Declarative mediation in distributed systems[⋆]

Sergey Melnik

Stanford University, Stanford CA 94305 USA[⋆⋆]
melnik@db.stanford.edu

Abstract. The mediation architecture is widely used for bridging heterogeneous data sources. We investigate how such architecture can be extended to embrace information processing services and suggest a framework that supports declarative specification of mediation logic. In this paper we show how our framework can be applied to enrich interface descriptions of distributed objects and to integrate them with other client/server environments.

1 Introduction

More and more information processing services are becoming available online. Such services accept data, process it, and return results. A variety of services like summarizers, indexers, report generators, calendar managers, visualizers, databases, and personalized agents are used in today's client/server systems. As more such components are deployed for use, the diversity of program-level interfaces is emerging as an important stumbling block. Interoperation of heterogeneous information processing services is hard to achieve even within a given domain like digital libraries [13].

The mediation architecture [17] has often been used for leveraging solutions for the interoperability problem. It introduces two key elements, wrappers and mediators. The wrappers hide a significant portion of the heterogeneity of services, whereas the mediators perform a dynamic brokering function in a relatively homogeneous environment created by the wrappers.

Frequently, mediation is implemented on top of distributed object architectures like CORBA or DCOM. Typically, a wrapper acts as a server object and provides a standard interface through which mediators can access heterogeneous components. This solution works well in environments targeted at querying of data sources. The reason for this is that it is relatively easy to develop a common querying interface that has to be supported by all wrapped sources. Serious complications arise, however, when the underlying components support a rich set of interfaces and protocols. In this case, even if the individual components are wrapped by distributed objects, their interfaces remain very diverse. Thus, mediators become more detailed and complex, expensive to create and maintain.

[⋆] This work was supported by a German Research Council fellowship and by the NSF grant 9811992

[⋆⋆] Permanent address: Leipzig University, Augustusplatz 10, D-04109 Germany

In this paper we describe a framework tailored for declarative specification of mediation logic that is required to integrate heterogeneous information processing services. We examine an environment in which services expose rich interface descriptions and mediators are specified using declarative languages. Such environment promises significant advantages over hard-coded mediators [14, 18]. In fact, developing mediators for disparate systems becomes an engineering task leveraging established formal methods as opposed to error-prone programming.

Although declarative mediation promises substantial benefits, it may introduce penalties in efficiency and additional complexity. Nevertheless, our initial experience suggests that the framework offers substantial flexibility that can be exploited in different application scenarios. In [10] we describe one such scenario based on heterogeneous retrieval services. In this paper we investigate how our approach can be applied to enhancing the distributed object technology and bridging it with other client/server environments.

The next section introduces a sample scenario that we use throughout the paper to illustrate the major tasks needed to implement mediator systems based on declarative specifications. Sect. 3 introduces canonical wrappers that provide mediators with logical abstractions of the components. Sect. 4 gives an overview of our approach to declarative mediation. In Sect. 5 we elaborate on the techniques that can be used to manipulate the content of the messages exchanged by heterogeneous components. Sect. 6 sketches our approach to representing the dynamic aspect of mediation, i.e. how message sequences originating at one component can be translated into message sequences expected by another component. Interface descriptions of services are examined in Sect. 7. Sect. 8 summarizes the challenges of building declarative mediators. Sect. 9 describes the execution environment used for our running example. Related work is discussed in Sect. 10.

2 Running Example

A typical operation needed for a digital library is the document conversion between different formats like PostScript, PDF, plain text etc. A rudimentary conversion model can be described by an operation which accepts source and destination format specifications and a sequence of bytes as the content of the document. The result of the conversion is a byte sequence in the destination format. In following, we examine two rather obvious implementations of the conversion, one as a CORBA object and another as a Web form. A CORBA client intending to use an HTTP-based service faces a number of obstacles that we sketch below.

For the CORBA implementation of the service it may make sense to provide a `BLOB` interface to large binary objects in order to enable the server to determine the size of the object to be converted in advance and fetch its pieces incrementally. A likely CORBA specification of the conversion service comprises two interfaces:

```
interface Converter {
  BLOB convert(in BLOB doc, in int sourceFormatID, in int destFormatID); }
```

```
interface BLOB {
  long getSize();
  sequence<octet> getBytes(in long start, in long end); }
```

A conventional Web form for an HTTP-based conversion service includes two fields, say `from` and `to` identifying the source and destination format and a `file` field which allows the user to upload a file to be converted from the local disk.

To enable the CORBA-based client to utilize the HTTP-based server, an intermediate component (mediator) is required. In our example, such mediator translates requests between two services. The translation could be achieved using the following algorithm:

1. Receive the request parameters `doc`, `sourceFormatID` and `destFormatID` via the `Converter` interface
2. Translate the format identifiers `sourceFormatID` and `destFormatID` into corresponding format strings `from` and `to` for the Web-based service
3. Retrieve the size of the source object using `doc.getSize()`
4. Retrieve the binary content of the source object via `doc.getBytes()` to fill the `file` field of the HTML form
5. Emit an HTTP POST request after completing the appropriate form fields
6. Create a BLOB instance for the binary data contained in the HTTP reply.

Our goal is to capture the translation between the CORBA client and the HTTP service in a declarative fashion. Such declarative specification would describe how the messages originating from the client are transformed into the messages understood by the server, and the other way around. For that, the mediator needs to be able to manipulate the content of the messages and the order in which they are exchanged.

3 Canonical Wrappers

Mediators need a convenient way to manipulate the content of the messages passed back and forth between different components. For our purposes, convenient means that the message manipulation operations can be described in a declarative fashion, ideally without using a programming language like C++ or Java. To do that, we use logical descriptions of the messages encoded as directed labeled graphs. Represented as a labeled graph, the message content can be manipulated using algebraic operations, transformation rules etc.

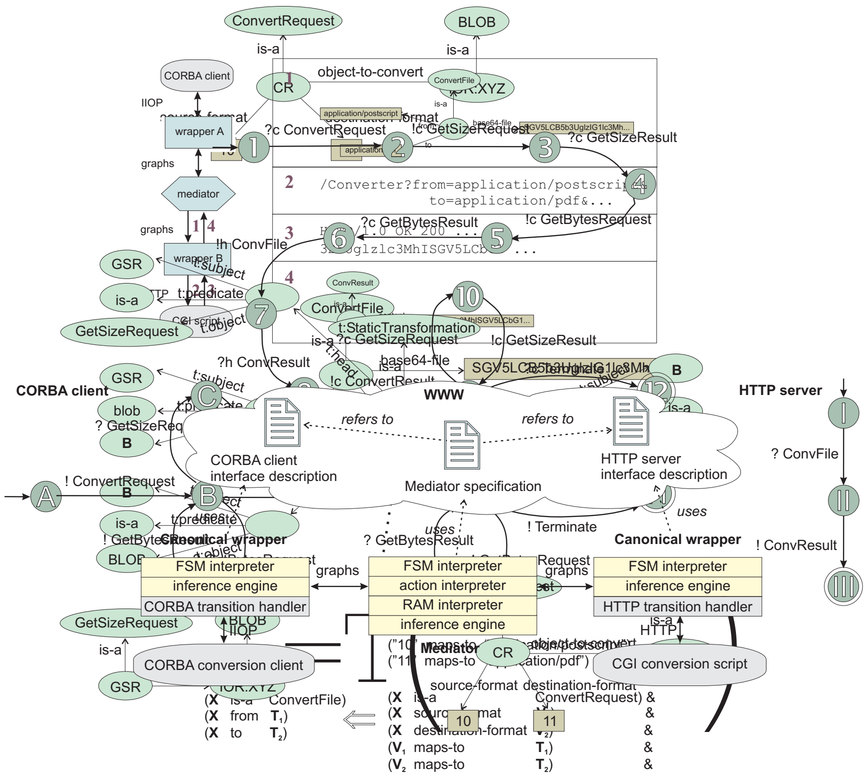
Logical descriptions of messages exchanged by the components can often be derived from their informal descriptions in a straightforward way. Consider how one could formulate a conversion request message as a CORBA invocation:

> 'This is a conversion request specifying which BLOB object to convert (`obj`), and what the source format (`sourceFormatID`) and the destination format (`destFormatID`) of the conversion are.'

This sentence can be represented using the following five logical statements of the kind '`subject predicate object`':

```
CR  is-a                ConvertRequest
CR  object-to-convert   obj
CR  source-format       sourceFormatID
CR  destination-format  destFormatID
obj is-a                BLOB
```

The entity `CR` designates an instance of a CORBA conversion request. The logical description of the request can be represented graphically as a directed labeled graph depicted in Fig. 1. The object reference of `obj` is `IOR:XYZ`, and the source and destination format IDs are 10 and 11 respectively. Ovals represent any entities that might have relationships with other entities. In the figure, such entities are, for example, the concrete BLOB object identified by its object reference, or the type of the object (`BLOB`), or the concept `ConvertRequest`. Arrows represent relationships among entities. They might be conceptual relationships, such as the `is-a`, or 'has-property' relationships, such as `destination-format`. Literals (string values) are depicted in rectangles. The representation used in the figure is similar to an entity-relationship diagram that includes instances of entity types.



ConvertRequest
BLOB
is-a
object-to-convert
CORBA client
IIOP
wrapper A
graphs
mediator
wrapper B
CR
IOR:XYZ
?c ConvertRequest
!c GetSizeRequest
?c GetSizeResult
/Converter?from=application/postscript
to=application/pdf&...
?c GetBytesResult
!c GetBytesRequest
!h ConvFile
?h ConvResult
!c ConvertResult
GSR
GetSizeRequest
CORBA client
WWW
refers to
CORBA client interface description
Mediator specification
HTTP server interface description
HTTP server
? ConvFile
! ConvResult
! ConvertRequest
! Terminate
uses
Canonical wrapper
FSM interpreter
inference engine
CORBA transition handler
action interpreter
RAM interpreter
HTTP transition handler
CORBA conversion client
CGI conversion script
Mediator
source-format
destination-format
10
11
(X is-a ConvertFile)
(X from T1)
(X to T2)
(X is-a ConvertRequest) &
(X source-format V1) &
(X destination-format V2) &
(V1 maps-to T1) &
(V2 maps-to T2) &