A Mediation Infrastructure for Digital Library Services

Sergey Melnik, Hector Garcia-Molina, Andreas Paepcke

Stanford University
Stanford CA 94305 USA
E-mail: {melnik,hector,paepcke}@db.stanford.edu

ABSTRACT

Digital library mediators allow interoperation between diverse information services. In this paper we describe a flexible and dynamic mediator infrastructure that allows mediators to be composed from a set of modules ("blades"). Each module implements a particular mediation function, such as protocol translation, query translation, or result merging. All the information used by the mediator, including the mediator logic itself, is represented by an RDF graph. We illustrate our approach using a mediation scenario involving a Dienst and a Z39.50 server, and we discuss the potential advantages and weaknesses of our framework.

KEYWORDS: mediator, wrapper, interoperability, component design

1 INTRODUCTION

Heterogeneity is one of the main challenges faced by digital libraries. Too often documents are stored in different formats, collections are searched with disparate query languages, search services are accessed with incompatible protocols, intellectual property protection and access schemes are diverse, and retrieved information is returned using dissimilar representations and ranked in inconsistent ways. Given that information sharing is of vital importance, there has been significant work on interoperable digital libraries in recent years, trying to bridge the gap between different information representations and systems [22].

Important advances have been made, specifically in developing *mediators* that can access information from multiple sources. A mediator typically receives a request (e.g., a query), submits a translated version of the request to several digital libraries, collects and merges the responses, and presents them to the user. However, today's mediators still have some important shortcomings:

- Current mediators are often hard to extend beyond the initial set of services they were designed for.
- It is difficult to incorporate into a mediator components that were developed elsewhere. For example, once a particular query translation algorithm has been implemented in a mediator, it is very hard to replace it by some other query translation package.
- Most often mediators do not tackle protocol differences. For instance, many mediators assume that all their targets communicate via HTTP.
- Usually it is not easy to extend a mediator to non-search tasks. For example, if a mediator is designed to query multiple search engines, it is hard to make it mediate among different payment mechanisms or among different document summarization services.

In this paper we propose a mediation framework that addresses these shortcomings. The framework presents a very flexible environment where different components (that we will call "blades") can be combined to address a specific mediation task. One of the components, in particular, will be responsible for translating protocols. For example, this component may receive a single synchronous message from a user, and in turn issue a sequence of asynchronous messages to perform the requested task.

Our proposed framework does *not* contain any new revolutionary ideas. Rather, we have taken a number of existing ideas, from component based software engineering, extensible database systems, programming languages, operating system kernels, and so on, and combined them in a way that we believe is especially well suited for a digital library environment. Also, we stress in advance that we are *not* claiming to completely solve heterogeneity problems. Our framework does offer substantial flexibility, but it may introduce significant performance overhead and additional complexity. And it is not clear to us yet if our approach scales well to scenarios with large numbers of services and many different mediation tasks. (Then again, we do not know of any other mediation approach that scales very well.) Nevertheless, our initial experience using this framework (in low complexity scenarios) indicates that the framework is good when it is important to quickly adapt to new services and information models, or when it is important to experiment with different mediation components and algorithms.

We start in Section 2 by briefly illustrating a typical mediation scenario. Then in Section 3 we present our proposed reconfigurable and extensible mediation framework. In Section 4 we discuss potential drawbacks of our approach, while in Section 5 we comment on the current status of an implemented prototype. In Section 6 we review related work.

2 CASE STUDY

In this section we illustrate the principles and components of a mediation infrastructure for digital libraries. We illustrate with a scenario, depicted in Figure 1, based on retrieval services. The client on the right of the figure issues queries via SDLIP [27], a protocol developed as part of the InterLib Project. This protocol, as well as the others used in our scenario, are simply examples to illustrate the diversity available in digital libraries. One of the servers on the left provides access to an NCSTRL [16] document collection using the Dienst protocol, and the other server implements the Z39.50 protocol.

The SDLIP client encodes queries in XML according to the DASL specification, and waits for asynchronous delivery of portions of the search result. The NCSTRL server deploys a stateless request-reply model and accepts URL-encoded queries. The Z39.50 server expects a query in Reverse Polish Notation encoded according to ASN.1. For a given query, the Z39.50 server returns the number of search results and allows the client to retrieve subsets of the result in separate requests. The client and the servers are the *native components* in our mediation scenario.

The wrappers shown in Figure 1 hide some of the heterogeneity of the native components. In particular, the wrappers communicate with the native components via native protocols, i.e. SDLIP, Dienst and Z39.50. In turn, the wrappers provide a relatively homogeneous message-passing environment for the mediator in the center of the figure.

The mediator performs dynamic brokering by taking requests from the client, translating those requests into requests to the servers, combining the information received from the servers, and passing it along to the client. Even though the mediator is shielded from the native components by the wrappers, it still has to deal with the semantic interoperability between incompatible representations of information provided by the wrappers. In our scenario, the major problems include:

- Protocol translation: a stateless protocol used by the NCSTRL server and a stateful protocol deployed by the Z39.50 server have to be mapped onto yet another stateful protocol used by the SDLIP client.
- Query translation: an SDLIP/DASL query has to be translated into an NCSTRL and a Z39.50 query.
- Data translation: the search results delivered by the servers need to be merged and translated into a representation that the client understands. In our example, the NCSTRL server returns a set of bibliographic records based on the BIB attribute model, Z39.50 deploys MARC, and the client expects Dublin Core records.

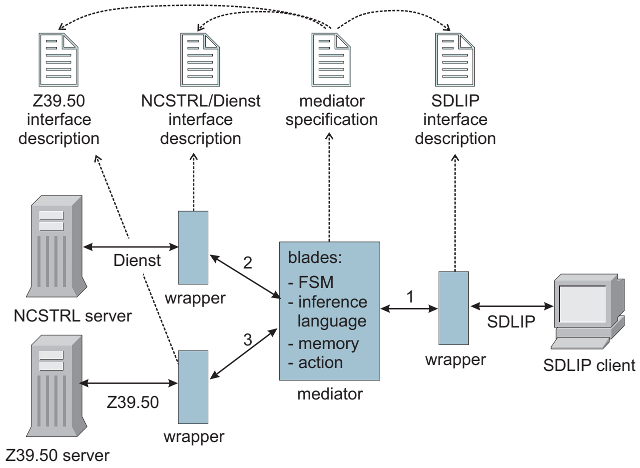

Figure 1: Mediation architecture that shows an SDLIP client accessing NCSTRL and Z39.50 servers

Protocol translation

Assume that the mediator takes the query submitted by the SDLIP client, translates it into an NCSTRL query, sends it to the NCSTRL server, retrieves the results that are encoded as BIB, translates them into Dublin Core, and asynchronously sends the results to the client. Then, it sends a translated Z39.50 query to the Z39.50 server which first returns the size of the result set. After that, the mediator fetches the result in chunks of say 50 records, does its best to convert them from MARC to Dublin Core, and forwards the chunks to the client.

A formal representation of the algorithm described above is depicted in Figure 2 as a finite-state automaton. The labels of the arcs specify events and actions performed by the automaton on a transition from one state to another. The notation `q ← 1?SDLIPquery` specifies that a message of type "SDLIP query" arriving on channel 1 is stored in a memory cell `q`. The channels 1, 2, and 3 correspond to the SDLIP client, NCSTRL and Z39.50 server, respectively (these are the labels used in Figure 1). The bang "!" denotes sending a message. A pseudo code expression like `q1=SDLIPQ2NCSTRL(q)` denotes that the application of function `SDLIPQ2NCSTRL`

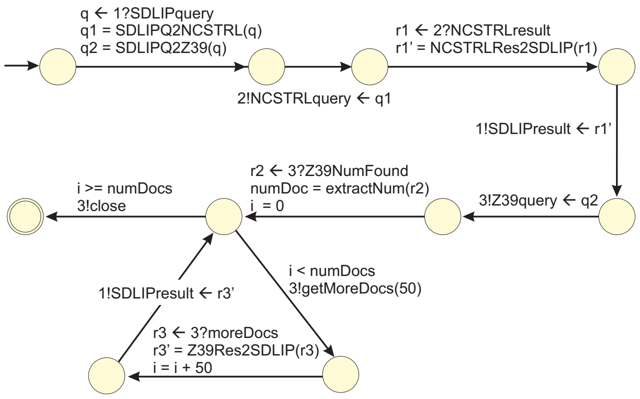

Figure 2: A finite-state machine representation of the mediator

on the message `q` yields the result `q1`.

We do not describe the rest of Figure 2 because our goal is not to fully define this finite-state automaton formalism. We are not even arguing that this formalism is the best for describing protocol translations. In our particular running example, we believe that this formalism is a convenient way to represent the work that must be done by the mediator, but perhaps there are other mechanisms (e.g., Petri nets, Java programs) that are better suited for other scenarios or for other implementers. The other formalisms that we introduce below (RDF, query translation rules, Datalog) are again only examples of the diverse types of machineries that are available to describe mediation tasks.

Query translation

Assume that the SDLIP client in our scenario submits a search request for music recordings authored by Lou Bega. The original query represented in XML according to the DASL specification is shown in the top left part of Figure 3. The wrapper of the SDLIP client receives the search request containing the query via the native SDLIP interface. Then, the wrapper converts the request into a logical representation. The top right part of the figure shows the logical structure of the native SDLIP query that is represented as an RDF model. RDF (Resource Description Framework [15]) is a metadata standard recommended by the W3 Consortium for describing objects and their relationships. The choice of the particular RDF representation for the query is made by the designer of the SDLIP wrapper.

In RDF, all information is represented by nodes (resources) in a graph. Each node may have a value, and labeled properties that link it to other nodes. For instance, consider the RDF representation of the SDLIP query in Figure 3. The leftmost node labeled θ is the root of the query structure. The `type` property of θ tells us this is a "Basic Search Query," while the `where` property links to the Boolean condition that selects documents. The rest of the nodes specify the components of this search condition.

The mediator receives the RDF model representing the query and translates it into the RDF representations suitable for the wrappers of the Z39.50 and NCSTRL servers. The server wrappers in turn transform the logical representations of the queries into a URL-encoded query and the Z39.50 syntax for the NCSTRL and Z39.50 server, respectively. Both native encodings and the corresponding RDF models of the target queries are depicted in the bottom part of the figure.

To translate the queries, the mediator calls a query translation component. In Figure 2, these calls correspond to `q1=SDLIPQ2NCSTRL(q)` and `q2=SDLIPQ2Z39(q)`. The task of the query translation component is to transform the RDF-based representation of the query chosen by the SDLIP wrapper into the representations used by the NCSTRL and Z39.50 wrappers. The query translation must take into account the capabilities of the servers, and the supported attributes and attribute models of the servers. In our example, the servers have only one searchable field for author. Thus, the first name and the last name submitted by the SDLIP client have to be concatenated and formatted in a proper way in the translated queries. This step is referred to as predicate rewriting.

In general, predicate rewriting is a very hard problem, and full solutions do not exist yet. However, rule based systems are often useful for performing the most common rewritings. Here we illustrate with two rules that may be useful in our scenario; please refer to [7, 8] for a more complete discussion of query translation.

```
[author = $1+" "+$2]  <=  [FirstName = $1] and
                          [LastName  = $2]

[1003 contains $2+", "+$1] <= [FirstName = $1]
                          and [LastName  = $2]
```

The first rule translates DASL-encoded search predicates to NCSTRL. If the DASL query specifies a conjunctive search condition on the `FirstName` and the `LastName` attributes, the values of these attributes, e.g. "Lou" and "Bega", are bound to the variables $1 and $2, respectively. Then the left hand side of the rule is triggered, yielding a corresponding condition for the NCSTRL query. This new condition searches for the "author" whose full name is the concatenation of the $1 and $2 variables. The second rule rewrites the predicates for the Z39.50 server in an analogous way.

To apply these rewriting rules, the SDLIP query graph shown in Figure 3 is first converted into a generic query tree. The search predicates are the leaves of this tree, whereas the internal nodes represent Boolean operators

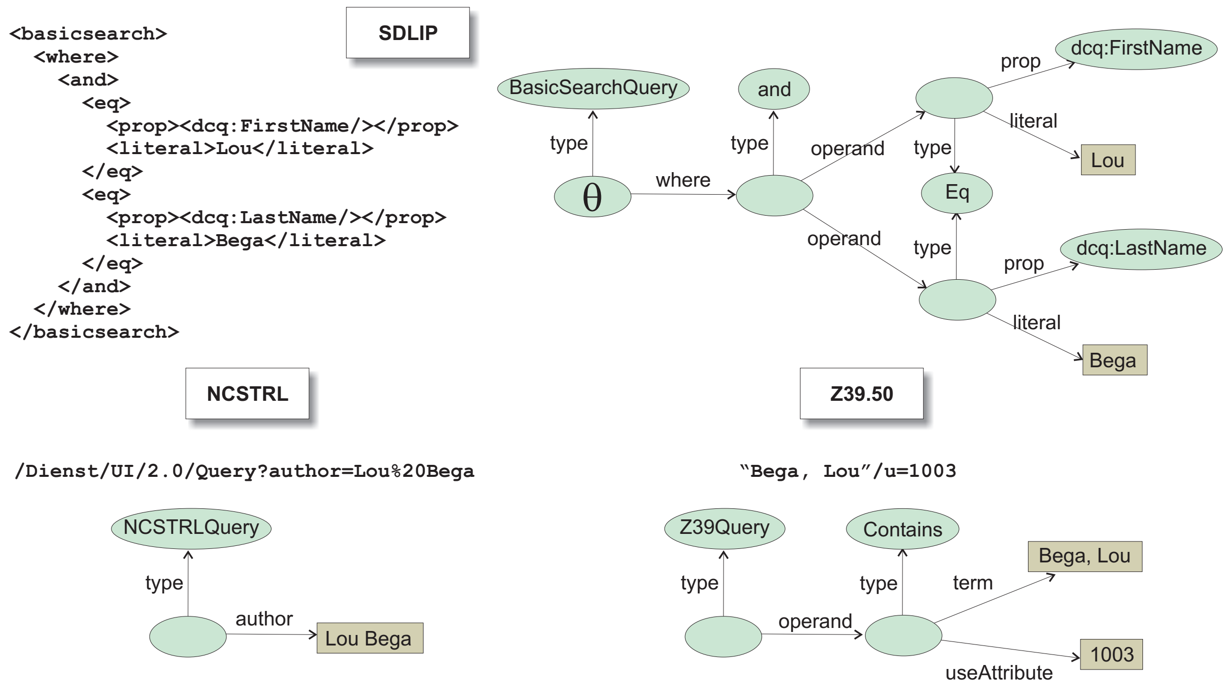

Figure 3: Native queries for SDLIP, NCSTRL and Z39.50 and their logical models

AND, OR, and NOT. After the application of the rewriting rules, the generic query tree is mapped onto representations of query graphs chosen by the NCSTRL and Z39.50 wrappers.

Data translation

The next problem that the mediator has to address is that the wrappers deliver search results using different representations. Thus, the mediator needs to convert semantically incompatible data instances returned by the search service wrappers into a format supported by the client wrapper. In our scenario, both servers return sets of bibliographic records. The mapping between attribute models such as BIB, MARC and Dublin Core can be specified, for example, using a set of Datalog rules. To illustrate, a mapping of MARC attributes 720 and 200 to their Dublin Core equivalents is shown below. Uppercase letters denote variables.

```
X dc:Creator C  <=  X marc:720 C
X dc:Title   T  <=  X marc:200 T
```

The rules transform the graph returned as a search result into another graph. For example, the first rule specifies that for every two nodes X and C connected by an arc labeled `marc:720` in the source graph, an arc `dc:Creator` will be placed between X and C in the destination graph. A mapping between different classification schemas used by the servers and the client can be expressed using a similar approach. For example, if the Z39.50 server uses the Library of Congress Subject Headings (LCSH), the translation to the Dewey Decimal Classification (DDC) could be implemented using one rule and a set of facts:[1]

```
                    X marc:650   L  &
X dc:Subject D  <=  F m:lcsh     L  &
                    F m:ddc      D

f1 m:ddc  '784'
f1 m:lcsh 'Guitar choir music'

f2 m:ddc  '784.1888'
f2 m:lcsh 'Choros'
f2 m:lcsh 'Mambos'
```

Applied to the MARC record shown on the top of Figure 4, the above set of rules produces the Dublin Core record depicted on the bottom of the figure[2].

3 PROPOSED SOLUTION

Suppose that we implement the mediator for the sample scenario of Section 2 by writing a large program in our favorite language. This program would implement the logic described by the finite-state automaton of Figure 2. If we did not want to hard code the query translation rules, perhaps we could develop a data structure to represent the rules we need, and code our own interpreter for these rules. For data translation, we could either hard code the transformation, or again develop some

[1] For the curious reader: Choro is an instrumental musical form characterized by improvisation and virtuosity that originated in Rio de Janeiro in the 1870's.

[2] ASIN: a number used by Amazon.com to catalog anything that is not a book.

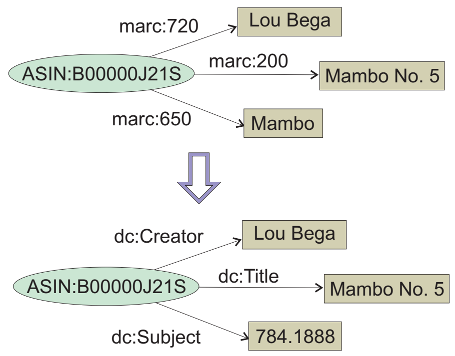

Figure 4: Result of conversion of a bibliographic record

more general rule execution machinery. This is actually the way many mediators are written today.

Unfortunately, this approach is not very flexible. For example, say we change our mind and wish to query the Z39.50 server first, because it has faster response times. This would require rewriting parts of the mediator and recompiling it. Say we needed to add a new search server in our architecture, or say we wanted to execute server queries in parallel instead of serially. Again, this would require a major effort.

Similar issues arise if we think of query or data translation. Suppose someone has developed a nice, generic query translation package. Using it would probably be hard, since it uses a different rule and query representation than the one we selected. Even if we implemented a rule interpreter for query translation, it may be a lot of work to maintain it as the attribute models of servers evolve.

Finally, search and retrieval is just one aspect of digital libraries. We eventually want to extend our mediator to cover other services such as conversion, annotation, and citation extraction. It will be difficult to add new mediation functionality to our large, monolithic program.

The solution we propose in this paper is a *modular, plug and play mediator*. With this mediator, the mediation tasks are well defined, and can be executed by independent, replaceable modules. For example, the predicate rewriting task is performed by a query translation module. The mediator can be configured with one or more translation modules, that can be called as necessary. We will call these plug and play modules *blades* since they are analogous to database system blades (in turn analogous to shaving blades used to "configure" a razor). Each blade can be coded and specified in the language that is best suited for the task at hand, representing information in the most convenient way. We use the term "plug and play" blades, because as we will see, the mediator will detect and configure a new blade in much the same way a plug and play operating system automatically detects and configures new hardware.

In our approach we take modularity to the extreme and make the mediator logic itself (illustrated in Figure 2) a blade. This means that to change the logic of the mediator (e.g., to go from serial to parallel query execution), we can simply replace the "main" mediator blade by one that has a new finite-state automaton. Furthermore, if we do not find finite-state automaton a convenient formalism, we can insert a different type of blade, that represents its logic in a different way. For example, we could use Petri nets to represent protocol dynamics. We call the main blade that drives mediator execution the *dynamic blade* since it captures the dynamic nature of the mediator. Pushing our analogy with plug and play hardware a bit, replacing the dynamic blade of a mediator would be equivalent to changing the operating system of a computer, say from Windows 98 to Linux.

Building a mediator from a set of reconfigurable blades seems like an attractive idea, but is it feasible? How will the mediator be able to represent all the different types of information, from queries, to rules, to automaton, to results? How will we coordinate the different blades, so that each one does what it is supposed to do at the right time? How will information be passed to and from blades? In the rest of this section we answer these questions, as we describe a plug and play mediator framework that has been implemented and provides the flexibility we advocate.

Mixing of specification languages

The challenge we face is how to describe mediator tasks and information using different formalisms which, nonetheless, can be plugged together in a seamless way. To achieve that, we need a *meta-language* that allows us to encode expressions in disparate languages, like Datalog and finite-state machines, in a uniform way. A further requirement is to be able to establish relationships across expressions stated in different languages. For example, it should be possible for a finite-state automaton description to point or link to a particular query conversion rule that needs to be invoked.

The meta-language that we use in our mediation infrastructure is RDF. To illustrate how different languages can be encoded in RDF, consider the first transition of the state automaton of Figure 2. The top part of Figure 5 presents the RDF encoding of that transition. The transition is centered on the node (RDF resource) labeled α. Node α has four properties (outgoing arcs): original state (`p:origState`), destination state (`p:destState`), `p:event` and `p:action`. (In addition it has a type property (`t`), indicating that α is indeed a transition node.) Notice that all properties and literals that are associated with a transition event have a pre-

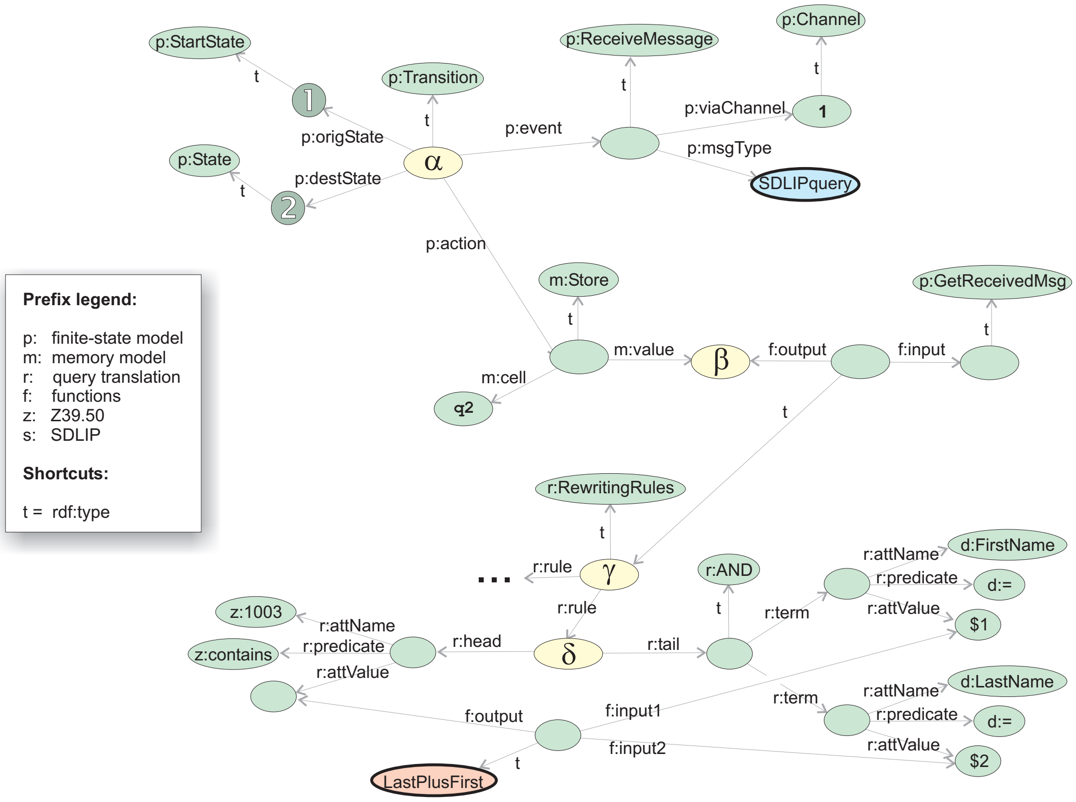

Figure 5: Mixing of declarative languages: a transition of the finite-state machine and a query rewriting rule are both encoded using RDF

fix `p:`. All these strings are part of the *vocabulary*, i.e. language elements, of the finite-state model.

The `p:event` property, for instance, points to a subgraph that describes the triggering event (the reception of a message of type `SDLIPquery` via Channel 1). Upon the transition, the automaton moves from the `p:origState` (labeled "1" in the figure for convenience) to the `p:destState` (labeled "2") and performs the `p:action` whose description is shown in the middle part of the figure. (For clarity, only one of the actions associated with this transition is shown in the figure.)

The central part of Figure 5, centered on node labeled β, represents the assignment statement for the action `q2=SDLIPQ2Z39(q)`, while the bottom part, centered at node γ, represents the set of rules that are used to implement the `SDLIPQ2Z39` query rewrite. Finally, the subgraph centered at node δ represents one of the translation rules (the rest are not shown). The conditions and variables that make up this rule are described by the `r:head` and `r:tail` subgraphs, although we do not discuss the details here.

Figure 5 illustrates how diverse formalisms like finite-state machines and predicate rewriting rules can be represented in a single meta-model. The figure also illustrates that it is easy to link the different formalisms, once they are all in RDF. For example, the connection between the transition description at node α (part of the finite automata formalism) and an assignment statement (part of an action execution model) is represented by the `p:action` link out of α.

Plug-and-Play configuration

Every formalism used in the mediator specification is processed by a separate blade. The challenge is to compose these blades so that they can dynamically interact with each other. To enable such composition, we introduce a special component, called the *blade manager*. The manager orchestrates calls to blades, acting as a mediator "kernel." For example, if the blade manager encounters the specification of the predicate rewriting rule `SDLIPQ2Z39` it makes sure that the appropriate query translation blade is called to evaluate the rule. Thus, the manager "binds" statements in different languages into the blade that can execute the statements.

The blade manager initially learns what resources are available through a *configuration* object, as suggested by Figure 6. Upon initialization of the mediator, the blade manager is given a URL pointing to the RDF-encoded mediator configuration. The configuration contains descriptions of the available *blade factories*. A blade factory is an executable module that generates

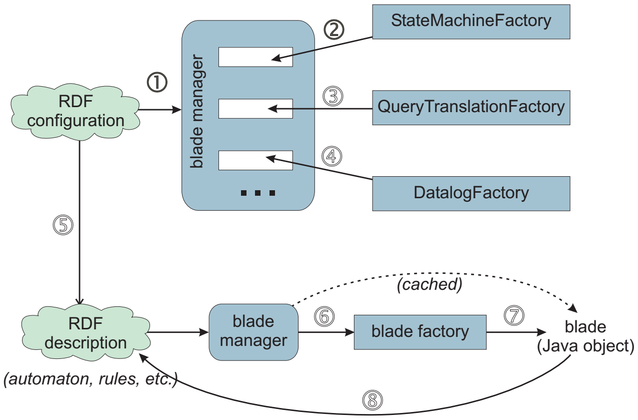

Figure 6: Execution of the mediator using the blade manager

specific blades needed for mediation. Factories are dynamically loaded into the mediator, and are called upon to generate blades, which are then also loaded and executed.

To illustrate this process, let us walk through the initialization and blade instantiation process, as illustrated in Figure 6. First, the configuration is read by the blade manager and the blade factories are registered (Steps 1-4). In our prototype, the factories are downloadable or locally installed Java modules. For example, the module `QueryTranslationFactory` generates `QueryTranslation` blades. The configuration also includes the RDF-encoded description of the actions to be taken by the mediator. In our example, this is the structure shown in Figure 5. We call this complete structure the *model*.

Next, the blade manager needs to instantiate a dynamic blade to execute the main logic of the mediator (Step 5). Initially, the manager does not know which of its installed factories can generate such a blade, so it questions each registered factory. Every factory offers a default interface for such questions, called the `BladeFactory` interface:

```
public interface BladeFactory {
  public Blade getBlade(Resource id,
                        Model m);
}
```

Parameter `m` is the model in use, while `id` identifies the particular subgraph within the model that the required blade will interpret. In our case, `id` identifies the portion of the model that captures the mediator logic (e.g., that includes the transitions like α shown in Figure 6). When the `getBlade` method of `QueryTranslationFactory` is called, the factory does not recognize the `id` structure as something it operates on, so it returns a null factory. Eventually a `DynamicBladeFactory` is called (Step 6), and it does recognize parameter `id`. In our scenario, `id` is recognized by the factory that understands finite-state automata descriptions. This factory generates and returns a dynamic blade `StateMachine` to the blade manager (Step 7). When the blade is generated, it is "configured" to execute the logic description `id`. For instance, in our example, the factory may parse the logic description `id`, and compile it into "code" that will actually be executed by the `StateMachine`. This way the parsing and compilation work is done once. This strategy is more efficient than having the `StateMachine` interpret the RDF description each time it is invoked to perform some action.

Once the dynamic blade `StateMachine` is installed by the blade manager, the mediation process can start up. Dynamic blades offer several methods for this start up. For example, the blade manager can invoke the `setChannel` method of `StateMachine` to bind the communication channels used by the dynamic blade (e.g., Channels 1 and 2 in our example) to actual TCP/IP connections for the client and server wrappers. (The blade manager gets the TCP/IP connections to use from the configuration.)

After startup, dynamic blade `StateMachine` waits for incoming requests in its start State 1. Once a request arrives via Channel 1 (see Figure 2), the dynamic blade tries to match the request reception event against the events of transitions that are defined for the State 1. When a message arrives and the blade verifies that it is of the appropriate type, the blade decides to execute the statement `q1=SDLIPQ2NCSTRL(q)`. However, the dynamic blade does not actually execute the statements. Instead, it asks the blade manager (step 8 in Figure 6) to locate the appropriate blade for this task. The blade manager contacts each available factory (using the `BladeFactory` interface), asking it to examine the statement in question. (Parameter `id` of `getBlade` identifies the statement.) The query translation factory `QueryTranslationFactory` determines that resource `id` involves a SDLIP to NCSTRL translation, which it recognizes. Thus, the factory generates and returns the appropriate blade. Again, the factory may decide to compile the statement, so its blade can operate more efficiently.

When the blade manager receives the query translation blade, the manager records in a cache the parameters that generated the blade (`id` and `m`), plus the identity of the blade. In general, before the manager invokes method `getBlade` to generate a needed blade, it checks its cache to see if the blade already exists. If the blade exists, then the search for a matching factory can be skipped altogether.

After the query translation blade has been generated, the manager calls it to perform the needed translation. All blades, except for dynamic ones, offer the same generic interface:

```
interface StaticTransformation extends Blade {
  Model compute(EvaluationContext ctx);
}
```

The single parameter ctx is an *evaluation context*. Such a context is an object that includes everything that is necessary to perform the blade's service. For our example, ctx allows the query translation blade to obtain the query to be rewritten. The context ctx for a blade is initially created by the blade manager, and is reused for all calls to the same blade. Thus, some blades actually store data needed in subsequent calls in their context.

When the blade manager invokes the compute interface, the query translation blade extracts the operands from ctx and transforms the query. The translated query, represented as an RDF graph (of type Model), is returned to the blade manager, which in turn returns it to the dynamic blade (as the result, the desired statement is executed). After computing the second translated query, the dynamic blade completes its transition to State 2. Then, it examines the transitions from State 2 and sends the translated query stored in q1 over Channel 2 to the NCSTRL wrapper.

In our description of blades and how they are called, we have omitted many implementation details, focusing only on the main steps. For example, the messages received by the dynamic blade are verified in our prototype using Datalog rules. Furthermore, there is great flexibility as to exactly what each blade does. For instance, in our example, we said that the dynamic blade made one call to the manager for each query translation statement it wished to execute. However, we can just as easily design a dynamic blade that packages all the actions for a given transition into a single *block* of statements. The dynamic blade can then ask for a blade to execute the entire block as one unit. For this, we would need a blade that takes as input a block, interprets it, and makes the appropriate calls to execute each statement. With this scheme we can isolate the statement execution logic from the state transition logic, enhancing modularity.

To summarize, the mediator used in our scenario uses a single dynamic blade, i.e. a state machine interpreter, and a number of static blades, e.g., a Datalog interpreter or a query translation component. A blade can either execute code that was compiled by its factory, or can interpret RDF graphs that encode its actions. In our initial prototype, for instance, the dynamic blade actually interprets an RDF description of the state automaton (Figure 5), while our query translation blade executes rules represented in a compact notation (analogous to "intermediate code" generated by compilers). As we described, the blades for different formalisms are linked together by the blade manager.

The plug and play mediator we have described offers tremendous flexibility. The rules and logic descriptions used can be easily changed by modifying their RDF descriptions in the mediator configuration. Furthermore, the formalism or language used to describe the rules or the operational logic can be replaced, as long as new blades are available to work with the new formalism. Of course, the added flexibility of our approach does imply a higher performance overhead, as compared to the static, monolithic approach introduced earlier. This is a classic flexibility versus performance tradeoff. We believe that for many digital library mediation scenarios, where servers and clients are so varied and change so rapidly, a flexible mediator that can rapidly adapt is preferable to a faster one that works in limited cases.

Wrappers

A wrapper is conceptually similar to a mediator, except that it only needs to translate between one native component and one mediator. For example, in Figure 1, the Z39.50 wrapper is the gateway between the mediator and the Z39.50 server.

In our mediation infrastructure, we use the same machinery for wrappers as for mediators. The core of the wrapper is again a blade manager that loads suitable modules at runtime. For example, the configuration of the Z39.50 wrapper describes where to find the interface description of the wrapper, and specifies that the StateMachineFactory should be used. The dynamic blade StateMachine is defined using a programming language (Java, in our prototype), sine it has to deal with the intricacies of the native protocol.

Philosophically, we advocate that wrappers be as simple as possible. For example, our Z39.50 wrapper only deals with the syntactic conversion of the RDF-encoded query into a native string. The native Z39.50 protocol is not translated by the wrapper; its messages are simply encoded in the appropriate RDF. The actual query and protocol translation work is left to the mediator.

Keeping wrappers simple encourages their construction. Since wrappers deal with native components directly, it is best if they are developed by the people who run the native services and clients. The less work the wrapper needs to do, the more likely these developers will provide the needed wrappers. Once the information from a native component is "packaged" in a way that the mediator can understand, the mediator can take over and do the harder translation and mediation work.

4 CHALLENGES

The system described in the preceding sections promises a variety of advantages to the designer of mediators.

In particular, it suggests a graph based representation of client/server operations, such as query submissions. The idea is that transformations of these graphs take the place of computer programs. We have also introduced finite state machines to model the dynamics of protocol interactions, and a rule-based approach to query translation. These building blocks are the components which can be combined with occasional code segments to construct mediation blades.

The intention of this approach is to introduce a maximum of flexibility into the construction of mediation modules. Client/server protocols are made into first-class objects that can be constructed by a designer, and can then be transformed or executed by "interpreters". The intended result is the easy, interactive creation of mediators, and convenient reuse of mediation modules. What are some of the challenges of this approach?

One issue is the potential complexity of graph-based approaches. Computer programs are very efficient for representing operations and data structures. But graphs can be better at giving overviews, and at displaying connections among the programs' various parts. The drawback of graph-based approaches is that they are difficult to scale, because they become complex quickly. One challenge will be the construction of mediator development tools that balance the advantages of textual programs and their graphical counterparts. Mediator designers must be able to specify protocol interfaces easily, and without being overwhelmed by cognitive complexity. After all, the standard to measure against for ease of program creation, reuse, and maintenance is the traditional programming approach. We will have to prove that at least for the application of digital library interoperability, the proposed approach is superior.

A second challenge will be support for debugging. Rule-based systems tend to be very powerful, but their inherent parallelism and non-sequential nature can make them difficult to debug. In order to analyze which rules fire when, and how the system undergoes state changes, traditional debugging techniques must be adapted.

A third challenge will be scalability in performance. Interpreted systems are intrinsically slower than compiled approaches. However, many interpreted programming languages have relied on increased processor speeds to make their performance acceptable. The success of the proposed approach will depend in part on optimization techniques that help speed up execution. Programming language optimization techniques will be of help.

5 CURRENT IMPLEMENTATION

This section briefly describes some implementation details of our framework. We have used the framework to implement a mediator and wrappers for the scenario of Figure 1. The wrapper interfaces are represented as state machines with four, two and five states for the SDLIP, NCSTRL and Z39.50 wrapper, respectively. Currently, not all possible error conditions are considered. The actions of the state machines that describe wrapper interfaces are implemented as native Java modules. The following statistics describe the amount of code needed to "glue" the declarative automata descriptions with the native libraries used by the wrappers.

The SDLIP wrapper is based on the SDLIP toolkit developed in the InterLib project. The amount of native Java code used by the SDLIP wrapper is about 300 lines. The NCSTRL wrapper was implemented from scratch using an HTTP server library and contains about 250 lines of code. The implementation of the Z39.50 wrapper uses the Z39.50 libraries developed by OCLC. The code needed is about 900 lines of code in addition to the libraries.

The implementation of the mediator involves the blade manager, i.e. the "kernel" of the mediator, and the blade factories needed to execute the mediator specification. The mediator uses ten states as depicted in Figure 2. The blade manager implementation in Java comprises about 200 lines of code. The major blades used in the scenario are `StateMachine`, `Datalog`, and `QueryTranslation`. The `StateMachineFactory` creates a `StateMachine` blade that interprets the RDF specification of the automaton. The interpreter contains about 500 lines of code. The `DatalogFactory` is built on top of the Java-based inference engine SiLRI developed at Karlsruhe university. The implementation of this factory required about 550 lines of code. The `DatalogFactory` translates RDF-encoded rules into the native encoding of inference rules supported by the inference engine (F-logic). The blade generated by the Datalog factory passes translated rules to the inference engine. The inference engine is invoked within the same Java virtual machine, i.e. it does not run as a separate process.

Query translation is currently hard coded in Java. We are working on the implementation of the query rewriting blade factory that uses declaratively specified rules. At present, we do not have an implementation of our mediation scenario that uses a hard wired mediator. Without such reference implementation we cannot provide meaningful performance comparisons with more traditional approaches.

In addition to the scenario of Figure 1, we have implemented a prototype of a mediator for document format conversion services. The mediator is capable of mapping a relatively complex CORBA-based conversion interface to a simple CGI-based one. The state automaton of the CGI wrapper contains only two states (wait for request, request processed). In the CORBA-based interface, the client passes to the server the object handle for the document to convert together with the specification of the source and destination format. The server issues a callback to the client to determine the size of the object, and then incrementally fetches the pieces of the document content. After that, it converts the document, say from PostScript to PDF and delivers the handle of the converted object to the client. The client fetches the content of the converted object analogously. The CORBA-based conversion interface is described as a finite-state machine using eight states. The mediator automaton is based on ten states. The mediator deploys the same blades `StateMachine` and `Datalog` that are used in our retrieval scenario.

In our current implementation, the wrappers and mediators exchange RDF models over the network using the standard RDF serialization in XML. The low-level communication interface is implemented as full-duplex asynchronous exchange of serialized RDF models via TCP/IP connections. However, other low-level interfaces, e.g. a CORBA-based, are also conceivable. In order to support the building of wrappers, we provide a comprehensive RDF API for manipulating in-memory RDF models accompanied by a lightweight RDF parser/serializer.

6 RELATED WORK

The focus of our work is integration of *information processing services* in digital libraries. A wealth of prior work addressed interoperability of heterogeneous data sources. For example, the following "monolithic" systems support mediator-supported querying: TSIMMIS [11], Information Manifold [17], Garlic [25], SIMS [3], InfoMaster [12], and MOMIS [5]. The work presented in [13] considers a mediation approach for updateable data sources. Rich interactions between users and services are examined in [18]. Semantic-oriented approaches to integration are discussed in particular in [26, 4, 19].

Interoperability for digital library services was extensively addressed in the Stanford InfoBus [21]. In InfoBus, every service like search, attribute translation or metadata service is a distributed object implementing a well-known interface. In contrast, our framework focuses on a mediation-based approach. Furthermore, wrapper implementers do not have to agree on a set of common application-specific interfaces. The extensibility of the infrastructure is increased by relaxing the rigid low-level interfaces typical in distributed computing. A detailed comparison of approaches to interoperability for digital libraries can be found in [22].

The formalisms used in our mediation infrastructure for protocol, language and data translation (finite-state machines, Datalog, query rewriting) are only examples of mechanisms that are available for description of mediation tasks and have been studied in-depth elsewhere (cf. [2, 28] for protocol translation, [9, 23] for data conversion, [7, 8] for query rewriting).

Our architecture borrows from several areas of computer science. We can thus draw on a variety of traditions and experiences. Component based software engineering is one such tradition. Some commercial databases use data blades, analogous to our mediator blades. Experimental database designs have gone beyond these more limited blade facilities to introduce far reaching extensibility. They have opened the entire database to modification through plug-in modules. The Open OODB at Texas Instruments was one such project [6]. It enabled designers to change large-granularity modules in order to introduce new or optimized functionality.

The operating systems community has also experimented with component based approaches. Maybe the most famous of these is the Mach operating system [1, 24]. It offered a kernel, analogous to our blade manager, and allowed designers to insert their own virtual memory manager.

Programming language designers have explored a third component based approach. They use metaobject protocols that make system internals into first class objects [14]. By manipulating these objects, fundamental behaviors of the language can be changed.

Our approach is heavily based on interface description. CORBA based systems also emphasize the separation of interface and implementation [20]. While we rely on this separation, the finite state machine portion of our proposed approach was inspired by CORBA's inability to describe the dynamics of a client/server interaction.

Rule-based systems have been used extensively in expert system design. Specific languages, such as Prolog, are available for processing rules. We used the SiLRI inference engine [10] that supports a variant of Datalog with negation to implement our prototype rule system.

7 CONCLUSION

In this paper we have explored what it takes to build a very flexible and dynamic digital library mediator, that can be adapted as services and technologies evolve. We described how diverse models and information can be represented, and described an extensible execution environment where "blades" perform different mediation functions, orchestrated by a blade manager. We believe that our approach holds promise in environments where target services and mediation tasks are not fixed in advance, and where high performance is not the main objective.

REFERENCES

1. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.
2. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Software Engineering and Methodology*, 6(3):213–249, July 1997.
3. Y. Arens, C. A. Knoblock, and Ch.-N. Hsu. Query Processing in the SIMS Information Mediator. *Advanced Planning Technology, Austin Tate (Ed.), AAAI Press, Menlo Park, CA*, 1996.
4. R. Bayardo, W. Bohrer, R. Brice, A. Cichocki, G. Flowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. InfoSleuth: Semantic Integration of Information in Open and Dynamic Environments. In *Proc. ACM SIGMOD Conf.*, pages 195–206, Tucson, Arizona, 1997.
5. S. Bergamaschi, S. Castano, and M. Vincini. Semantic Integration of Semistructured and Structured Data Sources. *SIGMOD Record 28:1*, March 1999.
6. J.A. Blakeley, W.J. McKenna, and G. Graefe. Experiences building the open oodb query optimizer. In *Proc. of the ACM SIGMOD Conf.*, 1993.
7. C.-C. K. Chang and H. Garcia-Molina. Approximate Query Translation (Extended version). Technical Report SIDL-WP-1999-0115, `http://-dlib2.stanford.edu/cgi-bin/get/SIDL-WP-1999--0115`, October 1999.
8. C.-C. K. Chang and H. Garcia-Molina. Mind Your Vocabulary: Query Mapping Across Heterogeneous Information Sources. In *Proc. of the 1999 ACM SIGMOD*, June 1999.
9. S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your Mediators Need Data Conversion! In *ACM SIGMOD Int. Conf.*, pages 177–188, 1998.
10. S. Decker, M. Erdmann, D. Fensel, and R. Studer. Ontobroker: Ontology Based Access to Distributed and Semi-Structured Information. In *Database Semantics - Semantic Issues in Multimedia Systems, IFIP TC2/WG2.6 Eighth Working Conference on Database Semantics (DS-8)*, pages 351–369. Kluwer, 1999.
11. H. García-Molina, Y. Papakonstantinou, D. Quass, A. Rajarman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems 8:2*, pages 117–132, 1997.
12. M. R. Genesereth, A. M. Keller, and O. M. Dushka. Infomaster: An Information Integration System. In *In Proc. ACM SIGMOD Conference*, Tucson, 1997.
13. 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.
14. G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
15. O. Lassila and R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. `http://www.w3.org/TR/REC-rdf-syntax/`, 1998.
16. B. M. Leiner. The NCSTRL Approach to Open Architecture for the Confederated Digital Library. *D-Lib Magazine*, December 1998.
17. 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.
18. B. Ludäscher and A. Gupta. Modeling Interactive Web Sources for Information Mediation. *Intl. Workshop on the World-Wide Web and Conceptual Modeling (WWWCM'99)*, 1999.
19. 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.
20. The OMG Homepage. `http://www.omg.org`, 1999.
21. 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.
22. A. Paepcke, C.-C. K. Chang, H. Garcia-Molina, and T. Winograd. Interoperability for digital libraries worldwide. *Communications of the ACM*, 41(4): , April 1998.
23. Y. Papakonstantinou, H. García-Molina, and J. Ullman. MedMaker: A Mediation System Based on Declarative Specifications. In *Proc. Intl. Conf. on Data Engineering*, pages 132–141, New Orleans, USA, 1996.
24. R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proc. 2nd ASPLOS Int. Conf.* Computer Society Press, 1987.
25. 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.
26. E. Sciore, M. Siegel, and A. Rosenthal. Using semantic values to facilitate interoperability among heterogeneous systems. *ACM Transactions on Database Systems*, 19(2):254–290, 1994.
27. Simple Digital Library Interoperability Protocol. `http://www-diglib.stanford.edu/~testbed/doc2/-SDLIP/`, 1999.
28. 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.