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.

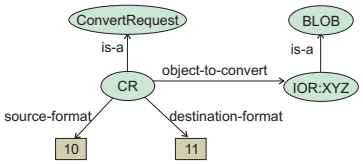

Fig. 1. Logical representation of the CORBA conversion request

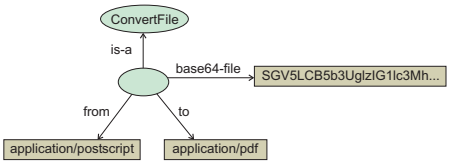

Fig. 2. Logical representation of the HTTP-based conversion request

A logical description of the request makes it possible to abstract out factors irrelevant to the purpose of the service e.g. whether it is implemented as a distributed object or a CGI script. The corresponding HTTP request is represented as shown in Fig. 2. The name of the entity in the middle is not relevant and is omitted for clarity. Note that using logical descriptions allows us to talk about things as different as CORBA calls and HTTP requests in a uniform language.

The mediator and the wrappers used in our running example are depicted on the left-hand side of Fig. 3. Wrapper A acts as a CORBA conversion server for the CORBA client. Logical descriptions of CORBA invocations are passed by wrapper A as object graphs to the mediator. The mediator irons out semantic incompatibilities in the structure and ordering of the messages received from wrapper A and forwards the translated logical representations (graphs) to wrapper B which invokes the corresponding CGI script. In this way, the wrappers A and B allow the mediator to treat the interfaces of the components uniformly. Recall that our example is very simple; a realistic mediation environment contains many more components.

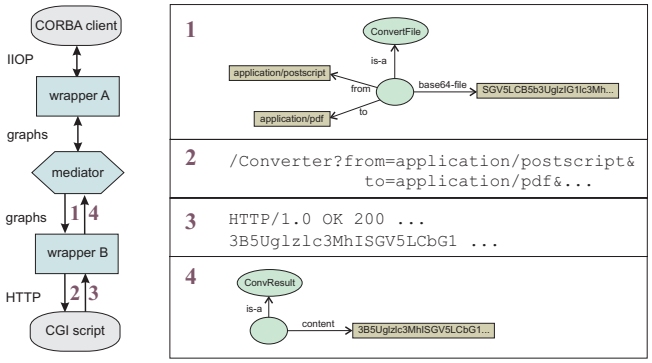

Fig. 3. Canonically wrapped HTTP-based converter

In our approach, wrappers are as simple as possible, and all data conversion and protocol translation logic is embodied in mediators. We call a wrapper *canonical* if it faithfully preserves the semantics of the component and captures information contained in the messages of the component by means of logical descriptions. A canonical wrapper is not required to do any processing beyond trivial syntactic transformation of the messages of the wrapped component.

The right-hand side of the Fig. 3 illustrates a request-reply interaction with the canonically wrapped HTTP-based conversion service. The wrapper accepts a logical description of the message, translates it into a native HTTP request and delivers the response of the component as a set of facts. Note that the wrapper performs no semantic translation of data and protocols used by the component and does not commit to any high-level query language. In this sense, the wrapper is 'thin' and requires a minimal implementation effort.

4 Declarative Mediation

Canonical wrappers like the one described above allow the mediator to manipulate the messages exchanged between the components as directed labeled graphs. To illustrate, consider the following example. In our scenario, one complication the mediator has to face in order to reconcile the representations of the CORBA and HTTP requests are incompatible format specifications. On the Web, PostScript is identified using the MIME type 'application/postscript' whereas for our CORBA service it is the integer 10.

The mapping of the format IDs can be formalized using one rule and two facts illustrated in Fig. 4. Every expression in parentheses is again of the form `(subject predicate object)` where variables are in boldface. Two top statements are facts representing the mapping between the integers and MIME types. The body of the rule (right-hand side) is a conjunction of expressions that produces a set of variable substitutions. The head of the rule (left-hand side) describes how new statements are obtained from the variable substitutions.

Applying this rule to the CORBA conversion request (Fig. 1) delivers almost the desired HTTP request (Fig. 2). The missing element are the bits and bytes

needed to fill the `base64-file` property in the HTTP request from the description (`IOR:XYZ is-a BLOB`). To retrieve the binary content the mediator has to issue at least two additional CORBA invocations, `getSize()` and `getBytes()`.

Hence, to complete the translation we need to specify the dynamic behavior of the mediator, i.e. how a series of messages of one component are mapped into series of messages of another component. We call such mappings *dynamic transformations* as opposed to the manipulations of the content of the messages, which we call *static transformations*.

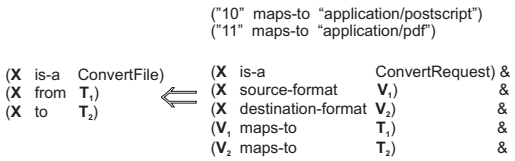

Fig. 4. Specification of the format translation from CORBA to HTTP

The need for dynamic and static transformations suggests that, in general, a number of different formalisms is required to describe the mediator logic. For example, manipulations of the content of the messages can be expressed using Datalog rules like the one depicted in Fig. 4. Transformations of message sequences can be described using finite-state machines, Petri nets etc. Thus, in order to build a fully functional declarative mediator, a variety of questions need to be answered:

- How do we express dynamic and static transformations performed by the mediator? Which formalisms are appropriate for that?
- How can different formalisms be used in the same mediator specification? How can we represent expressions that use different languages?
- How can declarative mediators be executed? Which middleware and APIs are necessary to build wrappers and mediators?

To facilitate mixing and reuse of different formalisms, the declarative languages used by our mediators are represented using a *meta-model* that is capable of capturing and linking expressions in different languages. To illustrate the role of the meta-model, consider that we choose to map the message sequences of the CORBA component to that of the HTTP component using a finite-state automaton. Let the automaton specification be encoded in some XML-based syntax. On transitions from one state to another, the automaton executes actions that manipulate the content of the messages. These actions could be formalized using rules that are stored in an ASCII file. Further imagine that the rule language does not support regular expressions or arithmetic operations, but we need both of them to transform one message represented as a graph into another. Regular expressions happen to use yet another syntax, and the arithmetic operations are implemented by a Java program.

A common meta-model allows us to represent expressions in all these disparate languages in a uniform way and to link them with each other. Our meta-model is based on directed labeled graphs, similarly to how the messages themselves are encoded. To represent expressions in different languages we deploy

the Resource Description Framework [8] that has been designed for the use on the Web. Using a Web-ready meta-model has the advantage that the mediator specifications can be disseminated on the Web and refer to each other. All graph representations used in the figures in this paper are based on the RDF model.

To execute mediators that deploy different formalisms and languages, a comprehensive runtime environment is required. Such environment has, for instance, to make sure that the appropriate interpreters are invoked for the declarative languages used in the specifications, or that the whole specification can be compiled into executable code. We describe the details of language mixing and the runtime environment elsewhere [10]. The following two sections demonstrate different formalisms for implementing static and dynamic transformations that can be used in our sample scenario. We also illustrate how expressions in different languages can be represented using a meta-model. After that, we describe the architecture of the prototype system that supports execution of declaratively specified mediators like the one used in our sample scenario.

5 Static Transformations

To illustrate how different static transformations can be described declaratively, let us return to our running example. After receiving the initial CORBA conversion request, the mediator first needs to retrieve the size of the binary object to be converted. The only parameter required for the `getSize()` invocation is the object reference received in the conversion request. Thus, the corresponding CORBA `getSize()` message can be generated by applying a static transformation T to the received request.

This static transformation T can be expressed using the following rule (let `GSR` be a constant that represents an instance of a `GetSizeRequest`):

```
(GSR is-a GetSizeRequest)
(GSR blob B)              <=  (B is-a BLOB)
(B   is-a BLOB)
```

This rule finds an instance B of type `BLOB` and constructs a new message that contains a reference to B. A possible encoding of the above rule in the meta-model is shown in the top part of Fig. 5. We omitted some portions of the graph like `(B is-a t:Variable)` for better readability. Prefix `t:` denotes specific language elements (vocabulary) used for representing static transformations of this kind. The bottom part of the figure shows the result of applying transformation T to the received request.

The language we use to specify transformations like the one above is a variant of Datalog with negation and allows us to encode arbitrary rules and facts. In this and further examples we are using this and other formalisms like finite-state machines merely for illustrating our approach to declarative mediation. They might not even be the best choice for our running example.

Using the Datalog-based language, more complex functions can be defined which can still be analyzed automatically. For instance, a request that may have an optional parameter of type `PType` can be described using the expression shown below:

```
(X is-a Request) & ( not exists Y (X opt-param Y)  ||
                    ((X opt-param Z) & (Z is-a PType)) )
```

For a graph representation of a message, this condition is satisfied only if every node of type `Request` either has no arc labeled with `opt-param`, or the node reachable via `opt-param` is marked to be of type `PType`.

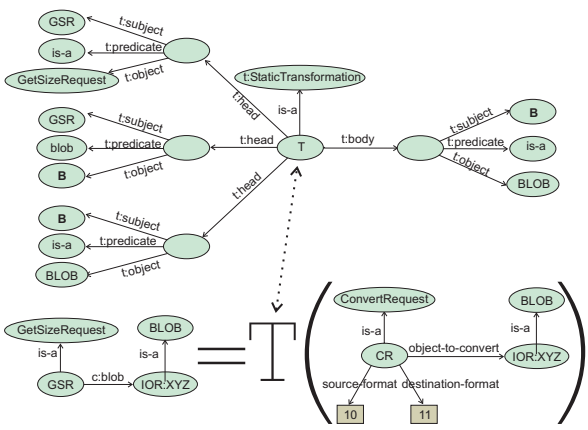

Fig. 5. Transformation T of the CORBA conversion request into a `getSize()` invocation

Although rule-based descriptions provide a powerful instrument for specifying static transformations, many useful transformations cannot not be easily expressed. For example, consider removing certain facts from a message. In this case, however, we can obtain the result as a set difference $m - T(m)$, where T selects the records to be removed from the message m. Therefore, we generalize static transformations as functions $F(m_1, \ldots, m_n)$ applied to sets of facts m_i. Many frequently used n-ary functions can be expressed as a composition of Datalog-based unary functions like the ones presented above and the conventional binary set operators $\cap$, $\cup$, and $-$. Together, the rule-based unary functions and the set operators form an expressive algebra for static transformations of logical descriptions.

In the next section we describe a language that can be used in our running example for describing the dynamic behavior of the mediator. We show how we represent the expressions in this language using the same RDF-based meta-model, and how we combine it with the static transformations to produce complete mediator descriptions.

6 Dynamic Transformations

In our example, after receiving a CORBA conversion request, the mediator has to perform a callback `doc.getSize()` asking for the size of the binary object and then retrieve the object content via one or more invocations of `doc.getBytes()` before forwarding it to the HTTP-based service. These interactions constitute a part of the dynamic transformation realized by the mediator.

A mediator compensating the discrepancies in behavior between these two interfaces can be modeled in different ways. For our scenario, it is convenient to use a simple finite-state machine with additional memory. Thus, we can use

a mediator that has a number of internal states and moves from one state to another until a final state is reached. The state transitions are triggered by external events like sending and receiving a message and may cause actions, e.g. storing the received message in a memory cell. Fig. 6 depicts a state machine that carries out the dynamic transformation between the CORBA client interface and the HTTP server interface. Assume that the mediator communicates with these components via channel `c` and channel `h`, respectively. In the notation we use, `?c ConvertRequest` means that a message of type `ConvertRequest` is expected via channel `c` whereas `!h ConvFile` denotes sending a message of type `ConvFile` over channel `h`.

Upon receiving a conversion request from the CORBA client (1-2), the mediator retrieves the size of the binary object to be converted (2-3-4). Knowing the size, the mediator fetches the binary content of the object[1] (4-5-6). In state 6, the mediator has obtained all information that is necessary to send a conversion request to the HTTP server. After receiving the content of the converted object (7-8), the mediator informs the CORBA client that the conversion has been completed (8-9). In state 9, the mediator gives the client an opportunity to inquire about the size (9-10-9) and the content (9-11-9) of the converted object. The mediation protocol terminates in state 12.

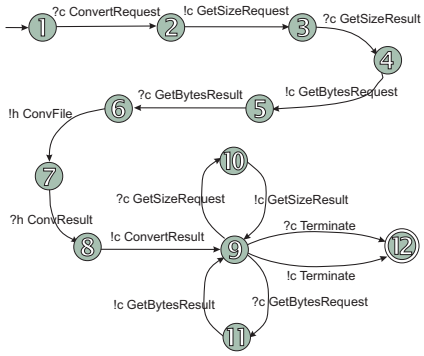

Fig. 6. Finite-state automaton of the mediator

Note that in our model a message is not atomic but consists of a set of facts. Every such message can be characterized by a 'classificator' function delivering `true` or `false` depending on whether the message is of a certain type. The state automaton needs such functions to recognize the types of the received messages and to trigger the appropriate transitions. Both classificator functions and manipulations on the content of the messages can be realized using static transformations discussed in the previous section. We implement classificator functions as rule-based transformations that produce some substitutions for `true` and deliver the empty set for `false`. For example, a simple classificator for the CORBA conversion request can be specified as follows:

```
(X is-a               ConvertRequest) &
(X source-format      V1)             &
(X destination-format V2)             &
(X object-to-convert  B)              &
(B is-a               BLOB)
```

[1] This operation could be split in multiple requests to retrieve very large objects if needed.

The format parameters contained in the initial CORBA request (transition 1-2) have to be stored until the binary content of the object is retrieved (2-3-4-5-6), and merged with the content in a combined HTTP request sent in (6-7). The actions used in our example are storing a message in a memory cell, reading from a cell, and arbitrary static transformations of the messages.

7 Interface Descriptions

The interface definition languages used in today's distributed object systems (e.g. CORBA IDL) are remarkably limited. For instance, given a file interface definition in CORBA IDL that has the methods `open` and `write`, it is not possible to specify that `write` can be called only after `open`. Or take our converter service as an example. The service comprises two interfaces, and it is not possible to derive valid interaction patterns directly from those interfaces. Rich interface descriptions and the ability to discover and compare interfaces of the components are, however, essential in a mediation environment with a variety of complex information processing services.

Similarly to the mediator specification, interface descriptions of the canonical wrappers for the CORBA client and HTTP service can be captured using finite-state automata (see Fig. 7). Having interface descriptions of the wrapped components has a number of notable advantages. First, they can be used to support the wrapper designer by generating wrapper skeletons automatically. The wrapper implementor solely has to provide the native code to perform component-specific actions for every transition. The wrapper skeleton can also enforce the correctness of the protocol. Secondly, for any pair of interface descriptions is it possible to determine algorithmically whether they are compatible [19]. Analogously, we can find all available mediator specifications that are capable of translating between the given server interface and other client interfaces. Finally, mediator specifications can be simplified by including by reference portions of the interface descriptions of the components. Evident candidates for that are classificator functions that determine the types of the messages.

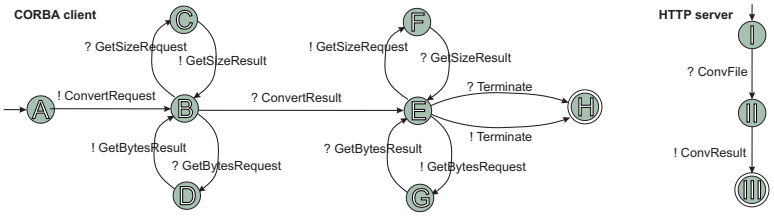

Fig. 7. Interface descriptions of the CORBA conversion client and the HTTP-based conversion service

In distributed object environment, there can be further immediate benefits of having rich interface descriptions of services. For example, the CORBA conversion service returns a reference to a `BLOB` object that contains the converted

file. For the server it is not possible to determine when it is safe to release the converted object without cooperation from the client (e.g. using special protocols for asynchronous garbage collection). However, if the server has an interface description for the client like the one presented in the top part of Fig. 7, it can dispose the converted object whenever the client reaches the state H.

8 Challenges of Integrating Distributed Services

The idea of using declarative specifications for mediators that integrate information processing services seems promising since it makes developing mediators an engineering task. However, it also presents a number of challenges.

One of the biggest challenges is harnessing the complexity of declarative specifications. Even for the simple scenario presented in this paper the complete specification of the mediator comprises about than 350 statements (i.e. contains as many arcs in the meta-model representation). Additionally, is uses interfaces descriptions of the wrapped components that contain about 270 and 190 statements for the CORBA and HTTP wrapper, respectively (they are so verbose because the structure of every message needs to be completely specified). Fortunately, this is merely the internally used encoding of the mediators that is not intended to be manipulated by human engineers directly. Ideally, every formalism used for mediation will have specialized graphical editor that allows to 'click' mediators together using graphical representations similar to that in Fig. 6. The rules used for static transformations can be input in their textual notation and translated into the meta-model representation automatically. To handle the complexity of descriptions, support for mediator composition is required. Supporting composition is non-trivial, since multiple formalisms may be used throughout the mediator. For example, a complex mediator may be specified as a finite-state machine on the high-level and may invoke Petri nets for executing subtasks that require concurrency control.

Another concern is the efficiency of mediation. For simplicity, we implemented the sample scenario described in the paper using a set of interpreters for the finite-state machines, Datalog-rules etc. (more implementation details are described in the next section). In turned out that runtime interpretation of state machine transitions and execution of the rules has a noticeable performance impact as compared to a hard-coded Java implementation. To deploy declarative mediators in realistic environments, more work on optimization is needed. On-demand compilation of mediators into bytecode may be a viable strategy.

Furthermore, we experienced a number of difficulties specific to the distributed object environments. Examples include adequate handling of streams or large binary objects, dealing with exceptions and mapping of complex nested parameters used in IDL specifications to our declarative descriptions. For instance, the complexity of mediator descriptions rapidly increases if all exceptions need to be represented. We found that it would be beneficial to use a representation where the exception handling does not belong to the core interface.

Moreover, to enable a closer integration with the existing distributed object systems the rich interface descriptions should reuse and extend the existing IDL specifications. For this, a meta-model representation for IDL is needed. Finally, a mediation environment of a realistic scale needs to address the problem of management of declarative specifications in database systems. The specifications need to be stored, queried, and analyzed. Powerful manipulations of complex declarative specifications will require use of advanced database techniques.

9 Architecture of the Prototype

In this section we briefly describe a prototype implementation of the elements of the mediation infrastructure that have been introduced in our running example. In our implementation declarative specifications of mediators are interpreted at runtime. The complete runtime environment for our sample scenario is presented in Fig. 8. The native components, i.e. the CORBA client and the CGI script, are presented on the bottom of the figure. The canonical wrappers are depicted above the native components. Both wrappers consist of two generic modules and one component-specific module. The component-specific module accesses the native API of the component. The generic modules are a finite-state machine (FSM) language interpreter and an inference engine that supports extended Datalog descriptions. These interpreters process interface descriptions of the components that are encoded in RDF and stored on the Web. Both interface descriptions contain exactly the FSM specifications presented in Fig. 7 plus the classificator functions of the messages.

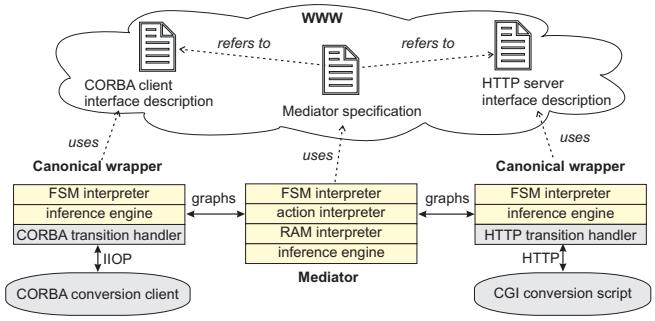

Fig. 8. Runtime environment for the sample scenario

Similarly to the wrappers, the mediator consists of four generic interpreter modules. In addition to the FSM and the inference engine, the mediator contains an 'action interpreter' for built-in and custom actions that can be incorporated into state machines. A random memory access (RAM) interpreter processes the descriptions of store and read memory operations. We use a generic executable module to launch the mediator and the wrappers. The parameters of this module are a reference to a declarative specification and a list of interpreters and native

components. This module loads the declarative specification and tries to find suitable interpreters that can process it.

10 Related Work

Integration of heterogeneous systems raised a variety of questions both in data-intensive and interaction-intensive application domains. Diversity of data-intensive systems inspired a number of projects focused on the integration of heterogeneous data sources. Examples include TSIMMIS [4], Information Manifold [9], Garlic [15], InfoSleuth [3], Infomaster [5], and OBSERVER [11], just to name a few. In essence, integration of data sources deals with translation and routing of queries and reconciling heterogeneous data structures returned by the sources. Some attempts has been made to cover database operations like updates [6].

Recently, incorporating declarative capability descriptions into query evaluation [9] has gained interest reaching beyond the scope of generation of executable query plans. For example, work presented in [20] examines computing capabilities of mediators based on the capabilities of sources they integrate. On the other hand, interaction-intensive applications urged research targeted at bridging the differences between applications that have functionally compatible but protocol-incompatible interfaces [19, 1]. The difficulties that arise upon tackling both data and protocol integration became tangible during building of the Stanford InfoBus [12].

In approaching this combined integration problem we tried to select the tools developed for data-intensive and interaction-intensive applications and to adjust them to our needs. In particular, data integration was addressed in-depth in MedMaker/MSL [14] and YATL [2]. Both approaches use nested data structures, Skolem functions, path expressions and powerful restructuring primitives that are desirable in the general case of data integration but may be an overkill for manipulation of logical descriptions of requests and replies of information processing services. Our decision to use logical languages for both capturing the mediation and representing the information flow between heterogeneous components was motivated by the work like [14, 7, 16].

Conclusion

We take a step towards integration of heterogeneous information processing services by developing a framework in which mediators are specified in a declarative fashion. To make declarative mediation feasible, we introduced canonical wrappers that provide logical abstractions of the underlying components. Furthermore, we examined how different formalisms can be deployed for mediator specifications and how the static and dynamic transformations performed by the mediators can be formalized. We demonstrated how our framework can be applied to addressing some of the mediation problems in distributed object systems, and discussed the benefits and challenges of our approach.

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.
2. S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your Mediators Need Data Conversion! In *ACM SIGMOD Int. Conf.*, pages 177–188, 1998.
3. R. Bayardo et al. InfoSleuth: Semantic Integration of Information in Open and Dynamic Environments. In *Proc. ACM SIGMOD Conf.*, pages 195–206, Tucson, Arizona, 1997.
4. H. García-Molina et at. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Inf. Systems 8:2*, pages 117–132, 1997.
5. M. R. Genesereth, A. M. Keller, and O. M. Dushka. Infomaster: An Information Integration System. In *In Proc. ACM SIGMOD Conference*, Tucson, 1997.
6. T. Härder, G. Sauter, and J. Thomas. The Intrinsic Problems of Structural Heterogeneity and an Approach to their Solution. *The VLDB Journal 8:1*, 1999.
7. V. Kashyap and A. Sheth. Semantic Heterogeneity in Global Information Systems: The Role of Metadata, Context and Ontologies. *In M. Papazoglou and G. Schlageter (Eds.), Boston: Kluwer Acad. Press*, 1997.
8. O. Lassila and R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. http://www.w3.org/TR/REC-rdf-syntax/, 1998.
9. A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of the 22nd VLDB Conference*, pages 251–262, Bombay, India, September 1996.
10. S. Melnik, H. Garcia-Molina, and A. Paepcke. A Mediation Infrastructure for Digital Library Services. In *Proc. ACM Digital Libraries 2000*, June 2000.
11. E. Mena, A. Illarramendi, V. Kashyap, and A. Sheth. OBSERVER: An Approach for Query Processing in Global Information Systems based on Interoperation across Pre-existing Ontologies. *Distributed and Parallel Databases Journal*, 1999.
12. A. Paepcke, M. Baldonado, C. Chang, S. Cousins, and H. Garcia-Molina. Using Distributed Objects to Build the Stanford Digital Library Infobus. *IEEE Computer*, February 1999.
13. A. Paepcke, K. Chang, H. García-Molina, and T. Winograd. Interoperability for Digital Libraries Worldwide. *Communications of the ACM*, 41(4):33–43, April 1998.
14. Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proc. of the 22nd VLDB Conference*, pages 413–424, Bombay, India, September 1996.
15. M. T. Roth and P. M. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *Proc. 23rd VLDB Conf.*, Athens, Greece, 1997.
16. K. Shah and A. Sheth. Logical Information Modeling of Web-accessible Heterogeneous Digital Assets. In *Proc. of the Forum on Research and Technology Advances in Digital Libraries (ADL'98)*, Santa Barbara, CA, 1998.
17. G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer 25:38-49*, 1992.
18. G. Wiederhold and M. Genesereth. The Conceptual Basis for Mediation Services. *IEEE Expert*, 12(5):38–47, 1997.
19. D. M. Yellin and R. E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, Mar 1997.
20. R. Yerneni, Ch. Li, H. Garcia-Molina, and J. D. Ullman. Computing Capabilities of Mediators. In *Proc. of ACM SIGMOD*, pages 443–454, Philadelphia, PA, 1999.