can be easily added.

6.4 Chapter summary

In order for a decentralized type graph to be widely used, it must be able to grow and change in a distributed fashion. At the same time, clients that use existing types should be able to rely on their semantics, and not be required to change in order to keep up with the development of types. We have seen in this chapter how both of these requirements can be fulfilled: using the design principles of TOM described here, type graphs can *conservatively* change so that they cover an increasingly broad range of related types and services, while still staying consistent with less-developed versions of the types. In particular, I have shown how type naming, type evolution, and method dispatch are specially designed to accommodate the growth of the type graph. This growth, in turn, allows TOM to scale gracefully to a global network of types and services.

Chapter 7

TOM in Action

In previous chapters, I discussed the design and implementation of TOM, explaining analytically how it is designed to make a wide, expanding, range of data formats easier to use. In this chapter, I show how well TOM works in practice. This chapter shows practical applications of TOM in two areas. In the first area, data conversion services, we will see how TOM conversion applications make it easy for everyday users to convert a wide range of data formats to forms they can use. In the second area, frame services, we will see how TOM allows new forms of data to be introduced easily into an information system, raising the system's level of abstraction while still allowing the new data formats to be widely used by existing programs. In both cases, I show how TOM performs useful services that cannot be easily matched by other systems. I also show how various aspects of TOM's design work together, and how they can be applied to important problems.

7.1 Conversion services

7.1.1 The conversion problem

The need for conversion services

One of the main problems with data communication is that data often comes in a form that is not useful to the recipient. Internet users at all levels of expertise encounter problems such as not being able to decipher some piece of information they wish to work with, not having appropriate programs to work with the information, or not receiving data in the most convenient form.

A common solution to this problem is to convert the data to a familiar format. For example, someone who gets a Microsoft Word document, but does not have a Microsoft Word program, might convert the document to ASCII text or HTML. A picture in an unfamiliar Windows bitmap format could be converted into a more familiar GIF format. A mail message with incomprehensible MIME attachments could be converted from an unreadable MIME-encoded format to a text, image, or audio format that the recipient could examine directly.

The demand for such conversion services is widespread, as we found in discussions with people from a wide range of communities, ranging from school teachers who were new to the Internet to professional computer scientists with state-of-the-art hardware and software. People at all levels of expertise, and with all kinds of hardware and software configurations, wanted such conversion services, and were highly interested in TOM's potential to provide such services. Therefore, even though TOM's Web-accessible conversion services are still in development, they are already being used hundreds of times a week by users all over the world.

Problems with existing methods

Since the need for conversion services long predates TOM, users have attempted a variety of approaches to meet this need. In particular, users have widely relied on four general solutions: settling on standardized formats, finding and installing single-purpose converters, using converters bundled with standard applications, and using standalone "universal" converters. These approaches have all had serious problems or limitations.

Chapter 1 discussed the problems of relying on a few standardized formats for all communication. Consensus on the formats is not practically achieved except in small or tightly-controlled groups. The "standard" formats will inevitably shift over time as new applications and revisions are released. Also, at any given time, a large, diverse community will require many formats, rather than just a small standard set, to do their work. Therefore, the need for conversions will not be eliminated by standard formats.

There are many programs available that convert from one specific data format to another. For example, `latex2html` converts LaTeXdocuments into HTML and image files, while `mcvert` converts Macintosh files into ordinary Unix files. Many of these conversion programs can be freely obtained through the Internet or other channels; others are sold as standalone products or bundled with related applications.

Unfortunately, installing and using these programs can be cumbersome, and many users will not be able to use them at all. Imagine that a user finds a file or a document in some unknown format, and wants to convert the file to a more useful format. First, he has to figure out the current format of the document. Then he has to see whether he already has a program that will convert it into a desired format. If not, he may have to search the Internet, or a software store, to find a program that will do the required conversion. If he are lucky enough to find one, and can verify that it will run on his machine, he still needs to download and install it on his machine, and learn the idiosyncrasies of invoking that particular program, before he can convert the document. Then, if he later gets another document in a different unfamiliar format, he may have to start this process all over again. This process requires careful attention to many low-level details, and is time-consuming for experienced computer users and impractical for inexperienced users. While the conversion programs can be useful for their particular tasks, once users have learned how to use them, the standard infrastructure for finding and using them is grossly inadequate.

Easier to use, but more limited in scope, are conversions that come bundled with larger application programs. For instance, recognizing that people who use different text editors often want to share documents, Microsoft Word programs permit users to load documents not only in Word's own formats, but also in the formats used by other popular programs. Often, programs that allow this sort of loading will convert documents in a "foreign" format into their native format in the process. Some of these programs also permit documents to be saved in a variety of formats. These features can take care of some conversion problems, if one already has the proper application program, and if the conversion one needs is supported by that program. But if the required conversion is outside the finite set of supported conversions, the program will be of no help. Furthermore, since new formats tend to get introduced over time, a user may increasingly need to invoke conversions on the new formats that the static program does not handle.

Another solution to the conversion problem is to install a standalone program intended to handle all (or nearly all) of one's conversion needs, by supporting conversions between a large variety of formats. Some of these packages are freely available on the Net, such as the PBM package for converting between different graphics formats. Others are more general commercial products, such as Debabelizer. While these packages can be quite versatile, they still only cover a finite set of formats and conversions. A user cannot convert to or from an unsupported format without installing or buying yet another program. Coverage of domain-specific formats is often weak, and formats defined and used only locally are not supported at all. Because of the continual introduction of new formats and the number of obscure formats that one might have to deal with, these "all-in-one" standalone programs are still less than ideal solutions to the conversion problem.

TOM offers a different solution to the conversion problem that is both easy to use and indefinitely scalable, thanks to TOM's broker architecture and extendable type graph. Conversions through TOM can also be invoked much more easily than they can be invoked through installing a special-purpose program, since all conversions are invoked through the same program interface, and can be requested through a variety of user interfaces.

Challenges for a general conversion service

A general-purpose conversion service must carry out a number of tasks to be successful. Some of these tasks are trivial for a standalone program to carry out, but nontrivial in a networked system. Others can be challenging to implement properly on either a networked or a standalone system, but were often straightforward to implement with TOM, thanks to its use of brokers, its extendable object-oriented hierarchy, and other concepts borrowed from work in distributed systems.

Specifically the tasks are:

- Retrieving the object to be converted.
- Determining the format of the object.
- Determining what conversion the user wants, and whether it should apply to the entire object or just part of it.
- Planning a conversion strategy, and invoking necessary conversions.
- Presenting the results to the user for viewing, saving, or further conversion.

As we shall see in the following sections, TOM's infrastructure supports both the conversions themselves and also many of the accompanying tasks.

7.1.2 TOM conversion applications

Three conversion services were implemented using TOM's infrastructure. Each service uses a different kind of user interface, optimized for different usage patterns. The development of each new service also reflects improved understanding of the needs of conversion clients. All three services now coexist on top of TOM, and serve as examples of how TOM's infrastructure is compatible with a variety of clients.

Netconvert: small and powerful

The first conversion service developed to use TOM is called `netconvert`. Users invoke the program from a Unix command line, specify a set of documents using filenames and URLs, and choose the format to which the documents should be converted. The results are written to standard output or saved as files in a specified directory.

Using `netconvert`, a human or a program can convert multiple files with a single command. For example, to convert all the HTML files in a particular directory to plain ASCII text, and write the results in the `textonly` subdirectory, a user would type

```
netconvert -t text -w textonly *.html
```

To convert a Microsoft Word file on the Web to HTML, a user might type:

```
netconvert -f word -t html http://www.bigarchive.com/important.doc
```

(In this latter case, the -f flag tells netconvert that the Web document should be considered a Microsoft Word file. If netconvert is not explicitly told this, it might still figure it out based on other information.)

Netconvert gives users access to a wide range of conversions. At present, hundreds of conversions are composable from about 70 primitive conversions. The conversions themselves run on a variety of machine architectures, such as Windows machines or various flavors of Unix. Netconvert allows them all to be run remotely, without using the CPU resources of the client's own machine. The range of available conversions can increase indefinitely, without any changes in the program running on the client's machine, as more formats and conversions are registered on various type brokers.

While netconvert gives users access to a wide range of conversion services, the netconvert client program is extremely simple, consisting of less than 200 lines of C in its original version. (Later enhanced versions are about 700 lines.) The client code has no knowledge of particular types or conversions, and only the most basic knowledge of URLs. It simply mediates between the user and one or more type brokers, translating the user's conversion requests into a series of transactions in TOP, and uploading and downloading input and output files as necessary.

The real work, as measured by computation cycles, takes place on the type broker, and on servers invoked by the type broker. For instance, to retrieve items from the Web for conversion, brokers invoke the fetch method on URLs the user supplies. Implementing this type, and its fetch method, on any server known to TOM, immediately makes the contents of Web servers all over the world available to TOM. Once the brokers have whatever items they need, they plan and invoke appropriate conversions, using the methods described in Chapter 5, and return the results.

Implementing netconvert revealed two problems that TOM's basic infrastructure had not accommodated. The first concerned format assignment. In TOM, shipped objects are tagged with a format specification consisting of a type and a series of encodings that, when composed together, represent the type as a sequence of bytes. Objects not managed by TOM, like those served by Web servers and stored in filesystems, do not have these explicit tags, so a format must be assigned to them for use by TOM.

The second problem concerned format specification. Format information as represented in TOM can be long and complex, especially since type names themselves can be long, as we saw in Chapter 6. Ordinary users typically find it difficult and cumbersome to specify that they want a conversion to "type net:html-090594@gs1.sp.cs.cmu.edu, encoded as e:text using the standard encoding." Users would prefer to simply request "HTML", and have the system respond appropriately.

Setting up an association table for formats solved both of these problems. The table allows suffixes, human-understandable aliases, and MIME types to be associated with different formats. When requests a conversion from "html", or a file with the suffix .html is uploaded, or a document is retrieved from the Web with the text/html MIME type, the system associates these with the standard encoding of the net:html-090594@gs1.sp.cs.cmu.edu file type. Similar associations can be made with other forms of file information as appropriate, such as the "type" and "creator" tags that Mac OS uses to identify formats. The association table is recorded in a configuration file

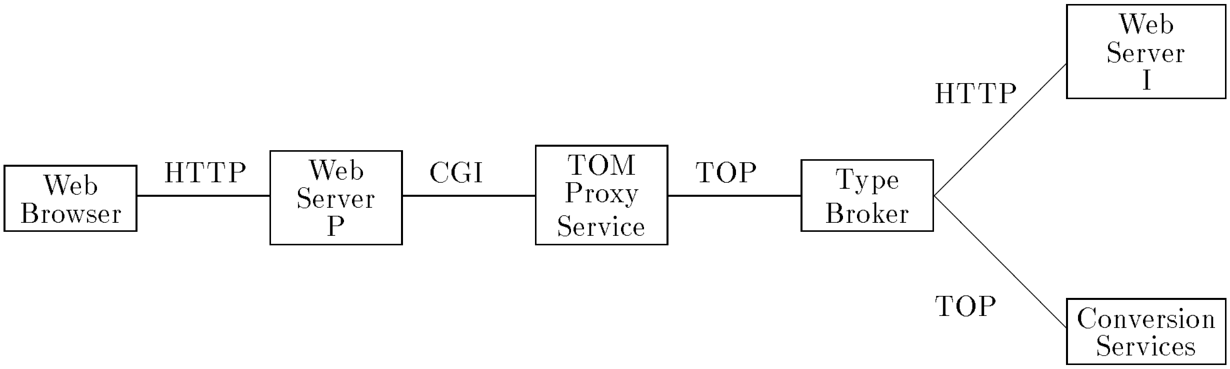


Figure 7.1: Using a TOM proxy to convert documents from the Web. The Web browser, instead of retrieving documents directly from Web server I, forwards its request to the proxy at server P. P's Web server then invokes the TOM proxy, which then uses TOM's type brokers to retrieve the document and carry out any needed conversions before passing the results back to the browser.

used by the `netconvert` program and by type brokers, and extended by the maintainers of these programs.

In current implementations, the association table is kept as a separate entity from type description objects, because this scheme allows for efficient lookups of formats from file information, and because it allows the annotation table to be easily changed to reflect local conditions. (Suffix conventions can vary from place to place and from time to time. For instance, `.tex` has sometimes been used to denote plain ASCII files, and sometimes to denote LaTeX documents.)[1]

In summary, the `netconvert` program shows that TOM allows a simple program to take advantage of the full range of conversions offered by TOM's type brokers. It also shows that it is possible to integrate a large body of existing information (such the contents of the Web and local filesystems) with TOM's functionality, and that one can add an interface to TOM that is easy for both human users and programs to invoke.

Proxy conversion service: convenient and transparent

As we have seen, the *URL* type, when defined in TOM, integrates TOM with information provided by Web *servers*. It is also convenient to allow TOM to be used by Web *clients*, so that people can take advantage of TOM's services directly from their Web browsers. The TOM conversion proxy service gives this capability to Web browsers. It retrieves documents from the Web requested by clients, converts any documents in uncommon formats to formats that most Web browsers recognize, and passes on the result to the Web client. As we see in Figure 7.1, the program is invoked from a Web server using the standard CGI interface for Web server scripts, and communicates with a TOM broker using TOP, acting as a client-side gateway to TOM.

Web users can use this conversion service either by adding a prefix to an ordinary Web URL that routes the request through the program, or by making the program's server the browser's

[1] While current implementations do not support it, it is possible for the data used in the association tables to be passed around with type descriptions. Those who use these tables, though, should be aware that associations are subject to local variance, and may need to be updated accordingly. The information one gleans from an association table can also be expressed as operations on types, and thus be integrated into TOM without any extra machinery. For example, a *suffix* type can be defined with a method that would take an object and assign it an appropriate format specification based on the suffix. Or, a type can be defined for the association table itself, allowing clients to retrieve mappings between suffixes and types based on the association table specified by the client.

proxy server. (When a browser has a proxy server, it sends all its requests for documents to that server, rather than sending them directly to the servers that have the documents.)

In either case, the conversion service transparently converts documents in unusual formats to a widely-used format, if this is possible, using TOM's conversion services. The service has its own internal list the formats considered widely usable, which is subject to change as Web browser standards evolve. If no conversion is available, the conversion service passes along the document in its original form. When the service is invoked through a URL prefix rather than as a direct proxy, it also rewrites links in the HTML documents it fetches so that they also have the prefix that makes their retrieval go through the same conversion service.

Using the conversion proxy, then, users see a wider variety of Web documents through their browser than they otherwise could. Moreover, the mechanisms that produce these benefits are almost entirely invisible to the user. At most, a user might notice that sometimes documents might take a bit longer to retrieve than usual (due to the time required for a conversion); or a user looking closely at the URL or HTML source for a document might see some unexpected additional information.

This transparent conversion service has been used in indexes like The On-Line Books Page [Ock98] to transparently uncompress books stored in compressed formats. The On-Line Books Page index allows users to choose to retrieve the books either in compressed or uncompressed formats. Most of the thousands of people who choose to view them in uncompressed formats probably never realize they are invoking special programs to read their books.

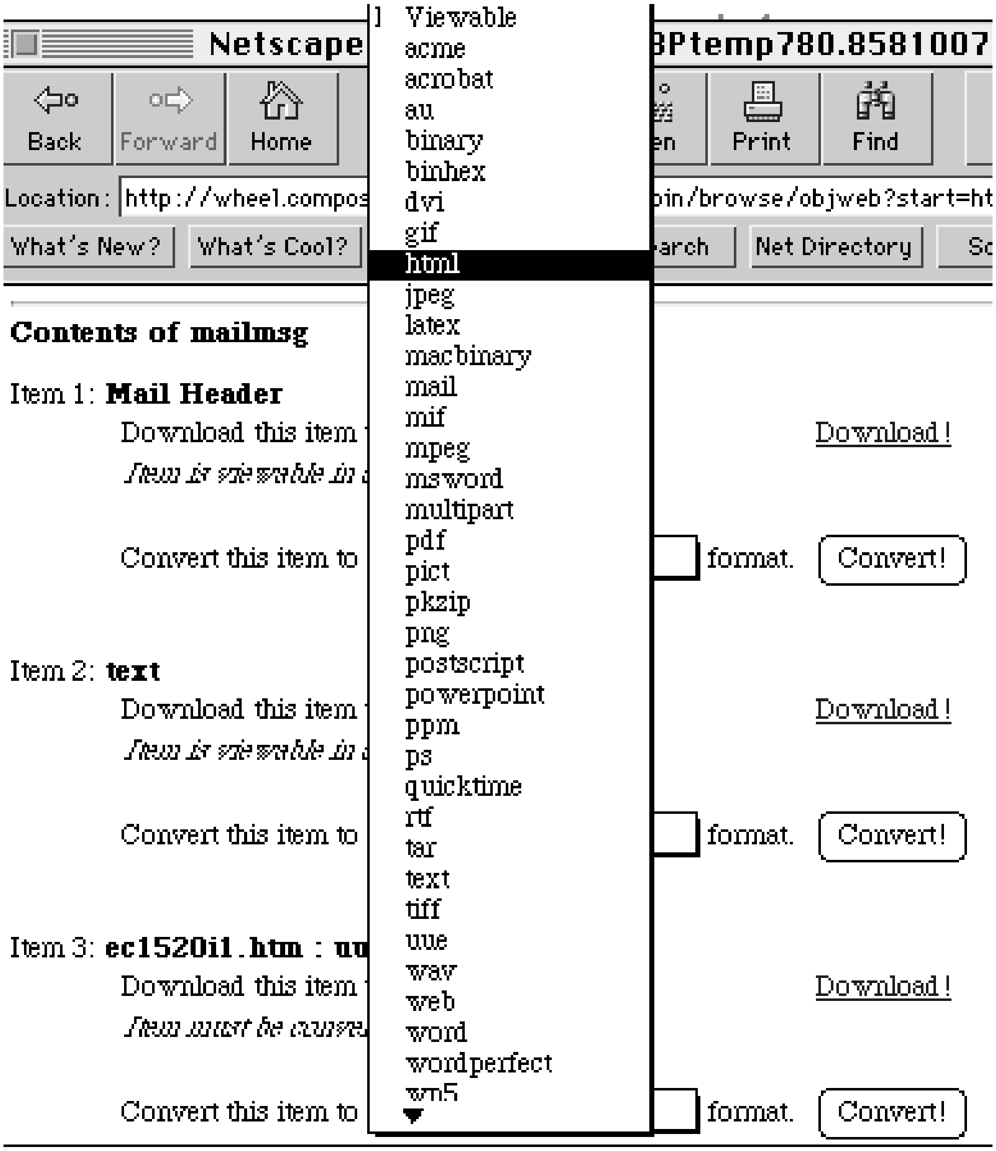
The conversion service was also extremely easy to implement using TOM's infrastructure. The program's core service is essentially accomplished using two TOP requests: one to fetch a particular URL, and another to request a conversion of an object to one of a particular set of "viewable" formats. Both of these requests are handled by the existing type broker structure. The service then needs to do little more than parse the CGI request generated by the client, and return a document that the Web server can send back to the user.

This conversion interface is convenient to users because it is virtually invisible. It is also convenient to authors of Web pages because it can be invoked simply by adding an appropriate prefix to normal URLs. However, its interface is necessarily limited, since the choice of conversion is determined by the conversion program, based on its best guess of the format the user might want to see. Users and Web page authors cannot override this choice. For those who want more control over conversion, another interface is required.

TOM Conversion Service on the Web: interactive and versatile

A third conversion service developed for TOM by the TinkerTeach project (which included the author and other computer scientists) gives users more control over conversion than the previous two services. This interactive service, known as the TOM Conversion Service, lets any user with a Web browser select a document to convert, and select the format to which they wish to convert it. In some cases, the user can also select a particular part of a larger document to convert. At this writing, it is the most popular of the TOM conversion services, and handles hundreds of conversions per week from clients worldwide. (Chapter 8 shows figures on usage.)

The TOM Conversion Service lets Web users specify a document that they want converted either by giving its URL, or by using the file upload feature implemented in Web browsers like Netscape and Internet Explorer. Users also specify a format to which they want the document to be converted, using the same mnemonic aliases used in `netconvert`. Alternatively, users can request a conversion into a "viewable" format, that is, any of several formats that can be easily



Contents of mailmsg

Item 1: **Mail Header**

Download this item

Item is viewable in

Convert this item to format.

Item 2: **text**

Download this item

Item is viewable in

Convert this item to format.

Item 3: **ec1520i1.htm : uu**

Download this item

Item must be conve

Convert this item to format.

Figure 7.2: A user mail message being converted through the TOM Conversion Service. The user can convert or download the entire message, or a particular part of the message.

displayed by most Web browsers, or that can be examined and manipulated from an HTML form.

In Figure 7.2, the user has selected a mail message to convert, and has requested that the mail message be shown in a viewable form. The TOM Conversion Service then shows the parts of the mail message, which include a mail header, a text portion, and a UUencoded HTML file. The user can download any of these parts, or convert any of the parts into another format. (In the figure, the user has a menu open to select the desired final format, and has "HTML" highlighted. Each of the segments has its own conversion menu; in the figure, the conversion menus for the different segments are partly obscured by the open menu on Item 2.) Once the user selects a format from the menu, and clicks on the "Convert!" button for the segment to be converted, that segment is converted to the requested format, if such a conversion is possible.

Interactive conversions can be quite complex, as this figure shows. When designing this conversion service, we noticed three challenges that required particular attention. The first was that many of our clients needed to convert documents like mail messages, zip files, and other documents where the data that needed to be converted was *part* of a document, rather than the entire document. The second was that the system often had to determine the format of a document without any help from the user, and without other metadata like a filename suffix or MIME type. The third was that the interface needed to be usable by novices and experts alike, since we intended the tool to be used both by computing professionals and by school teachers new to the Web.

While each of these challenges was nontrivial, we found in all three cases that the infrastructure of TOM made it possible to address them in a scalable, straightforward manner. The following paragraphs discuss how the conversion service handled each challenge, and the role that TOM played in solving them.

Packages: The first problem stems from a difference between a typical user's notion of conversion and a programmer's notion of conversion. From TOM's native point of view, if one wants to convert from a format of type A to a format of type B, one should find a conversion that will present the information defined in type A in the appropriate format of type B. Hence, if TOM is given an object of a "mail message" type and told to convert it to, say, Postscript, TOM will try to find a conversion that represents the mail message, headers and all, in Postscript. The user's intention, however, is often different. Often a user wants to convert a particular part of the mail message to Postscript, such as the body, or possibly even just an attachment, instead of the entire message with headers. In other words, rather than converting the overall *package* of information that the mail message consists of, the user really wants something *inside* the package to be converted.

Converting parts of mail messages was one of the most common types of conversion problems for our users. Our initial solution to this problem was to create new types specifically for mail messages and their segments, and add special code in the user interface to display the parts of a mail message in a structured manner. When a mail message was retrieved, the contents would be described and users could decide to retrieve or convert specific parts of the mail message, or the mail message as a whole.

We started to implement similar solutions, using TOM types and special purpose interface code, for other kinds of packaging, such as tar files and uuencoding. While this approach allowed users to interact with these kinds of documents easily, we realized that it would lead to an increasingly complex user interface, as more code would be required to handle different kinds of packaging. Not only would the interface be more complex and harder to maintain as more kinds of packaging were supported, but subtle differences in the interface for handling different kinds of packaging might confuse users.

TOM's subtyping system, however, offered an elegant solution to this problem. We defined a generic *package* type, which included attributes and methods for describing items in the package, and extracting them from their packaging. We included some derived operations so that common forms of description and extraction could be handled with a single method call. We then made mail messages, MIME multipart messages, and the other packages subtypes of the generic package type. Because they were subtypes, the conversion service and its user interface could treat all of them uniformly through the *package* interface, resulting in a more consistent user interface and less complex client code.

With the common *package* supertype in place, the TOM Conversion Service user interface does not have to know any details concerning different package types. It simply checks whether a given object conforms to the package type. If it does, and the user has not requested a conversion of the package itself, the TOM Conversion Service calls generic package routines to display the object's parts, and lets the user select which parts of the package should be downloaded or converted. (If the package contains only one item, the service assumes that the user wants that item converted, and skips the selection phase.) We see through this example how the facilities of TOM allow a program to take advantage of the facilities of many different types, without requiring advance knowledge of all the types.

Implementing this service also demonstrated how easy it was to extend TOM's type graph. Even though certain kinds of packages, such as mail messages, pre-dated the definition of the generic *package* type, the existing types could be easily made subtypes of the new *package* type, while preserving full backward compatibility with programs already using the old types. Methods declared for the *package* type that had not been declared for the earlier types were added to those earlier types as deriving from existing methods on those types. Not only were we able to integrate existing types under the *package* supertype, but we could easily add new types as well. For example, I defined, implemented, and debugged a package interface for a new *zip files* type in less than 90 minutes, much of which was spent in ensuring that the method implementations would not compromise a server's security if applied to unusual input values.

Identifying formats: The second problem we encountered arose partly as a result of solving the first. As we noted above, one of the problems faced by a conversion service is to identify correctly the formats of objects given to it. One solution, described in the discussion of `netconvert`, uses meta-data like the filename suffix, the MIME type (if any), and user-supplied hints to determine the file's format.

Sometimes this information is not sufficient. In particular, an object of a *package* type may have many items in it and little information about the formats of items in the package. (ZIP and uuencode files, for example, do not contain much more meta-information than a short filename, which often lacks an informative suffix.) Furthermore, the user may not be able to identify the format of many files, and may be reluctant to make 12 manual identifications for a ZIP file that contains 12 items.

In such cases, the system has to examine the contents of a document to determine its format. Many formats include a telltale pattern (sometimes called a "magic number"), that appears in all documents of those formats, and is unlikely to appear in files of different formats. In some cases, the pattern is a predefined sequence of bytes in a certain location early in the file. In other cases, the pattern might consist of characteristic strings that might appear anywhere in the file.

For example, standard GIF file formats always start with the string "GIF" followed by a version number. DVI files start with a fixed 8-byte sequence that is very unlikely to appear accidentally in a file of some other format. Other kinds of strings appearing in a document are usually telltale signs of particular formats. For example, the string

```
\begin{document}
```

early in a document is a good indicator of a LaTeXdocument, and strings like HTML, TITLE, and A HREF=, when enclosed in angle brackets, are common signs of an HTML document.

For the TOM Conversion Service, we represent these characteristic patterns in a notation based on Perl regular expressions, and add them to the same association table that `netconvert` uses to associate MIME types and filename suffixes with formats. Once the patterns are recorded in the table, a type broker or the conversion client itself can use these patterns to identify the format of an unknown document. In the current implementation, the TOM Conversion Service itself reads the table to identify formats. Another possible implementation would define a derived method on a *byte sequence* type to identify a format based on a table of characteristic patterns, much like the methods I proposed for identifying formats from suffixes and MIME types. When this method is implemented, programs that use TOM could call this method when they need to determine formats. Moreover, if the method implementation gains a larger repertoire of recognizable formats (as it might if the table were augmented, for instance), all of its clients automatically take advantage of this additional expertise.

Using this technique, any program connected with TOM will identify an increasing number of formats over time. Unfortunately, the amount of computation required to identify formats by examining document contents necessarily increases as more formats need to be checked. Unlike the simple hash table lookup that suffices for mapping suffixes and MIME type names to TOM formats, content-based format identification by its nature cannot be reduced to a simple lookup, because the pattern for each format is different. Each possible format for a document may require a different pattern check; hence the more formats there are with telltale patterns, the more computation may be required to identify a given format. In terms of performance, then, this technique does not scale well.

Although TOM does not make this performance problem go away, in practice the most common formats can be identified relatively quickly, by doing the checks for the most common formats, and the checks that can be done in the shortest time, before making other checks. (More sophisticated optimizations of format identification are described by Schwarz et al [SS94].)

TOM's infrastructure also allows client programs to control carefully how thorough format checking should be. It is possible, for instance, to define an identification method in TOM that takes instructions on which formats to check, which formats to check first, and how long to spend trying to identify a format before simply labeling it a byte sequence. A program could then call such a routine with whatever parameters it desired, and could even call multiple routines simultaneously. These routines are not currently implemented in TOM, and the present TOM Conversion Service does not allow users to decide how much effort will be spent checking formats (except for allowing users to specify what they think the original format is at the start of a conversion). However, the more flexible routines described above could be implemented as methods of appropriate types. TOM thus allows programs to decide how rigorous they will be in identifying particular formats, and allows different strategies to be implemented for identifying formats from object values.

User interface: The success of a general-purpose application can depend as much on its user interface as on its underlying functionality. Much of the time and effort in developing the TOM Conversion Service was therefore spent in developing an appealing and easy-to-use user interface for conversions. While the full details of the user interface and its development are beyond the scope of this thesis, I note here two aspects of TOM's infrastructure that played an important role in the development of the user interface.

One aspect was the well-defined, modular interface that TOM provides through its type brokers. As we have seen from the various services described in this chapter, the basic architecture of TOM allows a wide range of user interfaces to be placed on top of it. The eventual interface for the TOM Conversion Service, then, was developed and modified without being dependent on implementation details of TOM's type brokers, and similarly the interface and implementation of TOM's brokers and types could be augmented without worrying about the details of the user interface.

In addition, the TOM Conversion Service exploits a number of user's concepts that proved to be easily implemented, or already implemented, in TOM's infrastructure. The generic *package* type described earlier in this chapter easily handles a general notion of "packages" that enables users to deal with various container types uniformly. The association tables TOM keeps for different formats make it easy to maintain mnemonic names and characteristic patterns for formats. Also, TOM's ability to search for a conversion from a given format to any of a set of desired formats makes it straightforward to implement the most common conversion request made by users: "convert this document into any viewable format".

7.1.3 Review and retrospective

The various conversion services implemented on top of TOM illustrate TOM's versatility, its feasibility for solving significant problems, and its scalability. We see how TOM can easily be composed with information services like the Web, and with a variety of clients with very different user interfaces.

We also see many of TOM's features being used advantageously, including conversion planning, subtyping (in the packaging example), the ability to augment the type graph, and the ability to add derived methods to types (also in the packaging example).

As we enriched the local suite of conversions, we also found that TOM scaled up beyond a single-person project without much difficulty. While I had defined the original TOM infrastructure, had implemented the broker, and had defined the first types, we were able to bring in an undergraduate to implement user interfaces on top of TOM, define new types, and bring in new conversions as needed. At the time of this writing over 100 types have been defined on two type brokers, and over 70 conversions defined on them. Types, their interfaces, and their conversions were defined and implemented quickly and easily. Many of the conversions were "off-the-shelf" tools, and others were specially written for TOM, but TOM's server configuration files allowed both kinds of conversions to be easily integrated into TOM, for the most part. We did experience some difficulties, however, with formats that were proprietary and only parsable by vendor-supplied programs. We will discuss this issue in more detail in Chapter 8.

The ability to bring knowledge and functionality incrementally into TOM makes it superior in functionality and ease of use to the more monolithic conversion techniques surveyed at the start of this chapter. Users no longer have to find and install standalone conversions to convert files; rather, they can request the conversion they want, and if the appropriate conversions have been installed anywhere and registered with a TOM type broker, the broker will automatically invoke the appropriate conversion programs. Likewise, users do not have to settle for a fixed set of conversions that a particular standalone conversion program supports; rather, TOM clients automatically take advantage of the additional types, operations, and conversions that are added to TOM's global type graph.

7.2 Frame Services

In this section, I look at an application that illustrates how TOM can be used to introduce new types of data into an information system, raising the level of abstraction in the system while still being compatible with existing programs in the system. The data type in question, known as a "frameset," was first introduced into the Web by Netscape, a well-known Web browser developer. I compare how the Web community adapted, or failed to adapt, to this new data type, to how TOM handles the same new data type. This comparison shows several advantages of TOM over conventional methods of introducing new data types.

7.2.1 Raising the common denominator for data

In the introduction to this thesis, I noted that current information systems require data providers to choose between data formats with complex structures and those that are widely understood. Many information clients cannot parse data in complex or uncommon formats. Therefore, much information is made available only in a "lowest common denominator" format, such as plain ASCII text. While such formats are widely readable, documents in those formats often lack important information. For example, an ASCII rendition of a French novel may be missing accent marks, typesetting, and illustrations. Sometimes the missing information is structural: for example, a text representation of a statistical table may show all the elements of the table in a human-readable format, but a statistical analysis program may be unable to analyze the text in the same way that it could analyze a spreadsheet file.

As we have seen, TOM allows more complex structures to be communicated on the network, and adapted to local needs as necessary. A French novel, for example, can be made available as a Microsoft Word document, complete with accents, layout, and pictures. Clients that can parse Word documents can display it directly. Simpler ASCII-based clients can convert it to plain ASCII text, through services like the TOM Conversion Service.

Data in many other formats, some of them more specialized, can also be found on the Web. For example, the US Census Bureau maintains mapping information in a specialized format known as TIGER. Library catalog records, likewise, are typically stored in a specialized format known as MARC. These data formats are generally presented to Web clients in a simpler form, such as a map image, or a text rendition of a library card. However, the original form of the data includes substantial information that is not apparent, or is harder to recognize, in the simpler form. If an abstract interface to this data were made available, clients could use it to access the data in a convenient form through this interface.

One might question whether this ability to accommodate the interface of new formats is a major concern to most network users. Perhaps conversion of such formats to familiar ones is enough. How likely is it that a new data format will come along will require Internet clients to adapt to its interface, instead of just being convertable to a familiar format?

In the short history of the Web, at least one common data format of this sort has already arisen: the *frameset*. Framesets, introduced by Netscape, are now used on many Web sites, and have introduced a host of usability and compatibility problems that TOM largely averts. In the next section I describe the impact of frames on the Web, and the usability, compatibility, and portability problems they created when they were introduced. Then, I illustrate how TOM types for frames and framesets solve these problems.

7.2.2 The framing of the Web

World Wide Web pages were originally designed to be viewed one at a time. Web browsers normally displayed a single HTML document, sometimes accompanied by embedded images or other subordinate objects. When the user selected a link, the previously-displayed HTML document would be replaced by a new document.

In 1996, Netscape introduced *frames* in release 2.0 of its Web browser, Netscape Navigator. When looking at a Web site that uses frames, their browser displays multiple HTML documents in different regions of the browser window. Clicking on a link in one region might change the document displayed in that region, or it might change what is displayed in some other region, or in the entire window. Many Web sites soon used frames to enclose their data, often for information that could just as easily have been presented unframed.

Frames enhance the presentation of some forms of data in three important respects. First, they can make important information persist on-screen. For example, a persistent navigation bar can make it easier for users to return to key locations of a Web site, without the need to hit the "back" button multiple times or scroll to a link at the top or bottom of a long document. Some sites also use persistent frames for advertisements or important messages,

Second, frames allow controls to be placed around a complex object, so that users can easily change the presentation of the object, or operate on it in other ways. A good example of this use is the Universal Library's Book Object [Thi96], which supports several navigation and presentation controls for on-line books. In the Book Object, a central frame displays a book page, and surrounding frames show tables of contents, and allow users to turn the page, jump to a particular book section, or change the way the pages are displayed.

Third, frames allow users to view data through multiple perspectives simultaneously. For example, a scholarly site on Dante can display Italian text in one frame, an English translation in a second frame, and commentary on the text in a third frame. Users can scroll through each of these displays as they see fit. Frames even allow diverse data sources to be displayed together without the cooperation of the original sources. For example, the "McSpotlight" Web site [McS98] uses frames to display the McDonald's web site side by side with criticism of the restaurant's on-line marketing.

Netscape's frame design, however, also has several problems. First, frames are incompatible with older Web browsers that were designed for displaying only one HTML document at a time. When those browsers encounter a page with frame directives, the directives are ignored, and the contents of the frames do not appear.

Although Netscape's frame specification includes an area where Web page authors could include alternate HTML for frameless browsers, maintaining this alternate area requires extra effort on the author's part to duplicate the information given in the frames. Many Web sites, therefore, do not use this alternate area. Viewers who visit such sites without frames enabled see only a blank screen, or are told that they should "upgrade" to Netscape, with no other way provided to view the site. Users on machines that cannot install the new, resource-hungry versions of Netscape, or that have special needs (such as visual impairments) that require them to run other browsers, are left without access.

Second, Netscape's own browsers do not support the same sorts of interactions with framed documents as with normal documents. Neither Netscape 2 nor Netscape 3 supports printing, bookmarking, and navigation of framed Web pages as well as they supported these operations on unframed documents. In Netscape 2.0, when users navigate through a frameset and click on the "back" button, the browser does not return to the previous state of the frameset, but returns to the document that had been displayed before the frameset was entered. This behavior was widely criticized, and version 3.0 of Netscape lets users go back through previous frameset states. Version 3.0, however, still does not allow users to bookmark a particular frameset state, or to mail this state to someone else. At best, the user can record the URL of a particular frame in the frameset, or the URL for the start of the frameset. Netscape also imposes other limitations on framesets; for example, version 3.0 allows users to print out a single frame, but has no facilities for printing out the entire contents of a browsing window composed of several frames.

One of the sources of these problems is that Netscape has not publicly defined a representation for the state of a frameset. Netscape 3.0 evidently added an internal representation, since its "back" button, unlike that of Netscape 2.0, can return to previous frameset states. However, it still failed to define any way to represent a frameset state externally. Therefore, Netscape 3.0 lacks any way to save a frameset state in a bookmark, or mail it to another user.

Netscape's proprietary definition of framesets continues to create compatibility and usability problems on the Net. Framesets were not incorporated into official WWW standards until December 1997, in part due to concerns and disagreements over the design of frames. In the meantime, Microsoft, a rival Web software company, declared its own frameset extensions to the World Wide Web, which are subtly different from Netscape's. With two rival proprietary standards, a longtime lack of a universally agreed-upon standard, incompatibility with older browsers, and lack of a well-defined way to represent and work with frameset states, the introduction of frames into the Web created chaos and disorder. In response to charges that they were disrupting the Web, Netscape and Microsoft replied that they had to extend the features of the Web as they did. Waiting for a frame standard to be approved by an official standards body, they said, takes too long, so instead they acted unilaterally to meet the demands of their customers.

7.2.3 TOM types for frames

The problems raised by the introduction of frames to the Web can be averted through the concepts and facilities of TOM, by treating framesets as a new type of structured data. A *frameset* data type, with an appropriate text encoding, makes it possible to represent framesets in a way that can be bookmarked and communicated. Methods on framesets allow users to navigate through them (with the appropriate updates made when a link is selected), manipulate their layout, or convert them to a form compatible with frameless browsers. With the appropriate registrations on type brokers, TOM-aware browsers can call these methods to display and manipulate framesets, even if they were not written with any knowledge of frames. Even a browser that has no knowledge of TOM (or of frames) can work with frames, through a proxy or gateway to TOM's services. (I have built such a gateway, and describe it below.)

Furthermore, because TOM allows anyone to define new types, anyone who wishes to propose their own definition of frames can register a specification and services for their definition with a type broker. They can do this without the cooperation of Netscape or any other frame developer. Conversions and subtyping make it possible to reconcile multiple frame definitions and representations.

Frame type definitions

Figures 7.3 and 7.4 show the definitions for the *frame* and *frameset* types. The essential parts of the definitions are as follows:

- A *frame* object has two essential attributes: a `name`, consisting of a text string (which is

```
NAME net:frame-012098@tom.cs.cmu.edu
STATUS X
ADMIN {
CONTACT spok@cs.cmu.edu
}
SUPER e:obj
SEM
"This type represents the basic information stored in an HTML frame."

ATTR e:text name {
SEM
"The name given to the frame.  If none was given, this is an empty string."
}

ATTR e:bool hassubframes {
SEM
"True if (and only if) there are subframes."
}

ATTR s:url contentref {
SEM
"The URL that points to what should be the contents of this frame, if any.
 Undefined if hassubframes is true."
}

ATTR net:frameset-012098@tom.cs.cmu.edu subframes {
SEM
"The frameset contained by the frame, if any.  Undefined if hassubframes
 is false."
}

OPER e:obj content {
SEM
"Get the content of this frame.  This is a method, rather than an
 attribute, because the contents might change if the server pointed to
 by contentref changes its state."
DERIVED
"This can be computed by falling fetch on contentref.  It can also be
 cached."
}

ENC e:text tfs {
SEM
"The standard representation of name and URL given in the TOM Frame Service.
 It can be embedded in a URL."
}
```

Figure 7.3: The definition of the frame type.

```
NAME net:frameset-012098@tom.cs.cmu.edu
STATUS X
SUPER e:obj
SEM
"This type represents the basic information stored in an HTML frameset."

ATTR c:seq:net:frame-012098@tom.cs.cmu.edu subframes {
SEM
"The frames contained by this frameset, in order.  Note that nested
 framesets are assumed to appear in an (anonymous) frame."
}
ATTR e:text rows {
SEM
"A specification of the row layout, if any, represented in accordance
 with the W3C specifications.  If there is no row layout, this is an
 empty string."
}

ATTR e:text cols {
SEM
"A specification of the column layout, if any, represented in accordance
 with the W3C specifications.  If there is no column layout, this is an
 empty string."
}

OPER net:frameset-012098@tom.cs.cmu.edu makeset {
SEM
"This generates a new frameset.  If the object pointed to by 'ref' includes
 a frameset, that frameset is returned.  Otherwise, this routine generates
 a frameset with a single, anonymous, full-sized
 frame whose contentref is 'ref'."
CLASSOP
ARG s:url ref
}

OPER net:frameset-012098@tom.cs.cmu.edu update {
SEM
"This generates a new frameset, the same as the old one but with the
 contentref of one of the frames updated to have newref as its value...."
ARG c:seq:e:int from
ARG e:text target
ARG s:url newref
}

ENC e:text tfs {
SEM
"The standard representation of the frames given in the TOM Frame Service.
 It can be embedded in a URL."
}


CNVT tohtml32 {
TYPE net:frameset-012098@tom.cs.cmu.edu
ENC e:text tfs
TYPE net:html3.2-012098@tom.cs.cmu.edu
ENC e:text standard
}
```

Figure 7.4: The definition of the frameset type (with some semantic information removed to save space.)

empty if the frame is not named), and a `contentref`, a URL that references the contents of the frame. (From this latter attribute, one can also derive the `content` method, which returns the actual object referenced by the `contentref` URL.) Frames also have a couple of attributes that specify nested subframes, if any. The generic frame type shown in the figure does not have any attributes related to the way it is displayed, but subtypes might have additional attributes specifying border widths, scrolling properties, and other display-related features defined in Netscape's and Microsoft's frame definitions.

- A *frameset*[2] object includes a sequence of frames (which can be operated on like any other sequence). A frameset also has attributes that specify layout, such as instructions on whether the frames are laid out horizontally or vertically, and on the amount of display space to be allocated to each frame. Again, subtypes of the generic frameset type may specify more specific layout and presentation instructions.

  The `update` method on framesets generates a new frameset that would result from a user selecting a link in one of the frameset's frames. The method takes as arguments a reference to the frame in which the user selected a link, the URL of the link, and the name of the target frame specified in the link. It returns a new frameset with the target frame updated to contain the document referenced by the link's URL.

  Framesets also have one class method, `makeset`. This method takes a URL and returns the frameset specified in the object referenced by the URL. (If that object does not include a frameset, then the method returns a frameset with a single frame containing the referenced object.)

I have defined *encodings* for these types to allow frame and frameset objects to be transmitted, or stored in bookmark files or other locations. These encodings were defined by me, not by Netscape. If Netscape publicized its own internal frameset representation, that representation could be registered as an alternate encoding.

One particularly useful operation on framesets is a *conversion* from framesets to non-framed versions of HTML. The conversion `tohtml32` displays the contents of framesets in a form understandable to browsers that do not handle frames, by "unrolling" the frameset into a single HTML 3.2 document, with the contents of each frame in the frameset displayed in sequence. The conversion rewrites the hyperlinks in the resulting document so that selecting a link calls the `update` method of the frameset with the appropriate arguments, in order to return a properly updated frameset view.

The TOM Frame Service

To demonstrate how these types and operations for frames make them more widely usable, I implemented the Tom Frame Service (TFS), a program that uses TOM to make framesets on the Web accessible and bookmarkable in all World Wide Web browsers.

Web sites viewed through the TOM Frame Service have their frameset structure "unrolled" into a single HTML document, using the `tohtml32` conversion described above. Unrolling the frames allows any Web browser, even those without frame capabilities, to view them. When a user clicks on a link that would update one or more of the frames, TFS calls `update` to calculate the appropriate

[2] Strictly speaking, a frameset contains an ordered sequence of frames, not simply a set of frames. The term "frameset" is used here because it is the term Netscape and Microsoft use in their HTML extensions.

change to the frameset, retrieves any new documents, and then calls the tohtml32 conversion to update the "unrolled" display appropriately.

The complete specification of the frameset's contents is encoded in the URL used to access the document. Hence, if a TFS user bookmarks this URL, and later reloads it, the user will see the same frameset state.

Like the TOM Conversion Service, TFS is a CGI script invoked by a Web server when users request certain URLs. A typical TFS user starts from an opening page with a form to specify the URL of a framed page they would like to see. Authors of Web pages, if they wish, can also construct URLs that, when resolved, automatically show any frameset state "unrolled" through TFS.

Behind the scenes, TFS constructs its HTML displays by calling operations on the *frame* and *frameset* objects, as described above. TFS constructs a frameset by calling makeset, which parses an HTML document with frame directives. The frameset is then converted to an HTML 3.2 document. The conversion rewrites targeted links in the framed documents so that they go through TFS, which then invokes the update method on framesets to calculate the new frameset that would be created by clicking on the link.[3]

TFS includes two additional convenient features. As part of the conversion of framed documents to frameless HTML, other HTML extensions commonly used in framed documents are also converted to more portable forms. For example, when a client-side image map (another Netscape-authored extension to HTML) is encountered, TFS writes out the links of the map in textual form (after writing out the original image map), so that users with non-graphical browsers, or browsers that do not parse client-side image maps, can use them. TFS also gives users some control over the display of framesets. For example, if a user is not interested in particular frames, he can select links to hide them from view, and redisplay them later if desired.

I wrote the initial implementation of the TOM Frame Service in only a few days. The implementations of operations on frame types and the CGI/Web interface consist of about 1000 lines of C code (not including code already in the generic TOM server kernel). TFS is now being used for access to popular sites by users worldwide. For example, I have used TFS to create an alternative unofficial interface to the heavily frame-dependent Vatican Web site. This interface is used hundreds of times per month. TFS could be rapidly developed because it rested on a sizable infrastructure already in place for TOM and common TOM types. In particular, fetching Web documents and parsing HTML documents were already well supported by operations on URL and HTML types.

Enhanced Web browsing with TOM

The TOM Frame Service allows Web browsers with no prior knowledge of frames to take advantage of the new frame data structures. It allows users not only to *view* framed sites in ways that their browsers can display, but it also allows them to *interact* with the sites by clicking on links to update the framesets appropriately. TFS does not require any changes to the Web browsers, and takes advantage of types and operations that have been defined in TOM's type brokers and servers. The same principles used by TFS can be used to allow Web browsers to deal with other kinds of structured data that are introduced on the Web, such as the TIGER map data and MARC catalog records mentioned earlier in this chapter.

We see, then, that TOM allows a wide variety of static data structures, such as various kinds of

[3] To optimize for speed, the current TFS implementation does not actually call these operations via the TOP protocol, but directly calls the same routines that would be invoked if the operations were requested through TOP.

frames, to be added to the Web and viewed through ordinary Web browsers. Web browsers would be even better equipped to handle new data types if they could send and receive TOP messages as well as HTTP messages, for two key reasons. First, the browsers would no longer have to use a particular gateway (like TFS) that might be subject to failure or overloading. Second, the browsers could accommodate new data structures with absolutely no changes in the software, or other user intervention, required.

Currently, users wanting to view a site with its frames unrolled have to know about the TOM Frame Service, or follow a link created by someone who knows about the TOM Frame Service, and go to a particular Web location where the HTTP gateway to TFS can be found. If this gateway moves, or otherwise becomes unavailable, users are left without its services. Likewise, if the gateway becomes too popular, it can become overloaded, and become less useful for everyone.

If, however, Web browsers did not use a gateway, but rather issued TOM commands directly, they would no longer be dependent on a particular gateway location. Instead, Web browsers would simply ask any nearby type broker to convert framed sites into a recognized version of HTML, or to update a frameset after a framed link was selected. The broker would then invoke any agent that was known to carry out the appropriate operation. In this manner, the use of frame services could be distributed throughout the Internet as a whole, and remain usable even as particular frame servers go off-line or move.

Adding TOM capability directly to browsers would also allow them to automatically handle new kinds of data structures without intervention by either users or programmers of Web browsers. Suppose that someone defines a new data type *XYML*, and wants to make XYML documents widely viewable on the Web. The definer registers the type with a type broker, along with various operations on the type, including a conversion from XYML to a standard version of HTML. When a TOM-aware Web browser encounters this new type, it would have no idea what XYML is, or what can be done with it. But it would know how to work with HTML, so it would ask a type broker if the XYML document can be converted to HTML. The broker would find an XYML to HTML conversion, invoke it, and returns the result to the Web browser.

Adding TOM capability to Web browsers would make it easy for people with relatively modest hardware and software resources to browse a rich variety of Web resources, and would also make it relatively inexpensive for information providers to introduce new data types that would be widely viewed. Instead of continually upgrading one's Web browsers to handle an ever-growing variety of document types (where each upgrade is often megabytes larger than the last), one can simply add a module that understands TOM. A TOM-aware conversion module would be on the order of about 1000 lines (like the `netconvert` program described at the start of the chapter), and would use the net-wide infrastructure provided by TOM's type brokers to make a wide variety of information viewable. Likewise, the inventor of a new data type simply needs to register the type and an appropriate set of operations with a type broker, and put an implementation of the operations on at least one server. At that point the type becomes usable by any TOM-aware browser without any need to persuade users or browser vendors to adopt a new upgrade or plugin.

7.2.4 Summary

In this section, we have seen how TOM makes new data formats easy to introduce into a larger information system. Specifically, I showed how, using a TOM, one can define a new data type (the frameset) and make it widely usable both by new and by existing programs. I explained how the frame and frameset data types were defined, and showed that one could quickly build a working system that allowed existing Web browsers to view and navigate through objects of these types.

I noted that different definitions of frame types and formats could be defined simultaneously, and used widely without having to wait for standards bodies or even manufacturers. Finally, I explained how TOM-aware client programs could deal not only with frames, but also any number of other new data structures that might be invented, in a simple and seamless manner.

TOM thus allows one of the key promises of the World Wide Web to be realized: the ability to widely disseminate information in expressive but also widely-usable forms. Originally, the Web addressed this issue through adopting common standard formats, such as HTML, and the URL scheme of references. Over time, however, the variety of formats on the Web has increased, and now includes many exotic and vendor-specific formats like PDF and VRML. While these formats make Web documents more expressive, they also make users increasingly dependent on the whims of specific software vendors, and obligated to install ever-larger suites of programs and plugins. Through its facilities for defining new types and services on these types, TOM makes it possible to break this dependency. The TOM Frame Service is a working example showing how such facilities can be used to widen the range of information available to all Web browsers.

Chapter 8

Evaluation

In this chapter, I summarize the findings that support my thesis claim, by describing my analyses of the TOM design, and providing data on the use of TOM in practice. (Some of this data also appeared in the previous chapter.) I show how TOM satisfies its key requirements of expressiveness, composability, and scalability (as defined in Chapter 1) in ways superior to other systems in use. As I discuss how TOM satisfies the requirements of the thesis, I will also revisit five key design decisions of TOM, evaluating their benefits and drawbacks: immutable objects, strict subtyping, semantic declaration policy, naming policy, and type evolution policy. Following this discussion, I briefly explain how TOM remains viable amidst other systems and developments in the dynamic, ever-changing Internet community.

The principal claim of this thesis is that TOM allows a wide variety of data structures to be defined and used throughout the Internet, and that TOM's design provides significant advantages over existing systems in terms of expressiveness, composability, and scalability. Furthermore, TOM is a feasible system to implement and use, as I have shown through my implementation of type brokers, and through the use of TOM-based applications described in the previous chapter.

8.1 Expressiveness

The *expressiveness* of object-oriented systems can be measured along two important axes: breadth and depth. The *breadth* of a type system refers to the range and variety of types and formats that a system can describe and use. The *depth* of a system refers to the amount of detail that can be associated with the description of a type or format, and the range of services that can be supported for any given type.

8.1.1 Breadth

With respect to breadth, TOM supports a wider range of types and formats than does MIME, and it supports a wider variety of data representations than do common network-based object-oriented systems like CORBA, OLE, and Java. At present, there are more than 100 types defined on the two TOM type brokers operating locally. Most types have one or more encodings, which can be composed to produce thousands of formats.

I first compare TOM with MIME, still the most popular convention for describing structured data on the Internet as of early 1998. While there are currently fewer TOM types than there are MIME formats (157 MIME "types" were registered as of late August 1997 [Int97]), about 20 of the more commonly used MIME formats have counterparts registered with TOM type brokers, and

```
MIME TYPE NAME:
application

MIME SUBTYPE NAME:
slate

REQUIRED PARAMETERS:
version
        value of parameter is a text string

OPTIONAL PARAMETERS:
none

ENCODING CONSIDERATIONS:
Since BBN/Slate documents may contain binary data, transfer encoding will
normally be Base64

SECURITY CONSIDERATIONS:
BBN/Slate documents can contain enclosures which may be executed by the
recipient.  These enclosures can include executable programs or input to
interpreters like the shell.  The user is queried to assure that they really
want to execute the enclosure and the documentation warns about the dangers
of "mail bombs".  Users are cautioned to exercise care when executing
enclosures received through the mail.

PUBLISHED SPECIFICATION:
The BBN/Slate document format is published as part of the standard
documentation set distributed with the BBN/Slate product.  It is available
from:

        BBN/Slate Product Manager
        BBN Systems and Technologies
        10 Moulton Street
        Cambridge, MA 02138
```

Figure 8.1: A MIME registration for BBN/Slate documents

```
NAME mime:application/slate
SUPER e:obj
STATUS X
SEM
"The values of this type contain information found in BBN/Slate documents,
 as specified in the MIME Content-type 'application/slate'.  Published
 specifications of this information are distributed with BBN/Slate products."
ATTR e:text version {
SEM
"A version of the BBN/Slate document format."
}
ATTR e:text body {
SEM
"A byte stream that encodes the information in the Slate document,
 in accordance with the BBN/Slate document format specified by the
 version attribute."
}
ENC e:byteseq mimeheaded {
SEM
"A MIME-compatible header that includes the version attribute value as the
 version parameter value of the application/slate Content-type.  After the
 header comes the body attribute.  If there is a MIME encoding specified,
 the body attribute is transformed using that MIME encoding."
}
```

Figure 8.2: A TOM type corresponding to the MIME registration for BBN/Slate documents. Note that it says very little, so as not to go beyond what the MIME registration says.

the remainder can also be registered with little additional work per type. Specifically, one would record the abstract information specified in a MIME Content-Type specification in the TOM type's signature and type semantics, and record the concrete syntax of the MIME type in the semantics of one of the TOM type's encodings.[1] An example of a simple MIME format registration can be found in Figure 8.1 and its TOM equivalent can be found in Figure 8.2. MIME's limited range of Content-Encodings can also be expressed as TOM encodings in a straightforward fashion.

On the other hand, many TOM types have no MIME counterpart. In some cases, it would be easy to register a corresponding MIME type. However, in other cases a TOM type cannot be represented in MIME terms. For example, TOM's generic virtual supertypes cannot be defined in MIME, because MIME only describes concrete formats, and does not handle virtual types.

Compared to TOM, the design of MIME is inherently limited with respect to breadth in three important respects. First, MIME formats must be centrally registered or placed in their own uncontrolled "experimental" namespace. This requirement tends to prevent new formats from being registered and defined widely until they have gained widespread or prolonged use. TOM types, in contrast, can be publicized at any time from any type broker, and refinements can be revised and extended later. Second, MIME can only use a limited, fixed set of encodings. Specifically, MIME objects appear either in a standard encoding specified in its definition, or in one of a few standard transformations of that encoding such as `base64` or `quoted-printable`. A MIME type cannot easily encompass other sorts of encoding, or multiple levels of encoding. Third, MIME types are tied to a particular format even when this is not appropriate. MIME types must have one standard encoding, and there is no way to define a MIME type with no encoding. Nor is there

[1] While TOM types are not explicitly parameterized in the way that a MIME Content-Type can be, Content-Type parameters can be mapped into additional attributes on the TOM type, or, alternatively, different common parameter values on the same MIME type could correspond to different TOM types. Some other object-oriented systems have explicitly parameterized types; TOM does not.

any convenient way of defining two significantly different encodings of the same type, except by defining two distinct MIME types. Hence, MIME makes it difficult for clients to uniformly handle formats that express the same information but represent it in different ways. TOM, on the other hand, allows a common abstract interface for multiple formats.

TOM has advantages in breadth, not only compared to MIME, but also compared to other popular object-oriented systems that operate over the Internet, including Java, CORBA, and COM. While all of these systems support rich abstract interfaces, each lacks breadth in at least one of three aspects: the data representations they can use, the data representations they transmit, or the implementation languages they support.

Java, for example, is limited in the range of both its data representations and its implementations. This is because Java objects need to be both described and implemented in the Java programming language. Objects and operations implemented in other languages, or data not originally conceived as objects, cannot be easily imported to Java's object system without the use of wrappers and gateways. Java's "serialization" facilities allow programmers to implement mappings between formats not native to Java and Java objects, but do not easily accommodate multiple formats of the same type, or implementations that use non-native formats.

Microsoft's COM objects are also limited in representational range, since they are assumed to be represented in a particular binary format, and cannot easily be used for information represented in other formats. COM's implementations, however, can be written in any language that supports the COM API.

CORBA has somewhat different limitations. In theory, CORBA objects can be represented in any format compatible with a CORBA implementation, which can in turn be operated on in any language. However, CORBA's support of representations *that can be transmitted between machines* is limited. In CORBA, objects themselves do not migrate between machines; instead, CORBA defines a "tuple" format for transmitting structured data. These tuples can in theory be used to transmit the information stored in objects, but CORBA itself provides no standard way to transmit this information or to specify how it is transmitted. Two applications using CORBA to transmit objects must agree on an ad-hoc tuple form or migration protocol to use. They cannot take advantage of the infrastructure that TOM provides for transmitting objects in specific formats.

Design decision revisited: Immutable objects

A fair comparison of breadth, however, must note that systems like COM and CORBA support types for mutable objects, and TOM does not. Early in the design of TOM I decided to treat objects as values, rather than as variables (or "containers of values"). This decision reflected TOM's focus on information access and analysis, rather than on information maintenance and management. (This approach is not unique to TOM. MIME also defines its Content-Types by the information they express, not by how or where this information is stored.) This object model prevents users from modifying data (as opposed to accessing it). TOM by itself, then, cannot be the basis for remote database applications that include data modifications. It is not, therefore, a general-purpose object-oriented system like CORBA or OLE. At the same time, TOM still accommodates state changes through the use of references that are not always bound to the same value.

I decided to treat objects as values in order to minimize the requirements on information used by TOM, to make it easier to add and verify substitutable subtype relations, and to make TOM as easy as possible to compose with existing systems, data structures, repositories, and clients. Because TOM is designed primarily for information discovery and analysis applications, it usually suffices for TOM to be able to read data, and not manage how it is stored or changed. Limiting

TOM to this domain simplifies its object model, and thereby makes it more likely to be compatible with existing practices.

TOM servers get data from a variety of sources, including web servers, database programs, gateway services, and even email messages. To operate on this data in a read-only fashion, it is sufficient to know how the data is structured. TOM can then convert the data into more convenient forms as needed.

An object model that handles both reads and writes in a general fashion puts more demands on clients and servers. For TOM to successfully handle fully mutable objects, TOM would also have to understand and control the servers that manage the objects. In order to keep read-write semantics consistent, a TOM server would have to keep track of where an object was stored and how it migrated, so that it could make changes consistently. It would also have to be able to manage transactions, possibly over multiple servers, to carry out a mutation operation cleanly. It might have to make subsequent mutation operations serializable, and edit a transformed object correctly through views, if a "converted" object were changed. Not only would these challenges require that data servers be directly controllable by TOM's infrastructure, but many of these challenges, such as the editing-through-views problem, require highly complex (and sometimes highly time-consuming) distributed system algorithms.

Fully mutable objects would also restrict the flexibility of the type graph. For instance, it would be significantly more difficult to establish subtype relations, since subtypes would have to preserve not only the static properties of their supertypes, but also their history properties. (Liskov and Wing [LW94] discuss this problem in more detail.) The potential to take advantage of supertype interfaces for unfamiliar types, or to guarantee the preservation of information in conversions with intersubstitutable types, would be significantly lessened in such a system,

TOM's treatment of objects as immutable values, then, makes it possible for type definers to build rich subtyping structures, and allows a wide variety of resources to be clients or servers for TOM (either directly, or through gateways).

The decision to treat objects as immutable values has a cost, though. Many networked information systems are not completely read-only, but also allow the user to modify data to a limited extent, or affect the outside world. For example, it might be useful to allow users of a travel information system to make reservations. Or, a phone directory might give users the ability to change their phone numbers or addresses, or make their numbers unlisted. While TOM could model the data structures of these information systems, and provide useful conversions, searches, and other analyses on the information provided by these systems, it could not make reservations or change directory information on behalf of users. An auxiliary program might be able to perform these tasks, but TOM itself could not. This may make application programmers reluctant to use TOM, since they could use it for only part of their task. On the other hand, we will see later in this chapter how it is possible to combine TOM usefully with more general-purpose object systems, and reap the benefits of both.

8.1.2 Depth

TOM's depth of expressiveness is also superior to both MIME and common Internet-based object-oriented systems. Unlike MIME, TOM directly supports invoking conversions and other operations on objects. TOM also supports semantic descriptions and subtyping to a degree not supported by other networked object-oriented systems.

TOM, unlike MIME, supports a clear distinction between an object's representation and its interface. MIME's description of formats focuses largely on how the format is represented. It may also describe what can be done with the format, but it does not provide any mechanism for automatically invoking an operation on a MIME object. TOM, in contrast, supports an explicit interface for every type, including signatures that allow clients to automatically invoke operations on objects. TOM's separation of interface from representation allows TOM to be more flexible than MIME in working with objects with similar functions that are represented in different formats. TOM also has a more expressive subtyping model than MIME's. MIME's single-level "subtyping" is primarily an organizing convention, with little semantic import. TOM's subtyping guarantees are much richer, as we saw in earlier discussion of substitutability.

Systems like Java, CORBA, and OLE, have deeper semantic support than MIME. Like TOM, these systems provide signatures for object types that allow operations to be invoked on the objects, including operations that mutate the objects (which TOM does not support).

However, these systems do not provide much semantic support beyond signature information. TOM, unlike these other systems, also allows information about the operational semantics of a type and its operations to be provided, as described in Chapter 4, and makes this information widely available through its type broker network. Omitting this information can cause different implementors of object operations to assume different semantics for the same object type, which can in turn lead to inconsistent and unexpected object behavior.

Design decision revisited: Substitutable subtypes

While other object-oriented systems permit subtypes to behave in radically different ways TOM requires subtypes to be *substitutable* for their supertypes, honoring all the semantic guarantees their supertypes provide. This requirement allows TOM clients to automatically use a larger range of types in predictable ways, since they can call operations on unknown subtypes of known types, and get results that conform to their expectations.

As a consequence of TOM's stricter subtyping requirements, some subtype relations that would be possible in other object-oriented systems are not allowed in TOM. For implementors, this may seem like a disadvantage of TOM, since many object-oriented systems use subtyping to enable code reuse. However, code reuse is less of a concern for TOM than for other systems, because TOM type definitions do not describe code, but behavior, and because TOM servers can still reuse the same implementation for multiple types through conversions with appropriate intersubstitutability.

We saw an example of the advantages provided by substitutable subtyping in Chapter 7. When the TOM Conversion Service encounters any subtype of the *package* type, it correctly unpacks it without having to know the details of the type in question, since all package subtypes must conform to the semantics of their supertype. In other object-oriented systems, it might be possible for subtypes to follow subtly different semantic rules that would prevent the package from being unpacked as intended.

TOM's insistence on substitutability also makes it possible to define intersubstitutable types for conversions, which guarantee that the input object and the output object of a conversion cannot be distinguished from each other through the interface of the intersubstitutable type. Both objects need to be subtypes of the intersubstitutable type, and therefore both conform to the behavioral semantics of that type; hence, one can meaningfully decide whether the objects can be distinguished using those semantics.

Conversions with intersubstitutable types help servers reuse code to work with multiple formats. On the type brokers at Carnegie Mellon, a common implementation strategy for operations is to implement an operation on type T for a particular format F of T. Other formats, and other subtypes of T, can be converted to F by requesting a conversion with T as the intersubstitutable type. This strategy allows the operation to be invoked on any format that can be so converted, while still obeying the same semantics that would apply if the operation were implemented for the original format (since the original object and the converted object behave the same, as far as the semantics of T are concerned). Hence, a single routine can be reused for a variety of related formats.

If someone takes a type hierarchy of another object system and defines corresponding TOM types for that hierarchy, the TOM hierarchy may be flatter than the old hierarchy, because certain subtype relations from the other object system may be disallowed in TOM. However, two types that were formerly subtype and supertype in the other system can still be related by creating a new supertype that defines features common to both original types, and then making the two original types subtypes of this new supertype. The new supertype can be useful in its own right.

Design decision revisited: Formal and informal semantics

While the semantics of MIME types are specified in natural language, usually informally, and the semantics of Java, CORBA, and OLE objects are typically specified in machine-understandable ways, TOM allows both kinds of specification. TOM can therefore include descriptions of types from both of these systems (except for operations that mutate objects). TOM's accommodation of informal semantics that cannot be checked or enforced automatically may seem to be a weakness in some situations. However, it allows TOM to handle types that are "semi-structured" partially specified, or not easily defined in formal terms. TOM itself also imposes certain global semantic constraints on types and formats, most notably the requirement that subtypes be substitutable for supertypes.

TOM brokers, like the other object-oriented systems mentioned above, do not automatically enforce most of the semantic constraints imposed by TOM or declared in type definitions. Indeed, current technology does not provide any general mechanism for a program to automatically check or enforce semantic declarations made in natural language.

Two questions therefore arise: First, should TOM require semantic constraints to be made in formal notation, so they could be checked and enforced automatically? Second, what good are TOM's semantic guarantees if the system does not (or cannot) automatically check or enforce them? Lacking automatic enforcement or (in many cases) required formal notation, does TOM's infrastructure give any better support to semantics than does the ability to place comments in CORBA or Java type declarations, and informal programmer conventions?

Certainly there are advantages to formally-expressed, automatically enforceable constraints, as can be seen in other systems. Eiffel, for example, allows preconditions and postconditions to be added to methods in the form of code that is invoked before and after a routine is called. An exception is thrown if that code detects a violation. In TOM, it is possible for type brokers to check method registrations to ensure that their signatures are consistent with the signatures of methods they claim to specialize. Current type broker implementations do not do this, but it might be appropriate for them to do so.

Even so, it would not be appropriate to rely solely on automatically enforceable semantic constraints in a system like TOM. There are three main reasons: First, type definers may find it too difficult to learn how to use all the formal systems required to state such constraints. Second, some constraints simply cannot be neatly expressed in formal terms, but can be expressed clearly in informal terms. Third, even if a constraint can be expressed formally, it may be impractical to enforce it automatically.

Practitioners of formal methods are already aware of the significant investment required to learn just one simple formal method. One might need to learn *several* formal methods, however, to optimally express and enforce constraints. For example, in Eiffel, a method precondition might be best expressed as a logical predicate, but might need to be enforced through complex Eiffel programming code. Since one of the design goals of TOM is to make it easy for users to define new types and services, type definers should not have to come up to speed in one or more new formal systems.

Furthermore, some constraints are not practically expressible in a formal system. One of the example types in Chapter 4, for instance, included a constraint that a particular encoding would be "parsable by some version of Microsoft Word." Such a constraint is easy to express and understand informally, but virtually impossible to express formally. Informal constraints can also be useful for "semi-structured" data, such as is often found in mail messages. Indeed, in the TOM Conversion Service, we defined multiple types for mail messages that had looser or tighter structure for the contents of mail, and were able to easily move between looser and stricter types through conversion and inherited operations.

Finally, even when it is possible to express and enforce a constraint formally, enforcement may still be impractical. In Eiffel, requiring code to be run before and after invoking a routine increases the overhead of a method call, sometimes substantially. (Furthermore, if semantics are expressed in predicate form, the time required to check the predicate can in theory be unbounded.) The overhead may make it not worthwhile to enforce a constraint on a particular operation.

We turn then to our second question: What good are semantic guarantees that cannot be automatically enforced? The primary purpose of semantic assertions in TOM is not to cause the behavior of types to be automatically enforced, but rather to make the behavior of the types *expressible* in straightforward ways, so that type definers can easily describe how they want their types to behave, and so that implementors can easily understand what rules they should follow. TOM trades control for clarity: programmers are not *forced* to behave properly, but they have clear instructions on how they *should* behave. They can also give clear instructions to others as well.[2]

TOM's global semantic requirements carry more weight than typical programmer conventions do. The substitutable subtype requirement, for instance, is explicitly stated in TOM's basic definition, which is available to all TOM programmers and type definers. Furthermore, several other parts of TOM (such as the intersubstitutability properties of conversions) take advantage of this requirement, raising the incentive to abide by it. Social pressures, as in other systems, provide further incentives for type definers to cooperate.

Similarly, semantic constraints for TOM types are more central to a type's definition than optional comments are for CORBA or Java types. Not only does TOM provide an explicit field in type, operation, and encoding definitions for semantic declarations, but it allows such declarations to be objects, and not just textual comments. Such objects can be manipulated and analyzed just like any other TOM object. Furthermore, TOM's basic definition explicitly states that implementors can implement an operation in any way that is compatible with the operation's signature, supertype specialization, and semantic declaration. Definers of a precise operation, then, have a strong incentive to include a clear semantic declaration to ensure correct implementations. In systems like Java, where interface declarations and implementations are often done by the same person, there is less incentive to include such comments, and subtype behavior can drift as a result.

Still, even when behavioral constraints are clearly specified, definers and implementors of types and services can still violate the constraints, either accidentally or purposely. If this occurs, types

[2] This philosophy is shared by other programming systems as well. For instance, the writers of the Perl manual note that "[i]n Perl culture... you're expected to stay out of someone's home because you weren't invited in, not because there are bars on the windows." [WCS96]

may behave in unexpected or erroneous ways.

The design of TOM, however, makes some of these errors unlikely, and limits their effect when errors do occur. For instance, in Chapter 6, I showed how subtype relations can only be declared by the owner of the subtype. The owner of the subtype, then, is the only person who can make an erroneous, unsubstitutable subtype relation, and that person is unlikely to do so compared to other people, because he usually knows enough about the type to avoid such errors. Furthermore, because subtype relations are assertions about the subtype, the person who declares an erroneous subtype relation only affects the correctness of his own type, and any subtypes of that type. Types maintained by owners known to make unreliable declarations of this sort, furthermore, are likely to get fewer subtypes than types maintained by persons or groups with more reliable reputations.

The effect of incorrect implementations of object services can be contained as well. Though TOM defines no mechanisms specifically for dealing with bad implementations, broker maintainers can remove agents from their type descriptions that implement an operation incorrectly. Reputation can also play a role here. Furthermore, it is simple enough for a particular server or domain known to be unreliable to be excluded from one's agent lists automatically. (In Chapter 5, I discussed some possible mechanisms that would give automatic preference to more reliable agents.)

Faulty type definitions, likewise, can be ignored in favor of types that are well-defined. TOM's namespace divisions can help identify well-defined types, since organizations known to be trustworthy could assign names from a namespace they control onto types that meet their standards. If there are many instances of badly-defined types, a conversion to a well-defined type can be defined, and then invoked before any further operations take place. This technique can also be used for obsolete types and formats.

8.1.3 Limits on expressiveness

I have shown, then, that TOM's expressiveness is superior to that of other networked object systems in significant ways. I do not claim, however, that TOM is adequate to describe *all* kinds of on-line information. Like the other systems described here, TOM ultimately represents and transmits information in the form of finite byte sequences. TOM's ability to handle information that is not naturally accessible via a byte sequence or a context-insensitive API may be limited. Examples of such data include data with values dependent on real-time constraints, or on a previous pattern of interactions. For example, RealAudio data is a stream designed to arrive at a client's machine in time for it to be output to a speaker as it arrives. The content of the data stream self-adjusts dynamically to accommodate the speed at which the client receives and processes data [Pro97]. Such dynamic, temporally-based data delivery is not supported by TOM, and in fact requires a special protocol to satisfy the demanding requirements of real-time audio streaming over the Internet. TOM can, however, still represent the non-real-time aspects of this information. For example, it can convert real-time data streams into static representations (such as converting RealAudio streams into .wav files), and then transmit those representations over the network. Similarly, data with a value that is dependent on an interactive protocol can be modeled somewhat inelegantly in TOM by passing back and forth extra objects that represent the state established during an interaction. Current TOM type broker implementations also support only finite byte sequences, and not continuous byte streams, but implementation changes could allow them to support returning infinite byte sequences without any changes in TOM, and a small augmentation of TOM's protocol would also allow brokers to accept potentially infinite data streams as input.

8.2 Composability

The second main requirement of TOM is that it be easily *composable* with a variety of information systems and data structures. In Chapter 1, I noted three important aspects of composability for widely distributed information systems: first, the ability to use a wide variety of data structures, including legacy data; second, the ability to interoperate with a wide variety of systems and application programs; and third, the ability to be adopted and integrated incrementally, with little investment required to reap initial benefits, and larger benefits reaped with further investment in the system. (Or, to express all this in another way, the system should tolerate a wide variety of practices in managing and using data.) Below, I summarize TOM's strengths in all three of these categories.

8.2.1 Data heterogeneity and legacy data

As I noted above, TOM type brokers currently define over 100 types, ranging from common MIME types to integers to virtual data structures like packages. TOM's types and applications support both previously-existing data structures (like Word documents and zip files) and newly-defined data structures (like ACME architectural description files).

An increasingly important kind of data is "legacy data", that is, data stored and formatted in ways that are no longer widely supported. Publicity about the "year 2000 problem", for instance, has drawn attention to the significant amounts of decades-old data formats and programs in many businesses and government agencies, the high reliance on these structures, and the high cost to keep them working as conditions change. The legacy data problem is also important in new application domains like digital libraries, which, like traditional libraries, may need to keep information usable for many centuries. In almost any field where new formats frequently supplant old ones, legacy data will play an important role. TOM's facilities for handling legacy data are superior to the facilities provided by systems like MIME, CORBA, or Java.

In particular, TOM provides two ways of dealing with legacy data and other heterogeneous data: conversion and encapsulation. I defined conversion in TOM in Chapter 4, and showed how inter-substitutability could be used to control information loss in conversions. I described how brokers support complex conversions in Chapter 5, and showed conversion services in action in Chapter 7. I also illustrated encapsulation through abstract interfaces in Chapter 4, and showed how strict substitutability allows subtypes to be encapsulated in supertype interfaces without any surprises. Chapter 7 showed an example of the use of supertype encapsulation through the "package" interface used by the TOM Conversion Service. Other networked object systems do not provide the rich conversion support TOM provides. They also limit the benefits one can obtain from using subtypes through encapsulation, because they do not require substitutable subtypes.

For heterogeneous data to be usable in the long term, though, clients need more than techniques like conversions and encapsulation. They also need to be able to find information about data types and formats, so that they can take advantage of those techniques. Information and services on legacy data, in particular, need to be preserved so that clients can continue to use them.

As we have seen, TOM supports long-term handling of this type information through type description objects. These objects record relevant information about types and their formats. Both the type description objects themselves and the brokers that maintain them are designed to enable long-term use of the described types.

Type description objects are themselves typed objects with published interfaces and formats. Therefore, even if new type description formats replace old type description formats, the old formats can be converted into the new type description formats, or encapsulated by the new interfaces, preserving the type information they contain. TOM thus “bootstraps” its meta-information on types and formats as standards change, ensuring the long-term usability of this information.

Moreover, TOM’s distributed broker architecture also encourages the long-term viability of large numbers of types and formats. Once a type or format is declared and registered with one type broker, other type brokers can copy this information and thus distribute the meta-information widely. Replication of type information makes it much harder to lose. Even if information about a given type disappears from some brokers, other brokers that had been given the information will still have it, and can propagate it as needed. Similarly, operations on types and formats can be replicated on multiple servers (and probably will be, if implementation code is made available for the operation).

Implementation code can also become obsolete over time, as languages and operating systems go in and out of fashion. However, it may be possible to package implementation code itself as an object of a particular language-specific format, like *awk*. It may be possible to later convert this code into a form that is more easily executed on newer computers, using the same techniques as described above. For example, conversions already exist to turn *awk* code into Perl.

In summary, TOM supports data heterogeneity more thoroughly than other systems do, through facilities like conversion, encapsulation, modeling of type information as objects, and replication of type information through a type broker network.

8.2.2 Computational heterogeneity

TOM supports computational heterogeneity in two important ways: First, TOM itself works with a wide variety of components. (By “components”, I mean interacting computational units, such as application programs, or clients, servers, or mediators.) Second, TOM acts as a mediator to enable components to work together more easily.

Components connect with TOM’s services either by talking directly to a TOM component using the simple TOP protocol, (described in Chapter 5), or by being invoked directly by a TOM server (via mechanisms like the script configuration file described in Chapter 5) or through a client-side or server-side gateway. In Chapter 7 we saw examples of gateways in action. On the client side, these included a simple Unix-based command-line interface and a few Web-based interfaces (some written by the author, some not). On the server side, we saw how the implementation of the `fetch` method on the URL type was used as a server-side gateway to retrieve information from various types of Web servers.

The TOM architecture also acts as a mediator to bring components together. In previous work [GAO95], Garlan, Allen, and I showed that a common obstacle to software reuse and interoperability is architectural mismatch, where heterogeneous components fail to work together. Common causes of this mismatch, we found, were conflicting assumptions about how components should operate and communicate, and what sort of data they should share. Shaw lists several common ad-hoc practices that programmers have developed to make components work together in the presence of such mismatches [Sha95]. The nine specific strategies, shown in Figure 8.3, cover a wide range, including publishing abstractions of components, adding converters, and maintaining parallel versions of components.

TOM and its applications support all of Shaw’s techniques that do not require expensive re-engineering of the components. Specifically:

- TOM types provide abstractions of components (Shaw’s technique 2) through abstract interfaces.

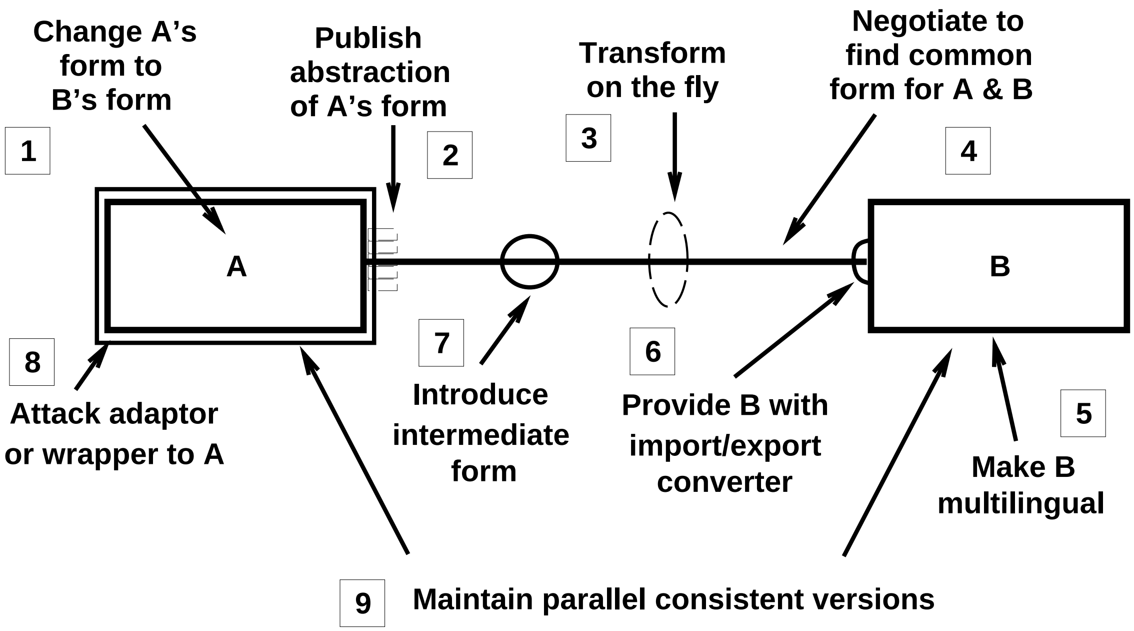

1
Change A's form to B's form
2
Publish abstraction of A's form
3
Transform on the fly
4
Negotiate to find common form for A & B
A
B
8
Attack adaptor or wrapper to A
7
Introduce intermediate form
6
Provide B with import/export converter
5
Make B multilingual
9
Maintain parallel consistent versions

Figure 8.3: Techniques for bridging architectural mismatch (redrawn from [Sha95]).

- TOM's conversion services change data from the form used by one component to the form used by another component "on the fly" (technique 3), via intermediate forms if necessary (technique 7). and also function as general-purpose import-export converters (technique 6).

- TOM components negotiate the form of data they wish to use, and can even switch to alternate protocols (technique 4). In particular, TOP's EXPECT directive allows a client to direct a server to return values in particular formats, and TOP's PROTO command allows clients and servers to propose and agree on protocol variants (or even protocols that have nothing to do with TOP). Appendix A has more details.

- TOM's facility to let clients invoke operations implemented on remote sites (and its proposed facility to supply code in various languages that a client can run locally) largely obviates the need for components to be multilingual (technique 5).

- Finally, server-side and client-side gateways built on TOM allow different interfaces and applications to be adapted or "wrapped" (technique 8), as we saw in the previous chapter.

Shaw's remaining techniques — rewriting the component itself (technique 1), and maintaining parallel versions of component data (technique 9) — both require substantial rewrites of component code, or meticulous programmer coordination. They are costly enough that they are generally used only as last resorts. TOM does not provide any special support for these techniques, but they are costly and error-prone, and should be avoided when possible. Since TOM does support all of Shaw's other techniques, though, TOM clearly supports a rich repertoire of techniques to handle component heterogeneity.

8.2.3 Computational heterogeneity: hard cases

Despite TOM's facilities for integrating components from a variety of sources, we still had difficulties making some services available through TOM. One particularly challenging problem for the TOM Conversion Service, for example, was to work with recent formats for Microsoft products like Word and Excel. The only programs we had that could work with these formats were the Microsoft programs themselves, which were designed to be used interactively by a human user, and could not easily be invoked and controlled automatically as TOM required. (While OLE in theory could control these programs, in practice we found that OLE only gave us a limited, and sometimes insufficient, degree of control.) Furthermore, since each new release of a Microsoft product often defined a new format, and slightly different OLE behavior, we kept having to spend significant energy to integrate the new program. Some in the TOM Conversion Service user community felt that the effort required to integrate Microsoft services into TOM was not worthwhile, especially when one could do some conversions (such as Word to Postscript) just through interactively using Word directly.

The Microsoft formats in some ways represented worst-case scenarios. A new format would start to get widely used shortly after its release, when the only programs that understood it were programs designed to be controlled directly be a human rather than by another program. Once we had a program that *could* control the Microsoft programs, a type broker or server could invoke it. Furthermore, the format specifications themselves were proprietary, and not published. If they had been published, we might have been able to use other programs, or written our own, to parse the formats. We did not have the resources to reverse-engineer the format or the program. These obstacles would be daunting not only to TOM, but to any other system that relied on automatically mediated operation invocation.

Even in these cases, though, TOM's infrastructure provides significant benefits and cost reductions over simply using Microsoft programs directly. TOM's ability to compose simple operations into complex operations broadens the range of operations available to a typical user. For instance, while a user may be able to convert a Word document to Postscript or RTF simply by using the built-in facilities of Microsoft Word, a conversion from Word to $\LaTeX$ poses a more difficult problem, since this is not directly supported in Word. A user without TOM would have to find a program that could convert to $\LaTeX$ from some format that Word could convert to, and then invoke that after running Word. This can be a nontrivial task. On the other hand, once a conversion from Word to RTF becomes available anywhere on TOM's server network, type brokers can then automatically compose it with other operations, such as an RTF-to-$\LaTeX$ conversion, to fulfill a client's request.

Furthermore, even if it is costly to provide a conversion from a proprietary format to a format that is easier to work with, when TOM is used the cost can be amortized over the entire user community. Without TOM, *everyone* who wants to convert from a new format of Word to RTF has to buy their own copy of Word (or another suitable program) to run on their desktop machine. With TOM, in contrast, once *one* person or group does the work of making a converter available, that service can then be used by everyone using TOM. The cost per user thus drops as the number of TOM agents and users scales up.

The analysis above can also be applied to many other "difficult" legacy systems, where clear format specifications are not readily available, and the programs available to operate on the formats are not easily made automatically controllable. Furthermore, sophisticated operating systems make it possible to build wrappers that automatically send input to programs, and receive the program's output, as if the programs were being controlled by a human user. Such wrappers can make even highly interactive programs invokable through TOM.

8.2.4 Incremental integration of TOM

Adopting any new system typically requires an investment. Someone adopting a new system incurs costs to acquire, integrate, learn, or use the system, and expects that the benefits will justify the cost. Systems that impose a high adoption cost before delivering benefits often fail to attract users, even if the systems promise significant benefits, because of the substantial costs or risks involved. On the other hand, systems that start benefiting users at a very low level of cost get adopted more easily. Furthermore, if the systems continue to provide additional benefits as the system is more fully adopted, they are likely to gain widespread use. If TOM required a heavy investment in time or effort before it delivered benefits, it would most likely never be adopted.

Fortunately, TOM's design and implementation encourage incremental integration. Users gain benefits from TOM's services, such as the TOM Conversion Service, at no more cost to them than visiting a Web site with a normal Web browser and filling out forms. Additional effort yields progressively greater rewards. Below, I describe some of the benefits, and their associated costs, for progressive levels of investment in TOM:

Use of public services. The TOM Conversion Service and TOM Frame Service convert files in various formats via a standard World Wide Web browser. Users simply specify what file they wish converted (using file upload, if appropriate), and fill out a form indicating the conversion they wish. Users do not have to know anything about TOM's typing or broker system, or install any special software, to take advantage of these services. A single, short documentation page appears to be sufficient to instruct users, since the first email we generally get from most users are thank-you messages or requests for particular conversions, rather than questions on how to use the service.

Use of private services. If users make a larger investment in TOM, and install type brokers and the TOM Conversion Service front end on their local machines, they get the extra benefits of faster conversions, since they cut out the overhead of sending files across the network, and no longer need to share broker usage with other remote users. They can also keep their documents private. To gain these benefits, users need to download, configure, and install a couple of programs. Specifically, they need to install the Conversion Service CGI program, and will also probably want to install a type broker program to run locally, although the Conversion Service is also configurable to use a remote broker. They will also need to install a standard HTTP (Web) server, if one is not already running locally. No additional programming is required.

Creating new services for existing types. If users wish to add functionality to TOM, they can add new services, such as conversions or methods, to existing types. To add a completely new operation, they must register its definition with a type broker. They must know how TOM operations are defined, but once they have a valid definition, they can use a Web-based user interface to register the operation. To add an operation implementation, a user must find (or write) a program that performs the operation, add it to a server, and register the server as a agent for that operation. As we saw in Chapter 5, adding a few lines to a configuration file can be enough to add a program to a server's repertoire. Users who want extra speed, or access to broker library routines, can compile the operation directly into the type broker or server code, instead of invoking it through a configuration file.

Creating new types. If users wish to use types or formats not previously defined in TOM, they can register new types with a type broker. They can then offer services on these types to clients and other users, or register encodings that specify representations of the types. They can relate the other types to existing types through subtyping and conversion. In order to create a useful type, users have to write a type definition for the type in question (which requires understanding TOM's type system), and register it with a broker. They also need to define enough operations or subtyping relations to make the type worth using. To create a new format, users need to register an encoding of a type that represents that type as a byte sequence. (This also requires some knowledge of the type system, but not as much as is required to create a new type.) Creating new subtype relations requires knowledge of the subtyping system and cooperation of the subtype's administrator, if any.

Creating new TOM-aware applications. Users can also write new TOM-aware programs for any desired information retrieval or analysis application. In order to do this, they have to be able to write a program that communicates using the TOP protocol. (As we saw in Chapter 7, programs that communicate using TOP can be as short as 200 lines of C.) TOM-aware applications can take full advantage of the information and services offered by TOM's type brokers and servers, instead of having to supply their own routines for handling every type and format they might encounter.

TOM thus supports several levels of integration, offers significant benefits even for minimal integration (such as the use of a Web-based conversion service), and offers increasing benefits for increased integration. Ease of integration makes TOM attractive to adopt.

TOM also avoids imposing unnecessary requirements for integration, and avoids requirements that other systems impose. TOM does not require that service providers code their routines in a particular language, like Java does. It does not require service providers to use a special format to encode their data, like OLE does. It does not require that information be managed by a system-specific broker, like CORBA and OLE do. Rather, TOM lets information providers continue to manage their data using their existing formats and servers. As long as there is some way to obtain this data over the network, the data can be integrated into TOM through the use of a server gateway (which does not have to be present on the same system as the data is stored, and does not have to be written or maintained by the data provider). Likewise, TOM lets information providers deliver data in whatever format they see fit; it can then be processed by clients either through conversion or through encapsulation, as described in the previous section.

In summary, because TOM gives wide latitude to different data formats, different computing interfaces, and different (and incremental) integration strategies, TOM proves to be a highly composable system.

8.3 Feasibility

A question that must be asked about any engineered system is: Does it work? There are three parts to this question: First, does it do what it was designed to do? Second, is its behavior reliable? Finally, is its performance acceptable?

The first part of this question can be answered affirmatively for TOM simply by noting the utility of the TOM-based conversion and framing services described in Chapter 7. These applications use all of the basic TOM functionality: querying type brokers, invoking conversions, fetching attributes, and calling methods. These applications are regularly invoked by users worldwide, handle thousands of operations per month, and work with a variety of documents and requested conversions.

Conversion	*Successes*	*Failures*	*Common failure mode*
Postscript to PDF	182	3	overloaded machine
Postscript to GIF(s)	102	0	
Word to web/HTML	65	8	Word encountered an error
GIFs to tar file of GIFs	69	3	
Mail message to structured msg	58	10	message not RFC-compliant
LaTeXto web/HTML	47	0	
DVI to PDF	42	0	
Word to Postscript	39	2	error retrieving document
Postscript to webbed images	25	1	ghostscript did not return
LaTeXto Word	25	0	
base64 to unencoded bytes	22	0	
Word to PDF	22	0	
HTML to Word	18	0	
Postscript to Word	15	0	
Word to LaTeX	0	15	no conversion implemented
PDF to Postscript	14	0	
LaTeXto PDF	14	0	
gzip to unencoded bytes	14	0	
Postscript to plain text	4	9	conversion did not return
Text to HTML	13	0	
Top 20 conversion requests	790	51	details above
All conversion requests	1035	192	no conversion implemented

Table 8.1: Most frequently invoked conversions, August 1997. Conversions to "web/HTML" return either a single HTML file, or a package consisting of one or more HTML files and accompanying inlined images.

TOM is also reliable, in the sense that it answers most well-formed requests successfully and correctly. The logs for the TOM Conversion Service, for instance, show that for the 20 most frequently requested conversions, success is reported over 93 percent of the time. (See table 8.1.) Many failures, particularly for the more obscure requests, simply reflect the absence of certain requested conversions. Some failures reflect the behavior of the user interface to TOM rather than TOM itself. (Early versions of the TOM Conversion Service, programmed by others, would occasionally mangle format information and thereby send a request for an inapplicable or nonexistent conversion. This request would then fail, even though the correct conversion would have succeeded.)

To get a complete understanding of TOM's reliability, one has to consider not only how often the *system* believes conversion has been successful, but also how often *users* believe conversion to have been successful. Such results are necessarily qualitative. As the TOM Conversion Service was being developed, we did get a number of reports from internal and external users that conversions did not produce the desired result. The vast majority of these bug reports, though, related to problems with the implementations of operations invoked by TOM, such as specific conversions, or format identification operations. Early bug reports also cited problems with the user interface to the TOM Conversion Service that have since been fixed. Few problems were reported with the TOM type brokers themselves.

Even if TOM was not directly responsible for problems with operation implementations, such problems could not be ignored. Earlier in this chapter I discussed the problems encountered with proprietary formats like Microsoft's, and how TOM's infrastructure still provides benefits and lowers costs in the face of these problems, particularly when it scales up. We also sometimes found we had integrated buggy third-party converters that did not always convert correctly. We fixed or replaced these converters as best we could, and we expect TOM to handle these problems more smoothly as it scales up. If multiple implementations of operations are available, broker maintainers can choose the most reliable among them to include in their agent lists.

As time progressed, and the interface and conversions improved, the bug reports we have received have decreased, and lately have been outnumbered by requests from organizations for their own copies of the software, so that they can convert their own documents internally. This feedback suggests that many users now find the system reliable and desirable.

TOM is also designed to recover from problems with particular servers, and from operation failures. As we saw in Chapter 5, type brokers replan when a server or operation fails. As more servers and services are added to the network, more alternate plans become available, further increasing TOM's reliability.

TOM's low overhead and user satisfaction also indicate acceptable performance. TOM imposes very little time overhead. Timings made of the conversions requested in table 8.1 show that type brokers only spend a few milliseconds of wall clock time planning conversion strategies on a Sparcstation-5, small enough to be ignorable. Most of the time spent in conversion is the time required to transmit data between client, brokers, and servers, and the time used by the conversion itself. The time required to compute conversions is outside of TOM's control, and can range from fractions of a second to over an hour for certain conversions from Powerpoint. The time required to transmit data depends on the size of data being transmitted, and on the available network bandwidth between clients, brokers, and servers. (At Carnegie Mellon, the type brokers and servers are all on the same local network, which allows for high-speed transmission between brokers and servers.) Some of the most frequent use of the TOM Conversion Service at Carnegie Mellon comes from European sites, indicating that users there find that the services provided by TOM are worth even the delay required for trans-Atlantic data transmission between clients and the TOM type brokers.

Space overhead imposed by TOM, on the other hand, is sometimes significant, since type brokers store intermediate results of multi-step operations, in case they need to use them in a modified execution plan. These intermediate results can sometimes be large. However, we have not to date found the space overhead of type brokers a problem, but if space were tight, intermediate results could be flushed. This would trade space savings for extra time overhead if a formerly cached value needed to be recomputed. Another space-time tradeoff, used in some conversion implementations, is to compress intermediate results. TOM operation implementations also use a certain amount of space outside of TOM's control. On a heavily used server, these space requirements might limit the number of TOM operations that could be in progress at any given time.

8.4 Scalability

The architecture of TOM is designed to scale gracefully as the number of types, users, and services increase. Experience with the types and services we have added to the two type brokers at Carnegie Mellon indicates that TOM scales up well in practice. TOM is designed ultimately to be deployed widely, with many independently-managed brokers on different sites on the Internet. It is difficult to predict with certainty whether new problems will arise at that scale, but early indications are promising.

8.4.1 Scaling up usage and services

Since, at this writing, TOM type brokers and servers are available only at Carnegie Mellon, we have not attempted to publicize them outside of the university, except at meetings of research groups, for fear of overloading the servers. Still, word does spread outside the university, and the TOM Conversion Service as of June 1997 was being accessed about 75 times per day, with more accesses from outside the university than from inside. This level of usage, and subsequent growth, has not caused significant problems.

Because of the distributed architecture of the type brokers and services, a much larger number of users should be easily accommodated if other sites install their own type brokers and TOM servers. Just as the wide distribution of HTTP servers made it possible for a worldwide, large-scale distributed hypertext system to arise in the form of the World Wide Web, so too should wide distribution of type brokers and services make a worldwide, large-scale network of structured information services possible.

Indeed, in some respects, TOM scales better than the Web does. For example, the Web's form of reference, the URL, requires that a particular server be contacted for resolution. This can cause bottlenecks for popular services delivered via the Web (as I noted in Chapter 7 when discussing the TOM Frame Service). In contrast, TOM's provides server-independent ways to refer to type services, and lets type brokers transparently contact any of multiple servers capable of performing a service. In addition, other kinds of server-independent references can be easily integrated into TOM by defining appropriate subtypes of TOM's generic *reference* type. TOM's data and metadata references, supported by TOM's mediator network, allow more graceful scaling of services than does the World Wide Web.

8.4.2 Scaling up the type graph

In Chapter 6, I discussed how TOM is designed to permit the graph of known types to grow in an unbounded, distributed fashion. I also showed how a type can be extended in ways that do not invalidate previously-defined semantics for the type. Here, I simply note that our experiences in building the type graph are consistent with the expectations raised by that chapter. An undergraduate staff member and I have registered over 100 types between us on two type brokers, one maintained by me, the other maintained by the staff member. Each broker has a separate subspace of the `net:` namespace for defining new types, so independently-defined types do not interfere with each other. TOP allows us to check what new types and services have been defined on the other person's broker, and update local type information as needed. Implementations of the operations defined on the types are distributed over the two brokers and an auxiliary server on a Windows NT machine.

We have not yet needed to automate the updating of types between brokers, but could do so if necessary, so that when a new type or service is registered on one broker, the registration automatically propagates to other brokers. In practice, as the staff member was learning to use TOM, we found it useful to review manually what types and services had been been defined, so that suggestions could be made regarding the best ways to define and implement a type. However, this manual review was not strictly necessary, since flawed type definitions can always be ignored, and more carefully designed type definitions adopted in their place.

Design decision revisited: Naming schemes

For the type graph to scale up, new type and operation definitions must not interfere with old ones. In Chapter 6, I discussed how TOM's naming schemes attempted to avoid such interference, while still attempting to keep names easily assigned, copied, and remembered. I argued for a system in which unique (but long) names could be assigned easily for any new type at any type broker, and shorter aliases could be assigned for more tightly controlled namespaces. At the same time, I argued that operations and encodings needed shorter, more mnemonic names than those used for types. Noting also that name conflicts within the scope of a single type would be rare and could be quickly detected and repaired, I concluded that it was acceptable for the definers of new operations and encodings on a type to choose their own names.

How does this naming scheme compare to that used by other systems? Given that other distributed systems give more freedom in selecting names than TOM's type naming mechanism does, is TOM's type naming system unnecessarily strict? Or, if TOM's type naming system is *not* unnecessarily strict, is TOM's naming system for operations and encodings too *loose?*

With regard to type naming, I note that systems like Perl, Java, and Emacs Lisp, all of which encourage sharing of named entities across the Internet, rely only on convention to avert naming conflict. However, all of them have found it useful to partition their global namespaces into *packages*, each of which defines its own naming subspace. Program authors are encouraged to place all globally scoped identifiers within a single package. This avoids conflict with anyone else's identifiers – if no one else happens to create another package with the same name. The main difference, then, between namespace management in these systems and TOM's type namespace management, is that the other systems do not control the assignment of package names (or require programmers to use packages at all). TOM, in contrast, requires all types to have names in controlled namespaces.

Although the "package" convention is sufficient in most cases to avoid name conflicts, it does not always work, and recovery can be difficult when conflict does occur. Perl, Java, and Emacs Lisp have largely avoided most name conflicts so far for two reasons: first, the number of widely distributed packages is not huge; second, there are central repositories for distributed code in these systems, making it easy to check to see whether package names are taken. These safeguards become weaker as the system scales up, and the number of identifiers becomes to unwieldy to be handled by a central repository. Since I intend TOM to be able to attain this scale, it needs to have stricter control of its type names.

Furthermore, even in the systems described above, conflict has occurred, due to the first-come, first-served nature of the package namespace. For instance, the first widely used CGI module for Perl took the package name "CGI". It had significant design limitations, but later, improved packages would have to use different, less mnemonic names. Some Perl gurus therefore found it necessary to post special notices directing users away from the original "CGI" package to an improved module [Chr96]. Eventually an improved module was able to reclaim the "CGI" name, but only because Perl had changed its namespace organizing conventions, allowing old names to be reused in the new namespace organization. Of course, TOM types with easily-remembered names can also become obsolete over time, but since certifying agencies can control popular namespaces, they can reserve popular type names for high-quality type definitions.

Even with only two local type brokers, TOM's stricter type naming system has already averted a similar name conflict. At one point, the maintainers of the two brokers were both independently, without the other's knowledge, defining types for the *tar* package that were subtly incompatible with each other. Because we were each working on different type brokers, our type names used different `net:` subspaces, allowing the two types to be distinguished from each other. After consulting with each other, we agreed on a particular type to use. This type, which had been tested and approved by our own consensus, could later adopt an alias in a more tightly controlled namespace that used shorter, easily-remembered names.

If there are advantages to tight control for type names, should operations and encoding names have similarly tight controls? In Chapter 6, I argued that such control was not necessary, because names of operations and encodings appeared in a much smaller, per-type scope, rather than the global scope used for type names; and because it was useful to have short, easily added operation and encoding names. In our experience thus far, we have not encountered any name conflicts for operations or encodings added to an existing type. This may be in part because of the relatively small size of our current type space and the small number of people adding new operations; however, we see from the *tar* example that we already have enough scale for potential type name conflicts, but have avoided operation and encoding name conflicts. As TOM scales up, the contact information for the maintainer that is placed in a type description object can also help avert naming conflicts within a type. Programmers can use this information to contact the originator of a type before adding new operations or encodings, to make sure the maintainer knows of no potential naming (or semantic) conflicts.

Design decision revisited: The evolution of type definitions

Chapter 6 discussed what is allowed to change in type descriptions after they are declared "published", and what has to remain the same. (Basically, modifications to a type description are not allowed to change the semantic behavior guaranteed by that type, either by loosening or tightening semantic guarantees.) TOM also allows type semantics to be changed during an "experimental" phase, when the type is still being developed locally. At Carnegie Mellon, we have found this experimental phase a useful one, since we often need to try out and rethink a type definition as we develop it, changing it as it is being "debugged." After some experience using the type and its operations, the type definition stabilizes.

The question arises: why not just keep all types "experimental" indefinitely? Indeed, most types on our local brokers are still registered as "experimental" rather than "published", even though most of the type definitions have stabilized. Types are more useful to other users, however, if they are registered as "published", because then other users can be assured that the semantics of the types would not change. Such types, and their operations, would therefore become more popular. There may be more of an incentive to upgrade types to "published" status once non-local people start working with the type definitions directly, and add their own definitions and operations. At the moment, only two people are registering type definitions, and we communicate frequently, so we have not had much incentive to freeze type definitions.

Still, published types and formats may become obsolete over time as Internet conventions and data formats evolve. Changes in format that are purely syntactic can be handled by defining new encodings for the new formats, and perhaps defining conversions between the old and new formats. Semantic evolution can be handled by defining new types, which then can be related to the old types through conversions, and possibly through a subtype relation with the old type (possibly using a new virtual supertype of both types, if the two types do not have a valid direct subtype relation). Applications can then use either the old types or the new types through appropriate conversions, or through the supertype's interface. Our experience to date with this sort of evolution has been limited, but the development of *packages* in the TOM Conversion Service suggests that it is feasible in practice, since we were able to define new supertypes and new variations of types in a backward-compatible way, and clients could use a variety of types, both old ones and new revisions of old types, through the same generic "package" interface.

8.4.3 Imperfect scaling: format identification

As I noted in Chapter 7, when we developed the TOM Conversion Service we found we needed routines to infer formats from byte sequences, in order to make the best use of data from outside TOM. Unfortunately, these routines do not scale well as more formats need to be checked. In theory, if all file format tests took the same amount of time, and each test placed the file into one of two equal-sized categories of formats, the amount of time to classify a file as one of n formats would increase by order $log(n)$. In practice, though, our current algorithms do not do that well, and for the most part have to test formats one at a time, since there are few patterns of identification common to a large number of formats. This means that the time required to identify a format grows more or less linearly with the number of formats. Furthermore, some of the format checks may require a long time to execute, especially if they have to scan a large input file.

The running time required for the algorithm can be reduced by doing quick checks, and checks for popular formats, first. However, this strategy may require frequent tuning of the format identification algorithm as new formats are introduced. Our current setup, while not finely tuned, does let broker maintainers introduce new format checks by placing regular expressions and scripts in a configuration file. They do not have to modify any program code or fine-tune the identification algorithm. The most practical approach, given that speed is often more important than complete coverage in format identification, may be a "90 percent" solution, in which most brokers identify only the most popular formats, but in a short period of time. We could also define and implement a more rigorous (and slower) format identification method on the *byte sequence* type that clients could optionally call.

8.5 TOM amidst the Internet: 1998 and beyond

In the more than three years since the basic design of TOM was first proposed, the Internet has changed dramatically. The World Wide Web has grown by several orders of magnitude, and now encompasses a much wider, and more sophisticated, set of systems and formats than it originally did. Some of the object-oriented systems now in use on the Internet, such as Java, had not been made public when this thesis work was starting; others, like CORBA and OLE, were still in early development. Development of the Internet in the next three years is likely to be as brisk as in the past three years.

The rapid development of the Internet raises an important question: Does TOM have something new for the Internet community that these other systems do not? A related question is: Can TOM take advantage of the new developments on the Internet, and not be pushed aside by the other object-oriented technologies and developments of the Web?

The answer to both of these questions is a definite yes. The rapid growth of the Internet and the diversity of data formats on the Internet make even more apparent the need for a system like TOM that can make these data formats interoperate through a brokered, distributed graph of types, formats, and services. We see the demand for such a system in the increasing usage and demand for the TOM Conversion Service, for example. The standard approach of only handling a few common formats is increasingly untenable, for two major reasons: First, as more formats are introduced by a growing number of groups, and information in old formats persists, there is ever more data not represented by the most common formats, particularly in specialized application domains. Second, the sheer volume of data now available on the Internet is increasingly difficult for humans alone to sort out; programs are needed to analyze, filter, transform, and abstract the data. Effective analysis often requires being able to work with the semantic structure of the data, and the "lowest common denominator" formats typically do not provide adequate semantic structure.

As I have shown, TOM allows clients to use an ever-growing variety of data types and formats, without requiring maintainers of the data to adopt a pre-selected format to represent their data, or a predetermined policy or protocol to manage and serve the data. TOM handles both new and old formats of data much more flexibly than systems like CORBA, Java, or OLE. It is not a replacement for any of these systems, since all of these systems handle general-purpose object manipulation, including mutation. The niche that TOM inhabits in the universe of Internet systems is similar to that of MIME, since (as we have seen above) TOM can describe all the formats that MIME can and more, and in ways that provide additional detail and easy access to servers that can operate on the formats. As with MIME, TOM types describe values, rather than mutable storage locations. However, TOM's potential role is considerably larger than MIME's, due to its wider expressive scope and its support for brokered services.

It is also possible for TOM to work alongside other object-oriented systems. Though I have not yet implemented programs that do so, it is a natural extension of TOM's current capabilities. In many cases the objects and types of those other systems can be treated as TOM types and formats. If Java or OLE objects are transmitted over a network, for instance, their encoding can be described as a TOM format, and the parts of their interface that do not deal with mutation can also be defined as the interface of a TOM type. Some operations that mutate an object can be redefined to return a new object, identical to the old object except for a change called for the mutation. TOM-aware clients may be able to operate on objects from other systems by using server-side gateways from TOM to one of those systems. Portable code systems like Java can also be used to ship implementations of TOM operations to be executed on a client that handles both TOM and Java (as I described briefly in Chapter 5).

On the client side, clients that understand CORBA, Java, or OLE may be able to look at TOM types through objects defined in one of these systems. This can be done, for instance, by modeling a particular TOM type as a CORBA (or OLE, or Java) object. It may be necessary to encode TOM objects of that type in a format compatible with the other object system, but one can always declare an encoding of the TOM type that fits that format, and then define a conversion from more usual TOM formats to a format compatible with OLE or Java. Alternatively, in systems like CORBA the object might stay on remote servers in any convenient format, and be viewed through a client-side gateway to a type broker that invokes appropriate operations on the object. Another approach would use a meta-object protocol, where the connection to a type broker, or the TOM interface of a type, would be itself modeled as a CORBA or Java object, and TOM objects would be created and manipulated through this meta-interface.

Why would one want to combine systems like this? One reason is that type and format incompatibilities can occur in any networked object system. After a system is in place long enough, and data types and formats continue to evolve, existing programs may eventually become unable to handle newer data formats, even though these newer data formats contain all the information present in the older data formats for which the programs were written. If the programs could first call a set of routines to ensure that their inputs conformed to known types and formats, they would be able to deal with new types and formats created after the programs were written. TOM, with its conversion facilities and type relations, can serve as the back-end for such routines, allowing the programs to handle an ever-growing set of data formats.

TOM thus makes it much easier for people and programs to handle the ever-growing range of data types and formats on the Internet, without requiring them to keep up with all the new data types or maintain their own routines to handle all the old data types. In addition, whether or not TOM itself becomes widely used as a system, my research provides a number of useful contributions towards further research and development of distributed information systems. I will discuss some of the major contributions, and some possible areas of future work, in the next, and final, chapter.

Chapter 9

Conclusion

I have now demonstrated the soundness of my thesis claims, and showed that, using an appropriate data model and a network of type brokers, a system can be built that allows clients and servers to use data in a wide variety of formats. In this concluding chapter, I briefly summarize the main contributions of this thesis research. I then suggest ways for future work to build on what I have demonstrated here.

9.1 Contributions

The goal of my thesis work is to make it easier for clients and servers across the Internet to take advantage of a wide range of complex data types and formats. The thesis work has focused primarily on "read-only" applications such as data discovery, display, and analysis, rather than on "read/write" applications such as those involving data mutations.

My contributions towards the goal stated above are of two kinds:

Useful programs, type information, and services: The artifacts and type and format information developed for this thesis in themselves are important contributions towards more flexible use of diverse data formats. The TOM Conversion Service is used worldwide to convert and unpack objects that clients cannot interpret themselves, and the TOM Frame Service provides "backward compatibility" of framed Web services with World Wide Web browsers that do not adequately handle frames. The type brokers implemented for this thesis mediate dozens of services on behalf of various kinds of clients, and handle registrations for new types, formats, and services in a clean, scalable manner. The graph of type information that these brokers maintain covers hundreds of commonly-used (and less commonly-used) formats, and includes useful hierarchies like the "package" hierarchy.

Lessons on how to make complex, heterogeneous data widely usable. Aside from providing the artifacts above, I have analyzed the design of TOM and discussed the strengths and weaknesses of the system. My design and analysis of type brokers gives an example of how mediators can be used in a loosely-managed distributed system to take advantage of knowledge and services from across the Internet. I have also defined and demonstrated the utility of a data model that combines many of the advantages of object-oriented "black-box" data types and low-level concrete representations. I have shown how that data model can be applied to a distributed, heterogeneous environment. I have shown how a "read-only" model of objects, though limited in its application, can take advantage of data from a wide variety of servers, including legacy data; and how a system using that model can raise the common denominator of data types that are widely communicated and understood. I have introduced a measure of information loss in conversions, known as "intersubstitutability", and shown how it can be used to safely and automatically convert between formats while preserving vital information, without requiring the use of a "standard" intermediate data representation.

9.2 Future work

While I have made important contributions in my thesis work, considerably more can be done, both in extending TOM's design and implementation, and in further researching some of the questions and issues that TOM has raised.

9.2.1 Extensions of TOM's design and implementation

Adding facilities for client-run code. Currently when a client requests an operation from a type broker, the broker forwards the request and its accompanying data to a server that carries out the operation. Sometimes it is preferable for the broker to return code that the client can execute locally to carry out the operation. This code could be run in a "safe" environment that isolated it from sensitive resources, as is done with Java scripts, or it could be certified to come from a trusted source. This extension would not require major changes in TOM's design. The code would be encapsulated as an object, and be registered as another kind of "agent" that could carry out a particular operation. (We would also have to define a standard convention for the code to get its input and provide its output, as we have done for scripts invoked from a TOM server's script configuration file.)

Using TOM in conjunction with other systems. In the previous chapters, I showed how applications can use TOM and the Web together, when those two systems are linked with appropriate client-side and server-side gateways. I also noted that similar gateways could be built for systems like CORBA, Java, and OLE. Such gateways would enable TOM and general-purpose distributed object systems to be used together to take advantage of the resources implemented with both systems, and would allow the general-purpose object systems to handle a wider range of heterogeneous data.

TOM can also significantly enhance many domain-specific systems. One domain for which TOM may be particularly suited is digital libraries, since such libraries involve the integration of data and cataloging metadata from many different sites, and in diverse and complex formats. Because libraries attempt to archive their materials for centuries, they need to maintain information and conversions for legacy data that they store. Type brokers are well-equipped to handle this task.

Extensions for security and electronic commerce. For simplicity, this thesis work did not consider issues related to security (other than reliability) or electronic commerce, but both of these issues are important in a widely used system. Security is important because data may be confidential, or may need to be certified by trusted sources. The maintainers of some services may want compensation, because the services consume expensive resources or are particularly valuable to clients. Others have done much work in these areas, and their work in many cases can be incorporated into extensions to TOP. For example, one can improve data security by encrypting data and protocol streams, and by specifying that brokers should only consult certain trusted servers for certain computations. Similarly, pricing and bidding protocols can be added to the TOP request protocol, and the cost of services can be taken into account when brokers plan conversion strategies or choose which agents will be delegated to perform an operation.

Adding extensions to the data model. Several extensions of TOM's data model are possible, some of them suggested by users. Three of the more interesting extensions involve exceptions, conversions, and parameterized types. Having exceptions return an object, instead of just a fixed error code, could give more information to help a client recover from an error. Allowing conversions to include optional input parameters can give clients more control over the result of a conversion. First-class parameterized types, supported directly by TOM's data model (rather than indirectly through the use of the ad-hoc templates implemented in our local brokers), might allow a graph of useful types to be built more easily. The benefits of each of these features would need to be weighed against the extra complexity they introduce into the data model, and the potential problems they may introduce. For example, parameterized conversions might pose problems for programs needing to automatically convert objects that could not supply needed parameters, and it might be more difficult to specify the information retained in a parameterized conversion.

9.2.2 Further research involving TOM

Empirically studying multi-site scalability. Because we have deployed TOM broker software only at Carnegie Mellon, my analysis of TOM's scalability in practice was limited to observation of the brokers at Carnegie Mellon. When type brokers are set up at other sites, there will be many more people setting up TOM services and defining new TOM types. This larger-scale usage is likely to test TOM in unanticipated ways. Studying TOM's behavior, and its problems, when many independent groups are maintaining brokers and registering new types and services, should teach useful lessons about how to design distributed information systems for large-scale use. Particularly interesting phenomena to watch include the performance of type brokers and their planning algorithms with a much larger-scale type graph than exists today, the ability of TOM to gracefully handle incorrect and erroneous information and services, and the degree to which changes to TOM's design can be gracefully made once the system is widely deployed.

Studying the utility of new specification mechanisms. In this thesis work, I allowed open-ended, informal specifications of the semantics of data types, encodings, and operations, and supported relatively few formally-defined assertions directly. (Some formally-defined assertions that TOM does support include typed signatures for operations, the subtype relation, format "signatures" for conversion, and an intersubstitutability specification for conversions.) Other formal specifications, encapsulated as objects, can be placed in the "semantics" attribute of type-related definitions, but TOM provides no built-in conventions for these specifications. It may be useful to investigate what kinds of formal specifications are best suited for types defined in a system like TOM, and what benefits (e.g., automatic verification) they can provide. In addition, it may be useful to compare the "intersubstitutability" specifications that TOM supports with other ways of measuring what information is preserved or lost in a conversion.

Studying how TOM can be applied to read/write applications. For simplicity, and for greater flexibility in working with diverse servers, objects in TOM are read-only values, and TOM does not provide a mechanism to mutate objects or change data stores. I have noted that

TOM can be used in conjunction with read/write systems; for example, I have suggested using TOM together with systems like CORBA. It also may be possible to extend TOM's own data model to support writes as well as reads. One way to do so in a limited fashion would be to generalize the "registration" model currently used for new type information, and also allow arbitrary objects to be "registered" with brokers or servers. This facility would allow certain kinds of updates to occur, but not require fully general read/write semantics for objects. It would, however, require applications to keep closer track of the particular servers with which they were interacting, rather than simply letting a broker hide these details. A more radical change would be to extend TOM to fully support mutable objects, in a separate type hierarchy from the read-only types that TOM now supports. Such objects would have to be handled differently from read-only objects, and it is unclear whether such a hybrid system would have any advantages over simply using TOM and a general-purpose object-oriented system together.

9.2.3 Some directions for further research

Advanced mediator research. I have shown in this thesis work how mediators, in the form of type brokers, can work with a wide range of clients and servers, and adapt diverse data formats to the needs of clients while accommodating the existing practices of data publishers. There are many other roles that mediators can usefully play in networks like the Internet. (We noted earlier that web-crawling search engines are useful mediators for resource discovery, for example.) It will be interesting to explore what useful tasks mediators can play in large-scale information systems. Furthermore, as the population and diversity of mediators on the Internet grows, it may be important to determine ways in which different kinds of mediators can cooperate usefully, both in sharing information and in conserving the use of network and computing resources.

Solving heterogeneity problems in other domains. Heterogeneity in widely distributed systems is not limited to data formats. The number of distinct application protocols on the Internet is also large and growing, as is the number of distinct APIs for networked applications. Can modeling and mediation techniques like the ones used in TOM help clients work with a large set of protocols and APIs? TOM's gateways address this problem to some extent, allowing some of these interfaces to be addressed through TOP. But this solution, like the standard intermediate data representations used by OEM and related systems, mandates a "hub" – in this case, the TOP protocol – through which all transactions must pass. Is there a way to model and translate these protocols and interfaces without requiring an intermediate Esperanto through which the translations must pass, just as TOM can convert between formats without requiring translation through a specific intermediate form? If there is, it may further help heterogeneous programs work together.

New applications for widely distributed data systems. The effects of an invention, once put into use, often go well beyond the particular domain for which the invention was designed, and have consequences in other areas that are often hard to predict. The printing press, for example, arguably had a major impact on religion; the telephone changed business and social life; the Interstate highway system changed urban and rural economies. Many commentators have predicted that the Internet will eventually have a similarly broad impact on society. The ability for clients to use a variety of expressive, heterogeneous data formats may increase the utility, and therefore the effect, of the Internet. This ability may also inspire new applications that have not yet been imagined. What sorts of applications will these be? What sorts of data structures will they involve? For instance, will it become easier to design search engines that understand the semantics of documents as well as the words or patterns that appear in them? Will this help users find needed information in an ever-growing collection of data? What would the social impact of these engines be? On the positive side, will they change for the better the way people learn and get informed? On the negative side, will they help further erode personal privacy, as disparate personal databases are more easily linked together? It is difficult to say at this point what the long-term effects of systems like TOM will be, but it is possible now to investigate what sorts of applications it enables, and the impact of those applications.

The population of the Internet – both the number of people that use it and the volume of data it contains – continues to grow rapidly. I expect the growth of the Internet, computing power, data storage capacity, and network bandwidth to continue at a rapid pace for at least 10 more years, and possibly substantially longer. New and significant distributed data systems and applications should continue to be introduced for at least that long. I hope that while this expansion period continues, my thesis work will serve as a basis on which people can build to facilitate data interoperability, and also serve as a case study for designing widely distributed information systems to accommodate diverse needs.

Appendix A

Specification of TOP: The Typed Object Protocol

A.1 Status of this specification

This appendix specifies version 0.2 of the Typed Object Protocol, used by information agents in the computation model of this thesis. This version is currently implemented by the server at tom.cs.cmu.edu as of April 1997.

The definition of TOP prior to version 1.0 is subject to change without notice. Versions of TOP prior to 1.0 need not be compatible with either previous or subsequent versions.

A.2 Summary of the protocol

TOP is used by agents in information systems to retrieve, convert, and perform various operations on typed objects. In TOP, a *client* contacts a *server* using a stream connection (such as TCP), and issues a series of requests. After receiving a request, the server sends a reply. A given information agent may act as both a client and a server, and may talk to multiple clients and servers simultaneously.

TOP servers, unlike HTTP servers, do not normally terminate a session after sending a reply, unless the client requests to end a session. However, TOP is not designed to be used directly by end users, nor are clients expected to keep connections open indefinitely. Therefore, a server may terminate a TOP session if it does not receive a request within 60 seconds of the last reply. The server may also terminate a connection if a client fails to complete a request within a certain time period. This time period should not be less than 60 seconds, and should probably be significantly longer. Of course, agents should assume that network and machine failures may cut off a connection at any time.

Requests consist of one or more lines, depending on the kind of request given. The first line of any request starts with the name of the request followed by a space or a line termination. The server is always expected to give some sort of response to the first line of the request; this may either be a final reply or a reply indicating that the client should continue. All lines are terminated with the carriage return character (CR) followed by the line-feed character (LF). Delimited VALUE blocks (q.v.) may, however, contain arbitrary data, including carriage returns, line-feeds, and nulls, between the delimiters. Outside of delimited VALUE blocks, all characters transmitted should use the ASCII character set.

In the descriptions below, a "word" consists of a contiguous sequence of non-whitespace printable ASCII characters. Words are delimited by whitespace, or by the beginning or end of a line. Material in <angle brackets> indicates required parameters. Material in [square brackets] indicates optional parameters. If no space exists between the brackets, the parameter is required to be a single word.

Responses to requests begin with a 3-digit code, followed by a space. In general, codes starting with 2 indicate an ordinary response. Codes starting with 3 indicate that the client is expected to provide more information to complete the request. Codes starting with 4 or 5 indicate an error or exception; 4 implies that the client is responsible for the exception, 5 that the server is responsible. (The actual assignment of responsibility is sometimes a judgment call.)

Unless otherwise noted, responses are a single line.

Version 0.2 of TOP defines the following requests:

PROTO	Used to negotiate protocol
NOOP	Used for synchronization
QUIT	Used to terminate a session
OPER	Used to call a remote operation
ATTR	Used to fetch an object attribute
CNVT	Used to convert an object to an alternate format
TYPQ	Used to get information about a type
TYPL	Used to list types that have been registered or changed recently
REGI	Used to register new type information
AUTH	Used to request an authorization

A.3 Request descriptions

A.3.1 PROTO: Protocol negotiation

This PROTO request is used to ensure that the client and the server agree on a protocol to use. It is expected that all versions of TOP will support this command.

The client sends a request specifying the protocol it wishes to use. The server then decides what protocol it will use to talk to the client and sends back a reply specifying that protocol. This may be the protocol that the client requested, or it may be another protocol.

Request format:

```
PROTO <protocol specification>
```

Reply format:

```
201 <protocol specification>
```

If the PROTO request results in a change of protocol, that change takes place for client messages following the PROTO line, and for server messages following the reply line.

The protocol specification consists of one or more words on a line. The first word indicates a particular protocol version. That version may define an interpretation of any remaining words. TOP version 0.2 uses the following protocol specification, with no additional words defined:

```
TOP/0.2
```

PROTO may be used to change protocol family altogether if the client and server are willing. (For instance, a particular TOP server might also function as an HTTP server.) A TOP server, however, should not switch to a non-TOP protocol unless the client specifically requests it.

A.3.2 NOOP: Synchronization

The NOOP command may be used to ensure that the client and server remain in sync with each other. It may also be used to keep a session open, in cases where the server might otherwise time out between commands.

Request format:

```
NOOP [optional extra text]
```

Reply format:

```
200 [optional extra text]
```

If the client supplies additional text after the NOOP, the server must echo back exactly that text after the reply code. Otherwise, the server may supply whatever text it likes, or none at all.

A.3.3 QUIT: Terminates a session

The QUIT command is used to terminate a session. Clients should, if possible, issue this command before dropping a connection.

Request format:

```
QUIT
```

Reply format:

```
205 [optional extra text]
```

After the server sends the reply line, it drops the connection with the client.

A.3.4 OPER: Carries out a remote object operation

The OPER command is a request for the result of a method (object operation) on an object. This is a multi-line command, and the format is more complex than the earlier commands.

The format of the request begins with the initial line:

```
OPER <opname> [optional-typename]
```

where opname is the name of the operation desired, and typename is the name of the type that defines the operation. If the typename is omitted on this line, it will be inferred from the object supplied later. Clients may wish to supply the typename at this point to perform an operation defined in the supertype, or to give the server a chance to see if the named operation is known before sending any more data.

After the server receives the initial line, it sends back:

```
300 [optional text]
```

to indicate that it is ready to receive the rest of the request, or a reply code between 400 and 599 to indicate that it will not carry out the request, and that no more of the request should be sent.

If the server indicates that it will accept it, the remainder of the request is then sent. This consists of the following (in any order), followed by a single line saying “END”.

- An OBJ block (unless the operation is a class operation).

- An optional FMT line.
- Zero or more EXPECT lines.
- Zero or more ARG blocks.

The OBJ block consists of a line with the single word "OBJ", followed by a data description block specifying the object that this operation is being called on. Data description blocks are described below.

An ARG block consists of a line with the single word "ARG", followed by a data description block specifying an argument. The first ARG block specifies the initial argument of the operation, the second the next argument, and so on.

An EXPECT line's first word is "EXPECT". Its second word is a type name, and the remaining words are encoding names. This line indicates that the client will accept a result value with the indicated type, encoded by the indicated encodings. Encoding names are optional, but will be applied strictly left to right if present, with no extra intervening encodings. If the type of the last encoding (or the base type, if no encodings are given) is not self-encoding, the server may apply whatever additional encodings it wishes, to represent the value in a self-encoding type (q.v.).

If multiple EXPECT lines exist, earlier EXPECT lines are considered higher priority than later EXPECT lines. Thus, a client can indicate an order of preference for result types and encodings.

EXPECT lines are strictly advisory. The server may return its result based on one of them if it wishes, or it can ignore them.

A FMT line consists of the word "FMT" followed by one of "VALUE", "REF", or "NOVAL". This is an advisory line, indicating whether the client wants to receive the actual result value of the operation, or just a reference to the result, or wants no value or reference at all. [*Note: This line may become more complex in later versions of the protocol.*] This can be useful when the result of an operation may be large.

Data description blocks

A data description block consists of the following:

- A TYPE line.
- Zero or more ENC lines (which must appear after the TYPE line).
- An optional META block.
- A REF block, a VALUE block, or a line saying "NOVAL". Only one of these may appear, and it appears at the end of the data description block.

A TYPE line consists of the word "TYPE" followed by the type name.

An ENC line consists of the word "ENC" followed by a type name, followed by an encoding name. The ordering of ENC lines is significant; the data in the data description block is of the type given in the TYPE line, encoded as the type given in the first ENC line by the method given in the first ENC line, further encoded in successive ENC lines. If a literal value is being given in this data description block, the last encoding must be in a self-encoding type. The byte-sequence type ("e:byteseq") and its subtypes are self-encoding.

A META block consists of a line saying "META", followed by a data description block describing the meta-data for the object.

A REF block consists of a line saying "REF", followed by a data description block describing the reference. The reference can later be resolved, according to the semantics of the reference type, to yield the value of the specified object.

A VALUE block may take one of two forms. If the encoded value of the object being passed in this data description block consists entirely of non-whitespace printable ASCII characters, and is no more than 200 bytes long, a VALUE block can consist of a single line with the word "VALUE" followed by the encoding of the value of this data description block.

Alternatively, a VALUE block consists of a line with the single word "VALUE", followed on subsequent lines by a delimited block of data. The first character following the end of the VALUE line is the delimiter. The delimiter is usually the `"` character, but can be another character. The data block ends with an unquoted occurrence of the delimiter character. Within the delimited data block, the following quotation rules apply:

- `\d` or `\"` substitutes for the delimiter character within the data block.
- `\q` or `\\` substitutes for the `\` character within the data block.

Note that using `\d` and `\q` instead of `\\` and `\"` will cause recursive applications of the substitution to grow linearly rather than exponentially (as long as the delimiter character is not 'd' or 'q').

The delimited data block is followed by a CR-LF.

Final reply to OPER request

If the operation is carried out successfully, the server replies in this format:

```
200 [optional extra text]
```

and follows that line with a data description block for the result. If the operation is unsuccessful, the server will return a single line with a reply code from 301 to 599, and an optional string further explaining the reason for failure. The following codes are currently defined:

301	Use a different server (followed by servers to use).
400	Bad request, or syntax error.
401	Unknown type.
402	Invalid inputs to operation (precondition failure).
404	Object not found.
410	The request is forbidden to you.
500	Generic server-side error.
501	Request unsupported here.
502	Object operation not implemented here.
503	System resources exhausted.

If the reply code is 301, it will be followed on the same line by one or more parenthesized tuples, each indicating the location of a server. The tuples consist of the hostname of the suggested server, followed by a space, followed by the port number of the server. Material outside the parentheses is ignorable, and parentheses cannot be nested. Multiple parenthesized tuples indicates that the client can choose among multiple servers to carry out the request (with preferred servers coming earlier).

Other codes may be defined later.

There is currently no exception model defined other than the use of these return codes. The '402' message can be used for precondition exceptions.

A.3.5 ATTR: Fetches an object attribute

The ATTR request returns a particular attribute of an object. The format of the request and reply are the same as for OPER, except that no ARG blocks are used.

A.3.6 CNVT: Gets an equivalent of an object in a new type or encoding

CNVT requests an object corresponding to the supplied object, but in a format compatible with one of the EXPECT lines supplied. The format of the first line is:

```
CNVT [optional-conversion-methodname] [optional-original-typename]
```

The format of the rest of the request, and of the server replies, is the same as for OPER, except for the following:

- No ARG blocks are used.
- At least one EXPECT line is required. A successful conversion must conform to one of these EXPECT lines.
- An optional TOP line may be included (before the end of the request). This consists of the word "TOP" followed by a type name. This means that the original object and the converted object must be indistinguishable from the perspective of the type given in the TOP line. (This type is also known as an *intersubstitutable* type.)

If a conversion method name is given, that method must be used for the conversion. Otherwise, the server can choose any method, or sequence of methods, satisfying the request. If the server cannot satisfy the request, it returns the error code 502 (object operation not found).

A server need not support CNVT at all (unless it's a type broker.) If CNVT is not supported at all, the error code 501 (request not supported here) is returned after the first line of the request. (If the request is okay, the code 300 is returned after the first line, as with OPER.)

A.3.7 TYPQ: Gets information about a type

The TYPQ request returns an object giving information about a type. The format of the request is:

```
TYPQ <typename>
```

The format of the reply to a successful request is

```
200 [optional text here]
```

followed by a data description block for an object describing the type. Type descriptions are described in more detail elsewhere.

A server does not need to support TYPQ at all, unless it is a type broker. It should return a response code of 501 if TYPQ is not supported, and 404 (Object not found) if it cannot find an object describing the requested type.

A.3.8 TYPL: Lists types that have been registered or changed recently

The TYPL request returns an object giving a list of all types, or of recently changed types. The format of the request is either:

```
TYPL
```

or

```
TYPL <date> <time> [GMT]
```

TYPL by itself returns a list of all the types registered at the server. With a date and time code, it returns a list of all the types whose descriptions have changed since the date and time given.

The date code is sent as 8 digits in the format YYYYMMDD, where YYYY is the year, MM is the two digits of the month (01 = January .. 12 = December) and DD is the day of the month (with leading zero, if appropriate).

The time code is sent as 6 digits in the format HHMMSS with HH being hours on the 24-hour clock, MM minutes 00-59, and SS seconds 00-59. The time is assumed to be in the server's timezone unless the token "GMT" appears, in which case both time and date are assumed to be using GMT.

If the operation is carried out successfully, the server replies in this format:

```
200 [optional extra text]
```

and follows that line with a data description block for the result. If the operation is unsuccessful, the server will return a single line with a reply code from 400 to 599. If the operation is successful, but no types meet the criterion, the operation will return a data description block with an empty-string value.

The encoding used in the data description block should consist of a sequence of type names separated by CR-LF. There may also be a CR-LF at the end of the value, but this is not necessary. Types known under multiple names may be listed under one or more of those names.

A.3.9 REGI: Registers information about a type

The REGI request registers information about a type. Only type brokers support this command. REGI can be used to register new types, aliases, type attributes or operations, encodings, conversions, or agents that can work with a type.

The REGI command gives no guarantees that the type broker will update its type database in any way, nor does it make any guarantees about when the database will be updated. Some kinds of registry information may be automatically processed by brokers; others may be handled in batches, or with human intervention.

The format of the request begins with the initial line:

```
REGI <kind> <typename> [additional arguments]
```

where *kind* indicates the kind of registration: one of `type`, `alias`, `supertype`, `attribute`, `operation`, `encoding`, or `agent`. The *typename* argument indicates the name of the type for which this information is being registered. The number and nature of the additional arguments depends on the kind of request.

The `alias` and `supertype` registration requests consist of a single line. For the other requests described here, the server will send back a reply code between 400 and 599 if it rejects the request and does not wish to receive the rest. Otherwise, it will send back

```
300 [optional text]
```

after receiving the initial line. The client should then complete the request with an appropriate data description block.

The formats of the different registration requests are as follows:

```
REGI alias <typename> <alias>
```

This is a single-line request for `alias` to be an alternate name for an existing type with the name `typename`.

```
REGI supertype <typename> <supertype>
```

This is a single-line request to register the type named `supertype` as a supertype for the type named `typename`. Both types must already have been registered. The two types must satisfy the supertype relation as defined in the TOP type model (described elsewhere).

```
REGI type <typename>
```

This is the beginning of a request to register a new type named typename. If the server indicates that the request should continue, the client sends a data description block for the type description object. (This object conforms to the type description for the "s:typedefn-0.2" type.)

```
REGI attribute <typename> <attributename>
```

This is the beginning of a request to register a new type attribute named attributename on the type named typename. If the server indicates that the request should continue, the client sends a data description block for the attribute description object.

Because the basic definition of types cannot change once a type is registered, the attribute must be derivable from already existing attributes and operations. The thesis document explains this in more detail.

```
REGI operation <typename> <opname>
```

This is the beginning of a request to register a new type operation (method) named opname on the type named typename. If the server indicates that the request should continue, the client sends a data description block for the operation description object.

Because the basic definition of types cannot change once a type is registered, the operation must be derivable from already existing attributes and operations. This was explained in more detail in the main body of the thesis.

```
REGI encoding <typename> <encname>
```

This is the beginning of a request to register a new encoding names encname on the type named typename. If the server indicates that the request should continue, the client sends a data description block for the encoding description object. This block includes both the name for the encoding and the name for the type in which the encoding is represented.

```
REGI conversion <typename> <cnvtname>
```

This is the beginning of a request to register a new conversion named cnvtname on the type named typename. If the server indicates that the request should continue, the client sends a data description block for the conversion description object.

```
REGI agent <typename>
```

This is the beginning of a request to register an agent for computing an attribute, operation (method), or conversion associated with the type named typename. If the server indicates that the request should continue, the client sends a data description block for the agent description object. (This object will designate the attribute, operation, or conversion the agent implements, as well as designating the agent itself.)

A.3.10 AUTH: Gives authorization

AUTH requests an authorization from the server. An authorization essentially establishes certain information about the session. Authorizations can be used to control access to certain resources or commands. For example, registration of certain kinds of type information may be restricted to clients that have the authorization specified in that type. Authorizations can also be used simply to record certain information about the client, such as its identity or its proxies.

The format of an first line is:

```
AUTH <authname> [optional-authenticating-information]
```

where `authname` is the name of the desired authorization. A given client may have multiple authorizations at once, each with a different name.

The server will not necessarily grant any authorization asked for. Some authorizations may require extra authenticating information, such as a password. This information can be provided after the authorization name.

The server will respond to an authorization request with a line starting with one of the following reply codes, or one of the generic server error codes (500-599).

200 The requested authorization has been granted.

202 The authorization request has been recorded, but has not been either granted or denied. This return code is often used for authorizations that register information, but are not meant to grant special access.

310 The requested authorization has been challenged. In order to be granted, the client needs to provide additional information, usually through a subsequent AUTH command. The contents of the reply line following the initial '310' may contain additional instructions indicating what information should be in the client's subsequent AUTH command. TOP does not specify anything regarding these instructions; rather, it is up to the client to know (possibly through prior arrangements with the server) how to respond to the server's challenge. (Examples of responses might be a fixed password, or an encryption of something in the challenge via a key.)

400 There is an error in the syntax of the authorization request, or the server does not implement authorizations

410 The requested authorization has been denied.

A.4 An example session

The following transcript shows the protocol being put through its paces. Text in *italics* is issued by the client; text in `courier` comes from the server. **Boldface** indicates commentary.

```
PROTO TOP/0.2
201 TOP/0.2
NOOP Hello there!
200 Hello there!


TYPQ e:int
200 Type description object follows.
TYPE net:typename-060394@gs1.sp.cs.cmu.edu
ENC e:text oracle-protocol
VALUE
"NAME e:int
SUPER e:obj
[semantics and some operations omitted]
OPER e:int plus {
ARG e:int arg
SEM
\dReturns the sum of the supplied object and the argument.\d
}

ENC e:text ascii-rep [One encoding defined for ints]
" [End of the VALUE block for the type description object]


OPER plus
300 Send object and parameters.
OBJ
TYPE e:int
ENC e:text ascii-rep
VALUE 9
ARG
TYPE e:int
ENC e:text ascii-rep
VALUE
”87”
END
200 Value follows.
TYPE e:int
ENC e:byteseq ascii-rep
VALUE 96
```

[The next section involves an attempt to uncompress a file retrieved from a Web server. The file is compressed, and (when uncompressed) ends with a carriage return-newline.]

```
CNVT
300 Send object and parameters.
OBJ
TYPE e:text
ENC e:byteseq unix-compress
REF
TYPE s:url
ENC e:text standard
VALUE http://www.cs.cmu.edu/~spok/foo.txt.Z
EXPECT e:text
END
200 Value follows.
TYPE e:byteseq
VALUE
"These are the times that try to trick the transcripts.
"


QUIT
205 Nice talking to you.
```

[The server terminates the connection.]

A.5 Summary of reply codes

200	Okay.
201	Remainder of this line is the protocol I'm using.
202	Authorization request recorded.
205	Goodbye.
300	Please continue.
301	Use a different server.
310	Authorization request challenged.
400	Bad request.
401	Unknown type.
402	Invalid inputs to operation (precondition failure).
404	Object not found.
410	Request forbidden.
500	Unspecified server error.
501	Request not supported here.
502	Object operation not found.
503	System resources exhausted.

Bibliography

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using style to understand descriptions of software architecture. In *Proceedings of SIGSOFT '93: Symposium on the Foundations of Software Engineering*, December 1993.

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

[All97] Robert Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon University, May 1997.

[AML$^+$93] F. Anklesaria, M. McCahill, P. Lindner, D. Johnson, D. Torrey, and B. Alberti. The Internet Gopher protocol (a distributed document search and retrieval protocol). Internet Request for Comments (RFC) 1436, March 1993.

[B$^+$71] Abhay Bhushan et al. The file transfer protocol. Internet Request for Comments (RFC) 172, June 1971.

[BCGP97] Michelle Baldonado, Chen-Chuan K. Chang, Luis Gravano, and Andreas Paepcke. The Stanford digital library metadata architecture. *International Journal of Digital Libraries*, 1(2), February 1997.

[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.

[BLFF96] T. J. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0. Internet Request for Comments (RFC) 1945, May 1996.

[BLMM94] T. J. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (URL). Internet Request for Comments (RFC) 1738, December 1994.

[Chr96] Tom Christiansen. Why not to use cgi-lib.pl. Available on the World Wide Web at http://language.perl.com/info/www/!cgi-lib.html, 1996.

[DLO92] Peter B. Danzig, Shih-Hao Li, and Katia Obrazacka. Distributed indexing of autonomous Internet services. *Computing Systems*, 5(4):433–459, 1992.

[FGM$^+$97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. J. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. Internet Request for Comments (RFC) 2068, January 1997.

[Fla97] Giovanni Flammia. XML and style sheets promise to make the web more accessible. *IEEE Expert*, 12(3):98–99, 1997.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: or, why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington, April 1995.

[GM95] James Gosling and Henry McGilton. The Java (tm) language environment: A white paper. Available from Sun's Web server at <URL:http://java.sun.com/whitePaper/java-whitepaper-1.html>, 1995.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, MA, 1983.

[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.

[Ini94] Text Encoding Initiative. *Guidelines for Electronic Text Encoding and Interchange.* Text Encoding Initiative, 1994.

[Int97] Internet Assigned Numbers Authority. Media types. Published on the Internet at <URL:ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/media-types>, August 1997.

[Kah91] Brewster Kahle. An information system for corporate users: Wide area information servers. Technical Report TMC-199, Thinking Machines Corporation, Cambridge, MA, 1991.

[KL86] Brian Kantor and Phil Lapsley. Network news transfer protocol. Internet Request for Comments (RFC) 977, February 1986.

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

[McS98] McSpotlight web site. Available on the World Wide Web at http://www.mcspotlight.org/, 1998.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* Prentice-Hall International Series in Computer Science. Prentice-Hall, New York, 1988.

[Moc87] P. Mockapetris. Domain names - implementation and specification system structure and delegation. Internet Request for Comments (RFC) 1035, November 1987.

[Nat95] National Information Standards Organization (U.S.). *Information Retrieval (Z39.50): Application Service Definition and Protocol Specification.* NISO Press, Bethesda, MD, 1995.

[Nat97] National Center for Supercomputing Applications. The Common Gateway Interface specification. Available on the World Wide Web at http://hoohoo.ncsa.uiuc.edu/cgi/interface.html, 1997.

[NKN91] Steven R. Newcomb, Neill A. Kipp, and Victoria T. Newcomb. The 'HyTime' hypermedia/time-based document structuring language. *Communications of the ACM*, 34(11):67–83, November 1991.

[Ock98] John Ockerbloom. The on-line books page. Available on the World Wide Web at http://www.cs.cmu.edu/books.html, 1998.

[PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *Proceedings of ICDE*, Taipei, Taiwan, March 1995.

[Pos82] J. Postel. Simple mail transfer protocol. Internet Request for Comments (RFC) 821, August 1982.

[Pos94a] J. Postel. Domain name system structure and delegation. Internet Request for Comments (RFC) 1591, March 1994.

[Pos94b] J. Postel. Media type registration procedure. Internet Request for Comments (RFC) 1590, March 1994.

[Pro97] Progressive Networks. RealAudio home page. Available on the World Wide Web at http://www.realaudio.com/, 1997.

[PST91] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z*. Prentice Hall, New York, 1991.

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

[SE97] Erik Selberg and Oren Etzioni. The MetaCrawler architecture for resource aggregation on the web. *IEEE Expert*, 12(1):8–14, 1997.

[SEKN92] Michael F. Schwartz, Alan Emtage, Brewster Kahle, and B. Clifford Newman. A comparison of internet resources discovery approaches. *Computing Systems*, 5(4):461–493, 1992.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, New Jersey, 1996.

[Sha95] Mary Shaw. Architecture issues in software reuse: It's not just the functionality, it's the packaging. In *Proceedings of the Symposium on Software Reusability*, April 1995.

[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.

[SM94] K. Sollins and L. Masinter. Functional requirements for uniform resource names. Internet Request for Comments (RFC) 1737, December 1994.

[SP94] Michael F. Schwartz and Calton Pu. Applying an information gathering architecture to Netfind: A white pages tool for a changing and growing internet. Technical Report CU-CS-656-93, University of Colorado at Boulder, Boulder, CO, 1994. revised July 1994.

[SPW$^+$96] Katia Sycara, Anandeep Pannu, Mike Williamson, Dajun Zeng, and Keith Decker. Distributed intelligent agents. *IEEE Expert*, December 1996.

[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.

[Thi96] Robert Thibadeau. Digital labels for digital libraries. *D-Lib Magazine*, October 1996.

[WCS96] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly and Associates, Sebastopol, CA, second edition, 1996.

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