Constructing XML-speaking Wrappers for WEB Applications: Towards an Interoperating WEB

Eleni Stroulia, Judi Thomson, Gina Situ
Computing Science Department
University of Alberta
Edmonton, AB T6G 2H1, Canada
{stroulia,thomson,qihua}@cs.ualberta.ca

Abstract

In this paper, we discuss an architecture for integrating WWW *applications that offer information and services in the same domain. At the center of this architecture exists a mediator, whose responsibilities are to interact with the user and to effectively exchange information with the underlying applications in order to accomplish the user's task. The integration and interoperation of the existing applications are based on the availability of a common domain model, explicitly represented in XML. More specifically, we have developed a general method for constructing wrappers for web-based applications, so that they exchange data with shared semantics such as defined in the XML domain model. At run-time, the user's requests result in the mediator's XML queries to the applications wrappers, which, in turn, invoke appropriate methods on the wrapped applications and extract XML data from their responses to these queries.*

1 Introduction

Today's plethora of applications available on the World Wide Web presents a great opportunity to consumers who can access, compare and combine a variety information and services using their browsers. This availability also presents a great challenge: different applications, even if they offer related information and services, use different technologies for presenting and receiving information from their client-side interfaces (ASP, java, javascript, perl etc.), different models for the system-user interaction (menu-driven, form fill-in, or command-driven) and different models for their application domains. Fueled by ever-increasing e-commerce needs, existing web-based applications are continuously being reengineered to offer more services and to interoperate with other applications in aggregate sites (portals) offering complex services such as comparative shopping and price negotiation for combined client orders.

The major impediment to such reengineering efforts is the lack of any agreement regarding the structure and the semantics of the information provided and required by these WWW applications. The present language of the WWW (HTML) only specifies how the material should appear to the user. Web-based applications expose HTML forms to their users' browsers, possibly enhanced with client-side scripts in different languages. The user has to appropriately interpret the semantics of the information required by the form and to fill it out correctly. Then, the server application responds with another HTML document containing information that the user can interpret as an answer to his original request. It is up to the user, then, to combine the information in the responses of multiple applications and, possibly, to use it to formulate new requests to yet other applications.

Integration frameworks such as DCOM and CORBA, provide a protocol for advertising, requesting and delivering services distributed within a network. These frameworks propose to repackage portions of (or even whole) existing applications as object libraries. The application services are specified using the framework IDL (Interface Definition Language) and then delivered to requesting clients by the framework ORB (Object Request Broker). The utility of these frameworks is limited because the frameworks of different vendors do not easily integrate with each other and, even more importantly, because they offer no support for semantic interoperability. That is, for every two applications that might interact, the internal processing assumptions that they make about the information they exchange must be checked for inconsistencies and any necessary translation needs to be explicitly compiled in one or both of the packages.

An alternative approach to the integration problem is the development of declarative, intermediate representations for domain-specific semantics. If each application explicitly specifies the semantics of the information it re-

1095-1350/00 $10.00

quires and provides, then any other application can check this specification against its own input and output specifications. Thus, these semantically rich representations can provide a "semantic glue" among interoperating WWW applications. Existing applications that do not "speak" these protocols can be "wrapped" with intelligent adapters whose role will be to use the native interaction model of the application to "call" the appropriate application methods, and to translate the results of these calls into messages conforming to the communication protocol. In addition, specific types of intelligent intermediary brokers can be developed to understand and execute value-added services, such as price negotiation or comparison shopping. The eXtensible Markup Language (XML) is an ideal candidate for such a declarative representation, by providing for semantic annotation of information, as well as providing for presentation information in the form of style sheets.

In this paper, we discuss an architecture for developing aggregate applications by combining existing WWW applications and enabling them to interoperate through declarative XML-based representations. In this architecture, WWW applications in a specific domain are encapsulated within *wrappers* that interact with a *mediator* that acts as an intelligent, task-specific intermediary. The mediator has a canonical model of the domain to interpret the information provided and used by the integrated applications, and a task model to coherently integrate its interactions with the user and the underlying applications. In order for a new application to be integrated, a set of rules must be formulated, for how to reverse engineer structured instances of particular domain-model entities from the application's response to a particular information request. This rule set is then plugged in a generic wrapper component of the architecture, creating a wrapper for the particular resource application.

The remainder of this paper is organized as follows: section 2 discusses the overall mediation architecture and illustrates the processes of the mediator and the wrappers using our travel-planning prototype as an example, section 3 discusses the general wrapper of the architecture, 4 discusses the specific wrapper construction problem on which this work focuses and the process developed to address it, section 5 discusses our evaluation of this process and some limitations we discovered from our experiments with it, section 6 puts this work in the context of other related work, and finally section 7 outlines some early conclusions that can be drawn from this work and outlines plans for future work.

2 System Architecture and Process

To illustrate the integration approach proposed by this work, let us discuss a particular instance of the WWW application integration problem. Today there are a multitude of WWW sites offering travel planning and reservation services. Knowledgeable consumers, with specific constraints and preferences, access several different sites to identify available options. Then they have to compare the results from these different sites prior to making a decision. We have integrated services and information from four different existing web servers to develop a travel-planning application that enables travelers to collect and compare travel plans from different sources.

The overall architecture of the resulting integrated application, shown in Figure 1, consists of four conceptual layers: the user interface layer, the mediator layer, the wrapper layer and the resource layer. These are mapped on three physical layers, with a thin-client interface, a set of existing resources at the lowest tier and the "business logic" of the integrated application, consisting of the mediator and the wrappers, in the middle tier.

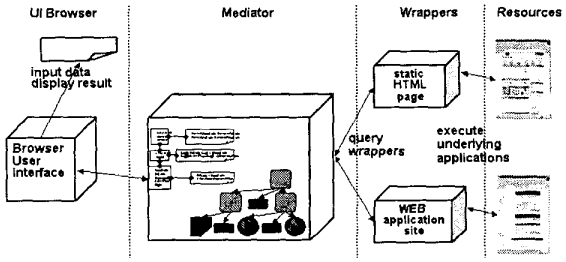


Figure 1: The Mediation Architecture.

The domain and task model required for the integration of a set of WWW applications are represented explicitly in XML in the mediator layer. The user interacts with the mediator through an XML-enabled WWW browser, such as Internet Explorer. We have developed a general XSL[1] stylesheet for presenting task models as hierarchical menus: the high level tasks of the mediator correspond to the top level menus whose options become subtasks represented in sub-menus. This gives the mediator of the integrated application a simple intuitive interface through which the user can access and traverse its task structure. To initiate a session with the mediator, the user must specify the high-level task desired for the session. In response, the mediator retrieves the task structure corresponding to the selected task, i.e., the set of simpler tasks into which the selected task can be decomposed, and recursively descends from the high-level task to the subtasks, while exposing the corresponding menus to the user.

Figure 2 depicts part of the travel mediator's task model. The overall task of the mediator is to *find airfares*

[1] Extensible Stylesheet Language (XSL) is a language with capabilities for presentation and manipulation of XML documents [3].

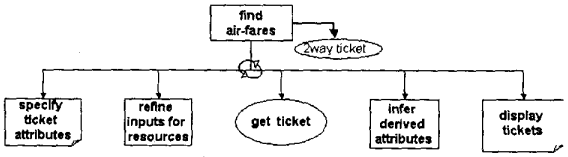


Figure 2: The travel-planning task structure.

which gets decomposed into the tasks of *specifying ticket attributes*, *refining the inputs*, *getting tickets* from the underlying applications, *inferring derived attributes* of the collected tickets, and *displaying* these tickets to the user. There are three different types of tasks in a mediator's task structure:

- *user-interaction tasks* that involve requesting information from (or displaying information to) the user. (subtasks 1 and 5 - denoted by rectangles with the bottom-right corner folded)
- *internal tasks* involving information processing internal to the mediator (subtasks 2 and 4 - shown as simple rectangles in the Figure)
- *information collection tasks* that involve interacting with the wrappers of the underlying applications to collect information (subtask 3 - shown as an oval).

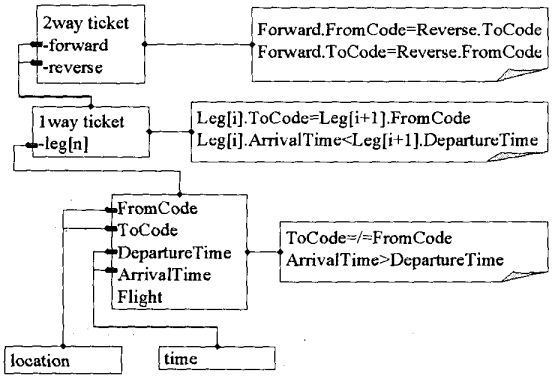


Figure 4: The Domain Model.

Figure 4 shows a part of the travel mediator's domain model. The domain model contains three types of information

1. the names and attributes of the entities that exist in the domain,
2. the relationships between these entities, including inheritance and composition relationships, and
3. the invariants of these entities.

In the model of Figure 4, a *2way ticket* is composed of two *1way tickets*, such that the origin of the first is the destination of the second and vice versa. Each *1way ticket* is described in terms of the airport codes of the origin and destination cities, *FromCode* and *ToCode* correspondingly, which are both types of locations, the departure *DeptTime* and arrival *ArrTime* times, and a *Flight* number.

As previously mentioned, some of the mediator's tasks involve interaction with the user. These tasks require user input for (or display to the user of) the values for some of the domain-model entities. For each entity in the XML domain model that must be input by (displayed to) the user, there exists an associated XSL stylesheet that provides an interface for the data input (display). When the user selects a *user interaction* task from the mediator's task-structure menu, the mediator sends to the user's browser the appropriate form. For example, in the case of the travel mediator, several different types of *location*s and *time*s must be input by the user. As it can be seen from the Figure 2, the first subtask of the overall task is to specify ticket attributes. The selection of origin and destination cities are subtasks of this task. When, in the process of finding an air fare, the user reaches the task of selecting the origin and destination, i.e., *FromCode* and *ToCode*, of the travel, the mediator sends to the browser the *location* entry form, which is generated by the *location* XSL stylesheet. Another *user-interaction* task is final ticket display: a collection of 2way-tickets is sent back to the user's browser, where it is presented as a table sortable by any of the ticket attributes.

Figure 3 illustrates the user interfaces generated by the XML domain and task models and their corresponding XSL stylesheets. From right to left, one can see the task-structure based menu, the location data entry form, the summary output including problem specification and a table of the results, and the full ticket listing.

When an *internal* task is reached, the mediator decomposes it further, if necessary, and then invokes an internal function to process the information. An internal task might be invoked to infer additional information from the user's input or to process the information collected by the wrappers before presenting it to the user. Internal tasks allow processing of the data using application-specific logic. For example, after the mediator has collected the user input, the *internal* task of refining these inputs is reached. This task involves the elaboration of the input to meet the level of detail expected by the wrappers. For example, while the user may specify the departure time as a date, some

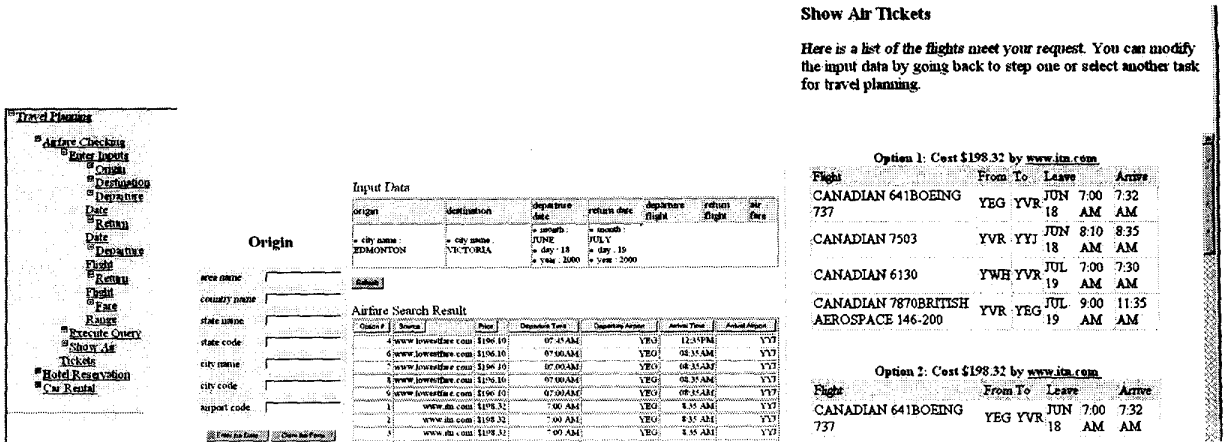


Figure 3: Snapshots of the travel-assistant user interface.

resources require a time-of-day or even hour information; the mediator can infer these more specific values from the user's input.

When an *information-collection* task is reached, the mediator identifies the application that supplies the necessary information. The mediator sends a request to the corresponding wrapper. For example, when the *information-collection* task of getting tickets for a specific travel problem is reached, the travel mediator sends the problem specification to the wrappers of the travel-planning applications and requests their solutions. Each wrapper, in turn, calls the appropriate method of its associated underlying application and extracts *2way tickets* from the response which are then returned to the mediator.

3 The Resource Wrapper

Information-collection tasks are initiated by the mediator and performed by the wrappers of the underlying applications. A wrapper must deliver two important functionalities:

1. map the problem request to a set of parameters for a particular method of the resource and invoke this method, and
2. parse the application's response to extract the relevant pieces of information expected as an answer by the mediator.

Most WWW applications provide information encoded using HTML. Our wrapper construction process assumes that the information in the underlying application's response is encoded in well-formed HTML (or possibly converted to well-formed HTML using an application like the HTML Tidy [4] utility). Presently we cannot automatically create wrappers for information that would normally require a browser plug-in for presentation (e.g. PDF or Shockwave). Henceforth, the term "response" (R) will refer to HTML documents returned by an application in response to a request for information (via some $HTTP :: GET/POST(\{param_i\}*)$ method). More specifically, we are interested in a particular class of wrappers that extract a set of multiple instances of an entity from the mediator's canonical domain model.

Each wrapper in this integration architecture contains the following information:

1. the canonical domain model of the mediator,
2. the URL of the wrapped application,
3. the method of the application that the wrapper invokes, the required parameters for this method and how to map a problem specification represented in terms of the domain model in a method call,
4. a domain-model extension that describes, XPATH[2] rules characterizing the location of instances of domain-model entities in the application's response,
5. a driver component that invokes the method on the underlying application, and
6. a parser that uses the information above to interpret the application's response.

The original domain model is shared by the mediator and all wrappers. The extension to the model, which is specific to each wrapper, defines a grammar, that is a set

[2]XPATH is a language for addressing parts of XML documents [2].

of rules, for appropriately interpreting the application's response to a particular method invocation in terms of the common domain model. The process of wrapper construction (described in section 4) involves automatically learning this domain-model extension.

For each request received from the mediator, a wrapper completes the following process:

1. it maps the problem specification provided by the mediator to the required parameters for the appropriate resource method,
2. it invokes the method,
3. it extracts the information contained in the response according to the XPATH rules specified in the domain-model extension, and
4. it returns this information to the mediator.

4 Wrapper Construction

The most critical step in the process of developing an integrated application, such as the travel-planning assistant described above, is the construction of the existing applications' wrappers. This is a "reverse engineering" process in nature: its objective is to extract a set of rules for mapping the application's original response to a common target representation, that is, the domain model.

Many wrapper-construction methods parse HTML documents using a linear inspection [6]. They attempt to identify irrelevant parts at the beginning and the end of the HTML document, and use that information to generate rules for parsing the rest for potentially useful information. Dynamically generated documents can be a challenge for such approaches, since the size, the appearance and the types of information contained in these irrelevant parts may change from request to request.

In this work, the hierarchical structure of HTML documents is traversed to select and inspect their contents. HTML pages served by WWW applications in response to a particular request are usually automatically generated. Although individual responses may differ in content, all responses from the same application for the same type of request usually have a similar organization. This implies that two HTML documents served by an application as responses to the same type of request are more likely to have commonalities at (and close to) the documents' roots than at (and close to) their leaves. After the HTML document has been parsed into a tree representation rooted at the $< html >$ tag, document subtrees that may contain information of interest can be efficiently located, using a DOM-like API [1].

Our approach to wrapping WWW applications relies on the use of XML technology for creating a semantic map of documents. As we have already mentioned, the mediator's domain model is represented in XML. When the mediator sends an information collection request to a wrapper, it expects to receive an XML document containing instances of some entity in the domain model, henceforth called the target concept (C_t)[3]. More specifically, this method is designed to deal with applications whose responses possibly contain more than one instances of the target concept.

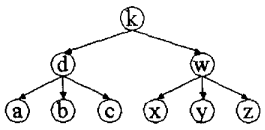
XML documents are hierarchical in nature. An XML element is composed of simpler XML elements. In order to extract an instance of the target concept from the application's response, instances of all the target-concept's constituents must be identified.

Figure 5 at the top illustrates a hypothetical target concept k that consists of two simpler concepts, d and w, which in turn get decomposed into a, b, c and x, y, z correspondingly. The set of k's constituent components consists of all the concepts represented by the trees rooted at k, d, w, a, b, c, x, y, z.

Let's assume that there are at least two instances of k (and consequently two instances of each d, w, a, b, c, x, y, z) in a particular HTML document, produced as a response of the application to be integrated. If all the parts of one instance of concept k are separated from all the parts of all other instances of k, i.e., the instances of the constituents of the different concept instances are not intermingled, then k is said to be *contiguous*. So for example the three trees at the bottom half of Figure 5 represent the internal structure of three different HTML documents. Each circular node in the tree represents a pair of matching HTML tags enclosing a part of the document. The label on the node corresponds to the entity of the target concept contained in this part of the document. k is contiguous in the first two trees in Figure 5. However, in the first tree, it is also *encapsulated*, i.e., there is a distinct pair of HTML tags completely enclosing all of the parts of one instance and none of the parts of any other instance. k is not encapsulated in the second tree, but d and w are. The third tree represents a case where k is not contiguous, i.e., the component instances of the different concept instances are interspersed in the HTML page.

In order to extract the instances of k from an HTML document, rules must be formulated for how to traverse the tree-structure of the document in order to locate the instances of the constituents of k. These rules can be formulated using XPATH rules. Therefore, the main objective of the wrapper construction process is to learn a grammar,

[3]There can be instances of more than one entities of interest contained in the response. However, for the sake of clarity of the discussion we use the singular number. Our method can also deal with multiple target concepts.



A tree representation of a target concept defined by the following XML domain model: <k><d><a><c></c></d> <w><x></x><y></y><z></z></w></k>

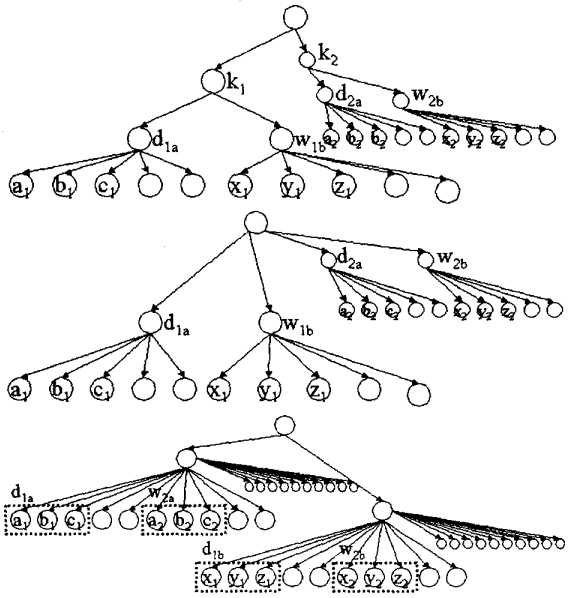


Figure 5: (top) k is the contiguous and encapsulated, (middle) k is contiguous, d and w are contiguous and encapsulated, (bottom) d and w are contiguous but not encapsulated.

specified as XPATH rules, for the locations of the instances of the leaf components in the target concept set.

The contiguity and encapsulation properties of the target concept in the application response have an interesting effect on the structure of the grammar. If the concept is encapsulated, then an XPATH rule can be formulated in order to localize the HTML subtree that spans all the concept constituents. The rules for its constituents can then be formulated relative to the root of this subtree. If the concept is not encapsulated but it is contiguous, then a set of indexed XPATH rules can be formulated in order to locate its constituents[4]. Finally, if the concept is not contiguous, then rules for the extraction of its elements must be discovered and then the constraints describing the invariants of the concept may be used to combine them into concept instances. We discuss in detail the nature of these rules and the algorithms to learn them in the next subsection.

The overall process for wrapper construction is shown in Figure 6. It consists of a demonstration phase and a learning phase. The purpose of the demonstration phase, which is not completely automated yet, (step 1 in Figure 6) is to collect examples, on the basis of which both the parameters of the method to be invoked on the underlying application and the grammar for parsing its response can be learned. The input to the demonstration phase is the specification of a set of problems and their solutions that the application can accomplish. The underlying idea is that when integrating a new application, the information-collection task that its wrapper is expected to accomplish is already known, and therefore examples of this task can be specified. The demonstration phase relies on a proxy, based on the Muffin library [5]. The purpose of the proxy is to sit between the web server and the user's browser and to record the interactions between them, while the user sends a sequence of requests to the server that accomplish the functionality of the resource to be wrapped.

1. Let a user demonstrate the use of the resource to be wrapped and generate the training examples
   (a) Define a set of example problems for the user to demonstrate
   (b) For each such problem

   While the user at his browser interacts with the resource server to solve the problem, record the trace of the interaction and the final response page that contains the instances of the target concept that are the solution to the current problem.
2. Learn the protocol of the interaction with the resource
3. Learn the grammar for extracting the instances of the target concept from the resource's responses

Figure 6: The Overall Process.

The next phase of the wrapper-construction process is the learning phase (steps 2 and 3 in Figure 6). The objective of this phase is twofold. First, the mapping of the problem specification into an appropriate method invocation to the underlying application server must be learned. Second, the grammar rules for parsing the response into the corresponding solution must be formulated. In this paper, we focus on the second task.

[4]The underlying assumption is that the concept instances have a common internal structure

4.1 The Grammar Learning Algorithm

The most challenging task of the learning phase is to learn the XPATH rules for extracting each component of the target concept from the response. The algorithm for learning how to extract instances of an encapsulated concept from the HTML response of a resource is shown in Figure 7. Given a set of specific examples of the target concept and a resource response that is known to contain the given examples, the learning algorithm generates a set of XPATH rules that will extract the information for all instances of the target concept from resource responses with the same structure. This section illustrates the given algorithm with a small example from a fictitious glossary web page.

Suppose that the resource response consisted, in part, of a glossary containing a list of words and their definitions. Further suppose that the task is to find the rule for extracting all the entries in the glossary, but none of the additional information on the page. Let us examine the process required to create a grammar for extracting the word-definition pairs from this hypothetical example.

In the glossary example, the target concept is the glossary *entry* and its components are the *word* and its *definition*. Figure 8(a) shows three instances of the target concept that are included in the resource response. These instances are part of the algorithm's input. Let us assume that each glossary entry in the HTML response of the resource is contained in a separate row of a table and the word and its definition are data fields in the table. Let us further suppose that the word of each entry is red. In the particular response page, the entries for the words "abbey", "abduct" and "capital" are located in the fourth, seventh and eighth rows of that table.

The first step of the algorithm (steps 1.a and 1.b) is to use a post-order search-traversal of the response document tree, to identify the locations of the entries on the page and to formulate the XPATH rules for these locations. Figure 8(b) shows the rules for the locations of the components of the three entries in the page. The word "abbey" can be located by the path "html/body/table[5]/tr[4]/td[0]" in the HTML document tree, the definition of the word, "convert" can be located by the path "html/body/table[5]/tr[7]/td[1]", and the word "capital" can be located by the path "html/body/table[5]/tr[8]/td[1]", and so on.

Based on the XPATH rules of the locations of its components, the rule for the location of the target concept instance can be formulated (step 1.c). The XPATH rule for the location of each *entry* is the minimum spanning tree containing its *word* and *definition*, i.e., the maximum common prefix of the locations of its constituents. In this example then, the location of the "abbey", "abduct" and "capital" entries are at "html/body/table[5]/tr[4]/", "html/body/table[5]/tr[7]/" and "html/body/table[5]/tr[8]/"

1. FOR EACH instance, p, of the concept c_t
   (a) locate the components of the instance
   (b) identify the XPATH rule, consisting of the index and the attribute list, for the location of the instance components
   (c) identify the XPATH rule, consisting of the index and the attribute list, for the location of the instance as the minimum spanning tree of its components locations
   (d) formulate the locations of the instance components, relative to the instance location
      i. IF the path expression of its parent component is the prefix of the current path rule, remove the prefix from the current hypothesis
      ii. IF it is a parent component, generalize the index of the last tag of the path to include all
2. FOR EACH index of every rule hypothesis
   IF it is constant, remove its associated attribute list
3. IF the set of XPATH rules for the examples vary only by index number, formulate one of the following (and increasingly general) hypotheses for the rule $xp(c_t)$
   (a) FOR EACH variable index, generalize the expression to a linear progression of the index values
   (b) FOR EACH variable index, generalize the expression to include all possible index values

Figure 7: The algorithm for learning grammar rules for encapsulated concept.

correspondingly. The fourth column of Figure 8(b) shows the locations of the constituents of each instance relative to the location of the instance itself, produced by step 1.d of the algorithm.

At this point, step 1 of the algorithm shown in Figure 7 has been completed. The goal of the next step is to formulate, first, an abstract hypothesis for the location of all *entry* instances in the resource response relative to the root of the document tree, and second, a set of hypotheses for the locations of the concepts' constituents, i.e., *word* and *definition*, relative to the locations of the concept instances.

Many of the resources' servers respond with HTML pages constructed by scripts collecting data from a database. These pages are highly regular in structure, and

more often than not, the XPATH rules of the locations of all the instances are the same with the exception of the indices for some of the path tags. Thus the generalization step attempts to discover a regularity in the values of the differing indices. This regularity may be that all values are possible, or that the possible index values form a linear progression (step 3).

```
<dictionary>
  <entry>
    <word>abbey</word>
    <definition>convent</definition>
  </entry>
  <entry>
    <word>abduct</word>
    <definition>kidnap</definition>
  </entry>
  <entry>
    <word>capital</word>
    <definition>money</definition>
  </entry>
</dictionary>
```

(a) An XML document with two sample instances of glossary entries.

Example 1			
Entry:		/html/body/table[5]/tr[4]/	
Word:	abbey	/html/body/table[5]/tr[4]/td[0]	td[0]
Definition:	convent	/html/body/table[5]/tr[4]/td[1]	td[1]
Example 2			
Entry:		/html/body/table[5]/tr[7]/	
Word:	abduct	/html/body/table[5]/tr[7]/td[0]	td[0]
Definition:	kidnap	/html/body/table[5]/tr[7]/td[1]	td[1]
Example 3			
Entry:		/html/body/table[5]/tr[8]/	
Word:	capital	/html/body/table[5]/tr[8]/td[0]	td[0]
Definition:	money	/html/body/table[5]/tr[8]/td[1]	td[1]

(b) The XPATH rules for the locations of the two glossary entries in the response.

Entry:	/html/body/table[5]/tr
Word:	td[0]
Definition:	td[1]

(c) The generalized XPATH rules for the location of the glossary entries.

Figure 8: A grammar-learning example.

From the XPATH rules for the location of the *entry* instances, shown in Figure 8(b), the learning algorithm can determine that the target concept (a glossary entry) is found consistently in rows of the fifth table of the document. The generalized XPATH rule for the location of all the glossary entries is "/html/body/table[5]/tr", i.e., the index of the row has been generalized to allow any value. Because this is a quite aggressive generalization step, the attributes of presentation of each entry, i.e., the color red, are kept as part of the XPATH rule for its location.

The same generalization process applies to the XPATH rules of the relative locations of the sub-components of the entry. In this example, the word and definition of each entry can be found in the first and the second cells of the entry row. The subcomponents locations then are "td[0]" and "td[1]". The final rule for extracting the instances of the *entry* concept in the resource response is shown in Figure 8(c).

If the "capital" instance were located in the tenth row instead, the algorithm would have chosen to generalize the expression for the *entry* location to "/html/body/table[5]/tr[4+3*i], i=0,1,.." instead.

If the target concept is not encapsulated, step 1.c of the algorithm will return the XPATH rules for extracting the constituents of the target concept instead of the target concept itself. The generic wrapper component of the architecture is capable of using these fragments of the target concept and the invariants of the target concept as defined in the domain model to combine the fragments into instances of the target concept. This way, if sufficient invariants are defined for the target concept, it can be extracted even if it is not encapsulated in the response of the application to be wrapped.

5 Evaluation

The representation language for specifying the wrappers is quite general, in fact we have constructed wrappers by hand that conform to the same representation and behave similarly at run-time as the learned ones.

The grammar-learning algorithm has been evaluated using both generated test data and actual HTML pages from travel-planning web applications. The generated test data consisted of encapsulated, contiguous target concepts in a variety of HTML structures (i.e. lists, tables, paragraphs). The algorithm was able to successfully learn the grammar when given one exhaustive annotated example. The learned grammar was then used to extract all instances of the encapsulated concept from the generated pages.

The algorithm was also used to wrap two popular travel planning web sites: www.expedia.com and www.itn.com. The target concept was encapsulated in the first and contiguous but non encapsulated in the second. We encountered an interesting problem in the effort to wrap these sites. Although all elementary (leaf) entities of the target concept existed in the applications' responses some of the intermediate constituents were not encapsulated. For ex-

ample, in Expedia the ticket concept is encapsulated but the forward and reverse legs are not. This means that the forward and reverse legs are not delimited by HTML tags in the applications response. In fact, the ticket consists simply of a set of hops, some of which constitute the forward leg and some the reverse. In this case, the algorithm fails to identify the XPATH rules for the forward and reverse legs. This problem is basically due to a mismatch between the structure of the concept in the domain model and the underlying database schema in the application: the first assumes a more complex structure than the second. At this point, to address this problem, we present to the algorithm the examples of the target concept assuming different variants of the canonical domain model. As soon as the examples of one variant are identified, the grammar for this variant is produced. Instances of this variant are reformulated into the canonical structure using the concept invariants. We are currently investigating the possibility of constructing the various variants of the domain model, and the corresponding restructuring of the input examples, automatically. A further complication occurs when the concept is not encapsulated in the application's response, as is the case of ITN. In this case, the algorithm fails to identify the examples of all concept variants. It then hypothesizes that the concept is not encapsulated and attempts to identify instances of its constituents, starting from the more complex towards the simpler ones. Again, once a grammar has been formulated, it can be used to produce a wrapper for this application that will construct instances of the target concept, as defined in the canonical model, based on the invariants of this concept.

The algorithm also faces problems if the HTML response of the application has more than one constituents of the target concept in the same HTML node, such as origin and destination of leg for example. An extension is needed to be able to further specialize the learned XPATH rules with expressions of string delimiters in addition to HTML tags. The wrapper-construction algorithm is still limited in that it does not address the problem of learning non-contiguous concepts. We are currently working on extending it in this dimension and we believe that the invariants of the domain model will be extremely useful in this case.

In summary, the algorithm is useful but still quite brittle and limited. As the complexity of the concepts to be extracted from the applications responses increases and the structure of the response HTML document becomes more flat, the algorithm becomes less effective. This is not surprising, since it corresponds to requiring the extraction of more information from less. We believe however that the approach exemplified by this algorithm is promising, since to our knowledge related research until now has focused mostly on simpler concepts.

However, the most challenging problem we have encountered in the process of integrating WWW applications is the fact that the different based applications change the structure of their responses quite often. Today we recognize that the underlying wrapped application has been changed when the resources wrappers fail at run-time. A more robust technique is needed to recognize when the wrapper must be updated and possibly automatically proceed to the learning process at run time.

6 Background and Related Work

Recently there have been several efforts to extract specific information from data available on the WWW. W4F [12] and XWrap [11] are tools that provide a "wizard" like interface for users to define and test extraction rules. Once the user is satisfied that the rules defined extract the elements of interest from a web page, then they create a java class defining the rules as a wrapper that can collect the extracted data and place it in a XML document. W4F requires the user to define the rules for both extracting the data from the HTML document and mapping the extracted data to an XML document. It uses its own languages, HEL(HTML extraction language) and NSL(nested string language), for describing the extraction rules and storing the extracted data, respectively. While XWrap provides a more automatic "wizard" which generates the extraction rules by allowing the user identify the key words and the presentation layout structure. It embeds the extraction rules in an XML document. XWrap extracts data from only three types of regions from the HTML document, table, list and paragraph.

Artificial-intelligence approaches have focused on learning wrappers automatically. Kushmerick [10] categorized wrappers into six different classes of complexity and described inductive learning algorithms for the construction of wrappers of each type, where instances correspond to pages, labels correspond to the pages' content, and hypotheses correspond to wrappers. WebKB [8] developed a model-based method for extracting data spanning across several web pages while describing rules in first-order representation.

Finally, Ariadne [6] exemplifies research in the second category. This project has focused on developing an infrastructure for integrating the wrapped resources of semi-structured documents using a planner. The MIX project [7] also belongs in this category. MIX takes a database-centric approach to integration and develops a query planning mechanism for querying various resources, including databases, GIS systems and Web sites, viewed as XML databases. Both Ariadne and MIX, also offer support in the construction of wrappers. Ariadne describes semi-

structured document in embedded catalog formalism and extraction rules as finite automata, while MIX employs XML for information modeling.

7 Summary and Conclusions

Incompatible interaction models and differences in the information semantics prevent users from combining the services of different web sites. Although these incompatibilities may be intentional in some cases, to prevent service comparison, WWW users are provided with more functionality if related applications could be reengineered to interoperate to a degree. We have presented an approach to WWW information integration, based on the development of a canonical domain model in XML and the wrapping of existing WWW applications with wrappers capable of communicating about entities in this common model with the applications and with an intermediary mediator.

The most interesting features of this approach are the following:

- All internal representations (task model, domain model, grammar) are represented in XML. These representations enable the lightweight construction of simple and consistent user interfaces. Furthermore, in combination with standard XML-related languages and APIs (XSL, DOM and XPATH) they provide the basis for the construction of wrappers and the communication between the mediator and these wrappers at run time.
- The basis of the integration is the domain and task models of the overall application. The domain model explicitly represents the agreed upon domain entities and their semantics and the task model represents the control of information exchange and interaction between the user and the underlying applications.
- The wrapper-construction process takes advantage of the hierarchical structure of the HTML document to generate rules for extracting multiple instances of complex domain entities interspersed among "garbage" HTML content, without requiring special landmark tags.
- Finally, the wrapper-construction process uses the explicit representation of the invariants of the domain-model entities to extract complex concepts whose constituents are not encapsulated in the HTML documents produced by the wrapped applications.

Acknowledgements

This work was supported by a research grant by NSERC 203221-98.

References

[1] Document Object Model (DOM) Level 2 Specification http://www.w3.org/TR/1999/CR-DOM-Level-2-19991210/

[2] XML Path Language (XPath) Version 1.0, W3C Recommendation 16 November 1999 http://www.w3.org/TR/xpath

[3] Extensible Stylesheet Language (XSL) Version 1.0 W3C Working Draft 27 March 2000 http://www.w3.org/TR/xpath

[4] http://www.w3.org/People/Raggett/tidy/

[5] Muffin, World Wide Web filtering system, *http://muffin.doit.org/*

[6] J.L. Ambite, C. A. Knoblock. Flexible and Scalable Query Planning in Distributed and Heterogeneous Environments Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems, Pittsburgh, PA, 1998

[7] C. Baru. Xviews: XML Views of Relational Schemas, Intl. Workshop on Internet Data Management Florence, Italy, Sept.1-4, 1999.

[8] M. Craven, D. DiPasquo, D. Freitag, A. McCallum, T. Mitchell, K. Nigam and S. Slattery. Learning to Extract Symbolic Knowledge from the World Wide Web. Proceedings of the Fifteenth National Conference on Artificial Intelligence, Madison, WI, 1998.

[9] C.A. Knoblock, S. Minton, J.L. Ambite, N. Ashish, P.J. Modi, I. Muslea, A. G. Philpot, S. Tejada. Modeling Web Sources for Information Integration, Proceedings of the Fifteenth National Conference on Artificial Intelligence, Madison, WI, 1998.

[10] N. Kushmerick: Wrapper induction: Efficiency and expressiveness. To appear, J. Artificial Intelligence, 2000.

[11] L. Liu, C. Pu, W. Han, D. Buttler, W. Tang. An XML-based Wrapper Generator for Web Information Extraction, To appear in the Proceedings of the ACM SIGMOD International Conference, June 1-4, Philadelphia.

[12] A. Sahuguet, F. Azavant. Looking at the Web through XML glasses. CoopIs'99 (1999)