More Flexible Data Types

Mike Spreitzer † (spreitze@parc.xerox.com)
Andrew Begel †‡ (abegel@cs.berkeley.edu)

Copyright 1998 Mike SpreitzerXerox Corporation; all rights reserved.

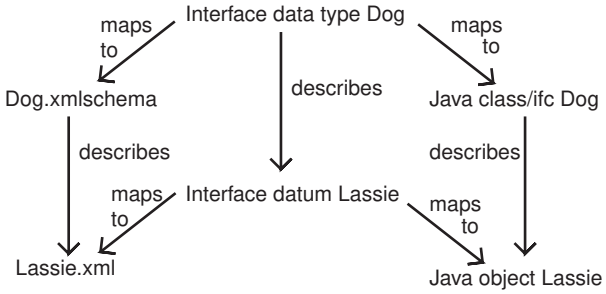
† Xerox Palo Alto Research Center
‡ University of California, Berkeley

Abstract

XML, as "the next generation ASCII", can play several roles in a distributed object system; one of the more exciting ones is as the basis for serialized data representations. This is exciting because XML-encoded data can be more self-describing than data encoded in many more traditional ways, which facilitates the kind of decentralized protocol evolution seen in Internet-scale development: XML's explicit "tagging and bagging" helps keep multiple extensions straight. However, today's common distributed object systems have type systems that are not flexible enough to describe such data. We suggest a way to make more flexible data types; this improves distributed object systems in general, and is critical to realizing XML's full potential. This approach has: (1) typing judgements based on type structure instead of type identity, (2) extensible record types with optional fields, (3) coarse record types, for which extension is compatible with subtyping, and (4) non-ignorable fields in record values.

1 Introduction

The following figure shows six data objects and descriptions that illustrate various roles for data and data descriptions in distributed object systems.



Many distributed object systems (such as CORBA [CORBA], DCOM [DCOM], and Java RMI [JavaRMI]) have an *interface language*, in which various data types can be defined. These systems then have standard mappings from the interface language to a serialized data representation "on the wire" and into various programming languages. The data architecture depicted above has the virtue that the interface data and types form a "wasp waist" that can join various data representations on the left with various programming language presentations on the right without a multiplicative explosion of complexity. This relationship has much more to offer than this other common vision:

… which is often extolled for its simplicity but is also limited by that simplicity.

As suggested by the first figure, XML can be involved in multiple ways:

1. the serialized data representation can be based on XML;

2. an interface data type might be mappable to an XML data type in some XML schema language(s); and

3. the interface language might be based on XML or an XML schema language.

The interface data types and XML data types can be connected in various ways. They all take the vertical arcs as fixed, but vary on the treatment of the angled arcs. In the simplest treatment, the way interface data are encoded into XML is fixed. That is, there is a generic (i.e., independent of the particular interface at hand) set of tags for marking up data such as records, arrays, numbers, and so on. In a more sophisticated treatment, the tags are specific to individual interface data types, but the encoding technique is still fixed. In a very sophisticated treatment, it would be possible for the interface author to exercise some choices in the way the data are encoded into XML. In general, the mapping between the interface data types and the XML data types may not have full fidelity, so there may be a need for two independent documents or one combined document.

The first role is generating a lot of excitement, because

it promises better support of decentralized evolution through XML's self-descriptive powers. However, the type system of the interface language must be unusually flexible in order for XML's promise for data representation to be realized. This paper illustrates the problem and shows how to make sufficiently flexible type systems.

Protocols deployed on a wide scale (such as the Internet) evolve in a particularly challenging way. In contrast with smaller deployments, the evolution of widely deployed protocols is not managed by a single engineering organization. Rather, many independent shops develop evolutionary changes to the protocol, and then incrementally deploy these changes into the existing system. Evolutionary changes may be either "optional", offering some degree of forward/backward compatibility, or "mandatory", meaning both sides of a conversation must support the change. Incremental deployment of independent evolutionary changes poses interoperability problems, both singly and in combination. The incremental deployment of a single evolutionary change presents the challenge of not only enabling new clients to work with new servers but also (at least in the cases where that makes semantic sense) enabling old clients to work with new servers and/or new clients to work with old servers. The kinds of data typing seen in existing distributed object systems such as CORBA, DCOM, and Java RMI have technicalities that make them unable to say much about data that follow an evolutionary pattern; an extended example appears in the next section. Worse yet, for popular and important protocols (such as the World Wide Web's HTTP [HTTP/0.9, HTTP/1.1]), at any given time there are several independently developed evolutionary changes in the process of being incrementally deployed. This presents a further interoperability challenge. In general, a given client and sever that wish to interact each support a different set of extensions, and it is desirable for their interaction to use the intersection of those sets. Using existing data type systems to do the necessary ad hoc negotiations makes for messy application code, and possibly additional latency; the example in the next section also illustrates these problems.

This paper shows an approach to creating more flexible data types for use in application-level network interface definitions. Using these more flexible data and types, developers can usefully describe their data and the degrees of forward/backward compatibility desired. These ideas could be applied in future versions of existing network interface definition languages.

The flexible data types that we show how to create are useful in more architectural contexts than the one suggested by "network interface". In particular, this flexibility is valuable when the producer and consumer are separated by time or access control and thus unable to negotiate (which rules out certain alternative solutions). This flexibility is also valuable when there isn't a single consumer for a given datum, for which negotiation-based solutions are (at best) problematic. Common examples include event distribution systems and database systems.

This paper is organized as follows. In section 2 we examine the subtyping-based technique for decentralized evolution and note that it doesn't scale well. In section 3 we outline our approach to making data types more flexible. We conclude with some brief remarks on related and future work in section 4.

2 Evolution by Subtyping

Let us consider what happens if we try to use subtyping to facilitate protocol evolution. Suppose a system starts out using the following type for its distributed objects:

O1 = {m: {a:A} → {x:X} }

An optional extension that adds an argument and a result makes a new type like this:

O2 = { m: {a:A} → {x:X},
n: {a:A, b:B} → {x:X, y:Y} }

The new type O2 is a subtype of the original type O1; for this we write

$O2 <:_s O1$

Old and new servers and clients have the following types:

old_server: O1
new_server: O2
old_client: O1 → T
new_client: O2 → T

There are four possible ways to combine these servers and clients.

```
old_clnt(old_srvr);     //works
new_clnt(new_srvr);     //works
old_clnt(new_srvr);     //works too!
new_clnt(old_srvr);     //doesn't work!
```

The last combination doesn't work because

`old_srvr` doesn't have the type required by `new_clnt`. So we have to write clients in a more complicated style. Here's the new client in that style (in C syntax):

```
T new_client(O1 server)
{
if (has_type(server, O2))
  ... narrowto_O2(server)...;
else
  ...server...;
}
```

In addition to the scaling problems to be seen below, this type introspection may require round trips between client and server before useful work can be done. This adds latency that some other techniques do not have.

Suppose an independent extension such as this:

O3 = { m: {a:A} → {x:X},
o: {a:A, c:C} → {x:X, z:Z} }

When a developer wants to use both extensions, she might create a type like this:

O4 = { m: {a:A} → {x:X},
n: {a:A, b:B} → {x:X, y:Y},
o: {a:A, c:C} → {x:X, z:Z},
p: {a:A, b:B, c:C} → {x:X, y:Y, z:Z} }

... and here's what a client cognizant of that combination would have to do:

```
T client(O1 server)
{
if (has_type(server, O4))
  ...narrowto_O4(server)...;

else if (has_type(server, O3))
  ...narrowto_O3(server)...;

else if (has_type(server, O2))
  ...narrowto_O2(server)...;

else
  ...server...;
}
```

In general, a client or server that understands N extensions has source code of size 2^N, or even greater. In many current distributed object technologies (including CORBA, DCOM, and Java RMI), types have identities as well as structure. If another developer were to independently write down a type with the same structure as O4, it would still be a different type; this developer's clients and servers would not fully interoperate with O4 clients and servers, because they would not recognize each other's combined types. To get full interoperability, the two developers would have to go to a global engineering body, such as the W3C, IETF, ISO, or ITU, for a single global declaration of the combination. This imposes a significant additional delay in both development and deployment projects, simply to satisfy certain technicalities of the object system.

Mandatory extensions do not suffer the exponential blow-up of code complexity due to all the possible combinations, but they do still suffer (in current systems) from the problem of agreeing on the identity of each combination. For example, if both of the above extensions were "mandatory", their combination would create a type like

O4' = { p: {a:A, b:B, c:C} → {x:X, y:Y, z:Z} }

… and the client code would also be simpler:

T client(O4' server) {…}

But in systems where types have identities, there would still be a need for a global agreement on an identity for the O4' structure.

Evolvable data *can* be described using existing interface languages — just not very precisely. Specifically, the extensible data can be described as tree structured "property lists" (also called "association lists" or "attribute lists"). As this is close to XML's information set, it is unsurprising to see excitement about using XML for representing evolvable data.

3 New Solution

To overcome the difficulties just explored, we use a more flexible data and type system. This system is unusual in four ways.

1. Type relations are judged on structure, not identity.

2. Each field of a record type has a "mode" flag that indicates whether the field must be present in corresponding record values.

3. There are some unusually coarse types, to cover evolutionary changes that would not otherwise create subtypes.

4. Each field of a record value is marked either 'ignorable' or 'non-ignorable'.

We will examine each of these three features in turn. In the course of doing so, we will note three relations among types: the familiar subtyping relation ($<:_s$), the new *extension* relation ($<:_e$), and the new *refinement* relation ($<:_r$). Extension is for evolutionary changes that are co-variant at procedure types (as opposed to the usual contra-variance for subtyping). Refinement is the transitive closure of the union of subtyping and extension.

As discussed above, we judge type relations based on structure rather than identity, so that developers do not need a global standardization body to produce interoperable clients and servers of combined extensions.

The next unusual feature is optional fields. These correspond to optional subelements in an XML element content model, to optional headers in HTTP, and so on. Using this feature, we can describe the first example extension like this (for succinctness, we suppose that the "optionality flag" defaults to "non-optional" when not explicitly written):

O2 = {m: {a:A, b:optional:B}
→ {x:X, y:optional:Y} }

An XML content model for an XML representation of m's request messages might be "A, B?".

This can be formalized as follows. A general record type looks like this:

$\{l_1:m_1:T_1, \ldots l_n:m_n:T_n\}$

where each l_i is a label (i.e., field name), m_i is a Boolean indicating whether the field is optional, and T_i is the type of the field. A general record value is also a set of triples:

$\langle l_1:ig_1:v_1, \ldots l_n:ig_n:v_m\rangle$

where each l_i is a label, each ig_i is an "ignorable" bit (see below), and each v_i is a field's value. A given label may not appear twice in a given record type or record value.

Every record type is implicitly extensible; a given record value (of the form above) has a given record type (of the form above) when:

- for all $1 \le i \le n$: $m_i \vee$ the record value has a field labelled l_i; and
- whenever l:ig:v appears in the record value and l:m:T appears in the record type, v has type T.

While many other type systems allow construction of what users may think of as "optional types" (e.g., "datatype OFoo = present of Foo | absent" in ML), by putting recognition of the concept of optionality into the type system we get additional flexibility. For example, a record value of the form "⟨a:v⟩" can have type "{a:A, b:optional:Foo}", whereas it can't have a type of the form "{a:A, b:OFoo}".

The rules for subtyping and extension are as follows. Record type

$R2 = \{l_1:m'_1:T'_1, \ldots l_{n+k}:m'_{n+k}:T'_{n+k}\}$

is a subtype of record type

$R1 = \{l_1:m_1:T_1, \ldots l_n:m_n:T_n\}$

when

$T'_i <:_s T_i$ for all $1 \le i \le n$; and
$m'_i \Rightarrow m_i$ for all $1 \le i \le n$.

The rule for extension is analogous: $R2 <:_e R1$ when

$T'_i <:_e T_i$ for all $1 \le i \le n$; and
$m'_i \Rightarrow m_i$ for all $1 \le i \le n$.

It is at procedure types that extension and subtyping differ:

$(T' \to U') <:_s (T \to U)$
when
$(T <:_s T') \wedge (U' <:_s U)$

and

$(T' \to U') <:_e (T \to U)$
when
$(T' <:_e T) \wedge (U' <:_e U)$

As usual, subtyping is reflexive and transitive; extension and refinement are too.

Thus, the request record type of O2.m is both a subtype and an extension of the request type of O1.m; the same is true of the response record types. As this is co-variant, O2.m's type (a procedure type) is an extension, but not a subtype, of O1.m's. Consequently, O2 is an extension, but not a subtype, of O1.

Since extension does not necessarily imply subtyping, we would have a very awkward type system if we stopped here. A client of type "O1 → T" could not be applied to a server of type O2! To solve this problem, we introduce "coarse types". Each *coarse type* is associated with an "ordinary", or *fine*, type; we write "[T]" for the

coarse type associated with fine type T. A coarse type [T] is the (untagged) union of all the refinements of T. That is, every value of every type U that is a refinement of T is also a value of [T]. In terms of type relations, this means that whenever

$$T' <:_r T$$

we also have

$$[T'] <:_s [T] \wedge T' <:_s [T]$$

In the example above, O2 is a refinement of O1, so O1, O2, and [O2] are all subtypes of [O1], and thus a client of type [O1]→T is applicable to a server of type O2.

In effect, coarsening a type removes some information. For a coarse record type R = [{a: A, …}], what is known about its "a" field is only that it has the coarse type [A]. For a coarse procedure type [A→B], what is known is only that the domain is some refinement of A and the range is some refinement of B --- an invocation with a general A value (1) might fail due to a mismatch with the actual domain of the procedure value, and (2) might return a value that does not have type B. For example, if we constructed the following mandatory extension of O1

$$O2' = \{m: \{a:A, b:B\} \rightarrow \{x:X, y:Y\}\ \}$$

we could statically pass the value ⟨a:anA⟩ to an invocation of the m method of an object of coarse type [O1] --- but the invocation would fail at runtime if the object had type O2'.

The final unusual feature, which supports "mandatory" extensions, is that each field of a record value is marked either as "ignorable" or as "non-ignorable". For example, in an XML encoding, we might suppose that each record field would be a distinct XML element and define a standard attribute (say, "xmlignorable") to carry the "ignorable" bit. If the first extension above were mandatory and the second optional, we might find a request message like this:

```
<O2.m.request>
    <A xmlignorable="false">…</A>
    <B xmlignorable="false">…</B>
    <C xmlignorable="true">…</C>
</O2.m.request>
```

The "ignorable bit" need not explicitly appear in every element: it could have a default value, or it's value could be implied by the schema for the particular message at hand.

When a record value field is marked "non-ignorable", this means it must ultimately be "understood" --- to at least some degree --- by the receiver. A complication is that we should allow the receiver to delegate partial or complete responsibility for this understanding (e.g., as a WWW proxy would). Thus, rather than build in a fixed policy for testing understanding (e.g., into the parameter passing mechanism), we make it the application's responsibility to eventually test that it understood each piece of input, and this testing responsibility can be delegated along with the understanding responsibility. We suppose there is an understanding testing primitive available, which tests a record value (the input whose understanding requirements are to be tested) against a record type (listing all the fields understood) to see if every non-ignorable field in the value is mentioned in the type. For example, a server that implements the first extension directly, and delegates nothing, would test its input against O2; a proxy that delegates all understanding would not test any understanding at all.

As an example of how non-ignorable fields can be used, consider how to extend the proxy-ignorant protocol of O1 with proxying functionality (as in the WWW).

$$Req = \{a:A, pi:optional:ProxyInstructions\}$$

$$ProxyInstructions = \{origin:[O1]\}$$

$$P = \{m: Req \rightarrow \{x:X\}\ \}$$

A server of type P is willing to either serve a request directly (if no ProxyInstructions appear in it) or act as a proxy for some other server (pi.origin). A proxying-aware client can pass either an unextented request ⟨a:false:anA⟩ or an extended request ⟨a:false:anA, pi:false:⟨origin:false:q⟩⟩ to a server of type [O1]. Without the "ignorable" bit mechanism, a client making a proxying request would have no assurance that the request will be properly interpreted: a plain old O1 server would simply ignore the unrecognized "pi" field and process the request itself. With the added mechanism, the client is assured that the server will return an error if it does not recognize the fact that it is being asked to proxy.

Note that this example can correctly encompass the hop-by-hop vs. end-to-end distinction. Hop-by-hop extensions can be added to the ProxyInstructions, which the proxy removes from the record before forwarding; the end-to-end extensions can go anywhere else in the request. A similar structure could be established for the response record type.

4 Related and Future Work

Extensible records are common in the literature of programming language design [Mitchell, ch 10]. They are involved in several hard problems that do not appear here. One of those is type inference, which is clearly not relevant. Another is efficient compilation; the practical problem for an interface language is rather mapping into various programming languages of interest. Another hard problem is how to explain implementation inheritance in object-oriented languages; again, that is not important for interface languages. In fact, it has long been recognized that interface inheritance and implementation inheritance are two separate things [Cook]; this has even been realized to a degree in Java.

The marking of extension fields with a "non-ignorable" bit is widely known; for example, it has been suggested for HTTP [HTTP-ext].

The OMG has recently adopted a limited form of extensible records, known as "objects by value" or simply "value types", into CORBA; the limitation is that only single inheritance is allowed.

The W3C's XML Schema Working Group [XML-Activity] is working on producing a new schema language for XML. However, they have not adopted any requirement for an XML Schema to also be able to serve as, or contain, a data type declaration in a higher-level interface language.

Several things remain to be done. One item of high importance is to work out good mappings into various programming languages of practical interest; another is to fully work this approach out in a fully fleshed-out type system. In particular, it would be good to actually integrate it into the type system(s) of CORBA, DCOM, and/or HTTP-NG [HTTP-NG].

Another important task is to incorporate ordering information among the fields in an extensible record. In so doing, we make this approach usable not only for sets of orthogonal extensions, but also for sets that semantically interfere to the degree that can be untangled by simply specifying an ordering for the extension usages. Experience with HTTP shows that this would be a useful enhancement.

It would be interesting to try to formulate this paper's approach using the "mix-in style" of inheritance [Bono].

5 Acknowledgments

This work has benefitted from discussions within the HTTP-NG project and with colleages at Xerox PARC. In particular, we would like to thank Paula Newman, Henrik Frystyk Nielsen, Jim Gettys, Bill Janssen, Dan Larner, John Lamping, Larry Masinter, and Dan Swinehart.

6 References

[Bono] Viviana Bono et. al. A Core Calculus of Classes and Mixins. Proc. 13th European Conference on O-O Programming. June 1999.

[Cook] William R.Cook, Walter Hill, and Peter S.Canning. Inheritance is not subtyping, in Proc. of the 17th POPL , 1990, Page 125.

[CORBA] Object Management Group. Common Object Request Broker Architecture. http://www.omg.org.

[DCOM] Microsoft Co. Distributed Component Object Model Protocol (DCOM/1.0). http://premium.microsoft.com/msdn/library/techart/msdn_dcomprot.htm. January 1998.

[HTTP/0.9] Berners Lee, T. HTTP/0.9 Specification. http://www.w3.org/Protocols/HTTP/AsImplemented.html . 1991.

[HTTP/1.1] IETF HTTP Working Group. HTTP/1.1 Specification. http://www.w3.org/Protocols/rfc2068/rfc2068. January 1997.

[HTTP-ext] Henrik Frystyk Nielsen, Paul Leach, Scott Lawrence. HTTP Extension Framework (work in progress). Internet Draft draft-frystyk-http-extensions-03. March, 1999.

[HTTP-NG] World Wide Web Consortium. HTTP-NG Project home page. http://www.w3.org/Protocols/HTTP-NG/.1999.

[Mitchell] John C. Mitchell. Foundations for Programming Languages. MIT Press, 1996.

[JavaRMI] Sun Microsystems Co. Java Remote Method Invocation Specification. http://java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html.1996.

[XML-Activity] W3C. XML Activity home page. http://www.w3.org/XML/Activity.html. 1999.