

# Wrapper Generation for Web Accessible Data Sources \*

Jean-Robert Gruser, Louiqa Raschid, María Esther Vidal, Laura Bright  
University of Maryland  
College Park, MD 20742  
{gruser,louiqa,mvidal,bright}@umiacs.umd.edu

## Abstract

*There is an increase in the number of data sources that can be queried across the WWW. Such sources typically support HTML forms-based interfaces and search engines query collections of suitably indexed data. The data is displayed via a browser. One drawback to these sources is that there is no standard programming interface suitable for applications to submit queries. Second, the output (answer to a query) is not well structured. Structured objects have to be extracted from the HTML documents which contain irrelevant data and which may be volatile. Third, domain knowledge about the data source is also embedded in HTML documents and must be extracted. To solve these problems, we present technology to define and (automatically) generate wrappers for Web accessible sources. Our contributions are as follows: (1) Defining a wrapper interface to specify the capability of Web accessible data sources. (2) Developing a wrapper generation toolkit of graphical interfaces and specification languages to specify the capability of sources and the functionality of the wrapper. (3) Developing the technology to automatically generate a wrapper appropriate to the Web accessible source, from the specifications.*

## 1. Introduction

The WWW has increasingly gained acceptance as a medium for querying and information exchange. However, the interfaces and protocols, e.g., data exchange formats, utilized on the WWW, are significantly different from query and answer exchange supported by a DBMS. Typically, Web accessible sources support HTML forms-based interfaces. Search engines use information retrieval techniques to query collections of suitable indexed data typically resident within a file system. In some cases, the data may be obtained from

a database. The information in the output is in a format to be displayed via a browser (HTML), and is not suitable for access by an application program that expects structured data.

Providing an application programming interface to such sources, in a manner similar to accessing data in a database, presents special problems. A Web accessible source typically does not support a schema, nor does it support a set of operators such as the set of standard (relational) operators or operators supported by an IR search engine. Querying the data in the source requires generating an appropriate URL for the document that contains the answer. Constructing the URL is quite different compared to translation from a set of standard operators to another set of operators native to a database-like source. The answers are not strictly typed objects in some standard representation. Instead, the answers have to be extracted from the semi-structured data stored in HTML documents, and these documents may contain data that is irrelevant to the query. The structure of these documents may be volatile and this affects the extraction process. Domain knowledge about the data source is also embedded in HTML documents and must be extracted.

Our objective, described in this paper, is generating a new paradigm of JDBC compliant *wrappers* [18] for Web accessible sources. In addition to answering queries, the wrapper will provide information on the capability of the sources, i.e., what queries they will answer, and the output types. The wrappers will also provide a variety of meta-information, e.g., domains, statistics, index lookup, etc., about the sources. Our research will develop the technology to define and (automatically) generate wrappers for Web accessible sources. Our contributions include the following: (1) Defining the capabilities of Web sources and defining a wrapper protocol to publish this capability; (2) Developing a wrapper toolkit that provides a graphical interface and specification languages to specify the capability of a source and the functionality of the wrapper; and (3) Developing the technology for a wrapper generator which produces Web wrappers.

To motivate the first task of defining the capability of

---

\*This research has been partially supported by the Defense Advanced Research Project Agency under grant 01-5-28838; the National Science Foundation under grant IRI9630102 and by CONICIT, Venezuela

a Web accessible source, we note that it does not support a schema and a set of standard (relational) operators. Its query capability is very limited. It may accept a “binding” or selection on some fields and it may require that some fields always have bindings. The output of a query is determined by the data (contents) in the documents that are retrieved and thus must be explicitly specified for each allowed binding. Thus, we must define the capabilities of a Web source so that the wrapper can determine if it can answer a particular query submitted to this source.

To facilitate the task of specifying the source capabilities and the wrapper functionality, we provide a wrapper toolkit which has graphical interfaces and specification languages to support the following features:

- A graphical interface is used to specify the capabilities of the sources, e.g., required bindings, output types, etc. Operators particular to the source, e.g., index lookup; contents of the domains of input attributes; statistical information; etc., are also identified through this interface.
- A graphical interface is used to define a simple query translation and answer extraction process. The process is specified using a wrapper capability table. The wrapper uses this table to convert a query into a URL, and extract answers from the corresponding HTML answer document. The wrapper does so using a URLConstructor and an Extractor to be discussed next.
- The toolkit provides a language to specify the URL-Constructor expression. The URL may represent a document that contains the answers to the query. Alternately, the URL may represent a script that will be evaluated on the source.
- The toolkit provides a declarative query language to describe a simple Extractor that can extract data from a single HTML document, and construct structured objects. The Qualified-path-expression Extractor Language, *QEL*, is used to express queries that describe this extraction process. *QEL* provides special features to (a) extract syntactical structures (tokens) from HTML documents; (b) identify these structures based on their contents (semantics); (c) traverse these structures using qualified path expressions; and (d) manipulate these structures to construct structured objects corresponding to output types.
- The toolkit also supports a Complex Extractor Specification Language, *CESL*. Complex extractors, specified in *CESL*, make the toolkit extensible, since complex extractors may utilize pre-defined capabilities of the wrapper. Complex extractors may be used when answers are to be extracted from multiple HTML documents; the complex extractor iteratively utilizes other

extractors. Complex extractors are also used when the output format of a HTML document may not be known a priori, as is common for many Web accessible sources; the complex extractor would conditionally utilize extractors appropriate to the HTML document.

The final task is to develop a set of utilities to automatically generate and construct a wrapper appropriate to each Web accessible data source. Each wrapper will support a `connect` call which will return a capabilities object. This object will have methods to specify the capabilities of the source and the domains. It may also provide methods to test queries (to test if they will be accepted), and to provide a cost estimation. The query and answer protocol for the wrapper is JDBC compliant [18], when the wrapper interface is defined using the relational data model.

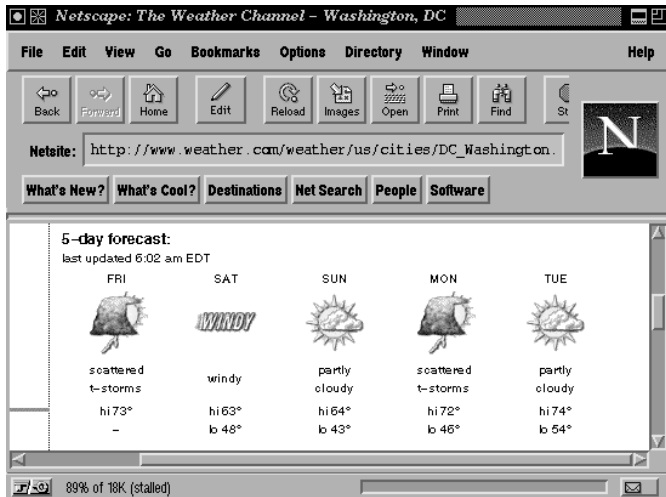
This paper is organized as follows: In section 2, we provide a motivating example of a Web accessible source and its wrapper. In section 3 we define the wrapper protocol and interface. In section 4, we describe the query translation and answer extraction process for a wrapper, using the wrapper capability table. The wrapper utilizes URLConstructors and Extractors. We also describe the graphical interface to specify the functionality of the wrapper capability table. In section 5, we describe the declarative query language *QEL* used to specify simple extractors which construct structured objects from a single HTML document. We present example queries in *QEL* and informally present the semantics of those queries. Next, in section 6, we describe complex extractors which are able to re-use the pre-defined capability of the wrapper. Complex extractors are specified in a language *CESL*. In section 7, we discuss the wrapper generation task and the prototype implementation. Section 8 compares our approach with related projects and concludes with future work.

## 2. Motivating Example

Consider a source that offers weather related information. An example of querying the source with a national (US) city binding, Washington, DC, is displayed in Figure 1.

A five day forecast, each with several fields, is displayed. This source allows a number of bindings, e.g., US states, US cities, international cities, etc.. The information in the answer document depends on the query binding. Thus, we must specify the particular input-output relationship between the fields that are bound in the query and the output data types. For a query with a binding for an international city, the output is the weather forecast for three days. For a query with a binding for a national city, the output is the weather forecast for five days.

For each set of allowed bindings, the translated query (URL) is different and we need to specify the particular



**Figure 1. An example html document**

URLConstructor expression for each binding. For a query with a US city binding, e.g., Baltimore, MD, the URL is:

```
http://www.weather.com/weather/us/
cities/MD_Baltimore.html.
```

Finally, structured data corresponding to the output type(s) must be extracted from the answer HTML document which contains irrelevant data as well. Figure 2 shows that portion of the HTML document with answers for the query with a national city binding.

The answer object is displayed in Figure 3. In Figure 2, each row of the table contains the value for one field, e.g., an image representing the forecast, for all five forecasts. The extractor needs to *transpose* the tabular data, to obtain all of the values for all fields of a particular forecast, for one day. This is the output object shown in Figure 3.

### 3. Wrapper Protocol and Interface

The wrapper comes in two flavors. The relational flavor supports tuples over the Java basic types. The complex object flavor supports user defined complex objects. We first define the Web wrapper protocol and then give an example of a wrapper interface for the Weather source.

#### 3.1. Web Wrapper Protocol






Web wrappers will support the following protocol:

In response to the *connect* method the wrapper returns an object with the following *methods*, that describe the wrapper interface:

- A set of *input attributes* of basic types that may be converted to strings, e.g., Character, String, Number,

```
...
< font face="arial,helvetica" size=4>
< B>Baltimore</B><BR>
...
28< TABLE width=100 cellpadding=2 cellspacing=0 border=0>
29< TR>
30< TD width=20 align=center>< font face="arial,helvetica"
size=2>< b>TUE</b></font></TD>
...
35</TR>
36< TR>
37< TD align=center>
<IMG SRC="/weather/wx_icons/forecast/N.gif"
alt="showers" width=52 height=52></TD>
...
42</TR>
43< TR>
44< TD align=center>< font face="arial,helvetica"
size=2>< b>showers</b></font></TD>
...
49</TR>
50< TR>
51< TD align=center>< font face="arial,helvetica"
size=2>hi 74&deg;<br>lo 55&deg;</font></TD>
...
56</TR>
57</TABLE>
```

**Figure 2. A section of the Corresponding HTML document**

Baltimore, MD	Bag(
TUE	 showers 74 55
WED	 partly cloudy 72 54
THU	 sunny 66 45
FRI	 sunny 74 50
SAT	 sunny 74 54
	)

**Figure 3. The result as Java Objects**

etc. These are the attributes that accept bindings in the query.

- A set of *output types*. The output type may correspond to a tuple, for the relational flavor. We support tuples over the Java basic types, which are Character, String, Number, URL and Image. Alternately we support user defined types, corresponding to a subset of the ODMG standard ODL. The output types are user-defined complex objects that are constructed using list, set, and bag constructors over the basic types and user-defined types.
- An *input-output relationship*. The *input* is a *subset* of the *input attributes*. The *output* is described using a restriction of the OQL projection declaration and can use set, list, or bag constructors and can only refer to the *output types*. We do not support the struct constructor.

Optionally, the *connect* call has methods to return domains and other meta-information; to test queries; and to estimate the cost of the wrapper query. They are discussed in section 7 describing the prototype.

The wrapper provides the following methods to support a query and answer protocol.

- A *submit\_Select* method. This method will accept a *submitTerm* representing a query which has bindings for some input attributes. It returns a true or false, for success or failure, respectively.

The *submitTerm* has two arguments, to specify the output type and a selection predicate. The output type is represented by

( { *set* | *bag* | *list* |  $\epsilon$  } *output type* ).

The predicate is a conjunction of (*input attribute* = *Value*).

- An *iterator* object contains the answer to a submitted query. The iterator object has methods to access its contents.

The wrapper server will create an *iterator* object for each query sent to the wrapper. It follows the standard iterator model [10] and has an open, a getNext and a close method. This iterator provides answer objects (query result) in a threaded pipeline. Each answer object supports methods extracting the values in the fields of the answer object. For the relational wrapper, we provide a JDBC compliant driver as an alternative to the *submit\_Select* method. However, since the source is not a relational DBMS, only those queries which match the capability of the wrapper will be successful.

### 3.2 An Example Web Wrapper interface

We now describe the interface Weather Channel source for a wrapper that returns complex objects. The interface is described in ODL.

- Output Schema

```
interface Forecast {String day; Image forecastImage;
                  String forecastDesc;
                  Int lowTemperature;
                  Int highTemperature}

interface USCity {String name;
                 bag(Forecast) fiveDayForecast}

interface State {String name; bag(USCity) cities}

interface IntCity {String name;
                  bag(Forecast) threeDayForecast}

interface Country {String name; bag(IntCity) cities}
```

- The input attributes

```
interface input_attributes {String StateName;
                          String USCityName;
                          String IntCityName;
                          String CountryName}
```

- Input-output relationship

Input Attributes	Output type
[]	set(State)
[StateName]	State
[StateName]	bag(USCity)
...	...
[IntCityName]	IntCity
[IntCityName]	bag(Forecast)

## 4. Wrapper Capability

The simple capability of a wrapper includes query translation and answer extraction from a single document. It typically requires that a URL representing the HTML document containing the answers (or a script which can provide the answers), must be produced. Then, a simple extractor extracts the relevant data from the corresponding HTML document and produces answer objects.

However, in many cases, this simple scenario is inadequate. For example, when answers of the same type must be extracted from multiple documents, a complex extractor would iteratively call another extractor to extract the objects from each document. In other cases, a complex capability is needed to use the output of another extractor, in constructing another URL, or in constructing an answer object. Finally, when the output format of a document may not always be known a priori, as is common for many Web accessible sources, then a complex extractor would conditionally call other extractors.

Row ID	Input Attributes	Output type	URLConstructor	HTMLExtractor
Row1	[]	bag(CityName)	ConstructorWeather1	EWxtractoreather1
Row2	[StateName]	bag(CityName)	ConstructorWeather2	ExtractorWeather2
Row3	[StateName]	bag(CityURL)	ConstructorWeather2	ExtractorWeather3
Row4	[CityURL]	CityName	ConstructorWeather3	ExtractorWeather4
Row5	[CityURL]	bag(Forecast)	ConstructorWeather3	ExtractorWeather5
Row6	[USCityName]	bag(USCity)	ConstructorWeather4	ExtractorWeather6
Row7	[USCityName]	Title	ConstructorWeather4	ExtractorWeather7
Row8	[USCityName]	bag(CityURL)	ConstructorWeather4	ExtractorWeather8

**Figure 4. Wrapper Capability Table for Weather Channel Web Source**

We implement the simple wrapper capability of query translation and extraction as a syntax directed translation, using a *Wrapper Capability Table*. An example is in Figure 4. Each row of this table can be seen as a production rule. The head of the rule corresponds to the two columns *Input Attributes* and *Output type*, and this is the input-output relationship of the wrapper interface. The actions of the rule are the column *URLConstructor*, and the column *HTMLExtractor*.

**Example 4.1** Consider the following example *submitTerm*:

```
submit_Select (bag(USCity), (USCityName = "Baltimore"))
```

The input attribute `USCityName` matches several rows (Row6, Row7 and Row8) of the wrapper capability table. The output type (`bag(USCity)`) matches Row6. Thus, a query with a binding for this input attribute will invoke the simple wrapper capability represented by Row6. The binding of `'Baltimore'` for `USCityName` will be passed to `ConstructorWeather4`. The contents of the document corresponding to the URL constructed by `ConstructorWeather4` will be streamed to `ExtractorWeather6`. `ExtractorWeather6` will create output objects corresponding to output type (`bag(USCity)`). Thus, the query will return a bag of (US city) weather forecasts for Baltimore.

#### 4.1. A Toolkit for Web Wrappers

In this section, we discuss the toolkit for defining the capability of the Web source, for specifying the functionality of the wrappers. We also describe the graphical interface for defining the source capability and the wrapper functionality.

##### 4.1.1 Toolkit Interface

The wrapper specification toolkit provides graphical interfaces to (1) specify the interface for sources; (2) specify

the wrapper capability table; (3) specify the URL constructor; and (4) specify simple and complex extractors. Each of these specifications, e.g., an extractor specification, can be directly tested using the toolkit. The toolkit will also generate executables, e.g., a wrapper server that can answer queries, or a client applet to accept queries, according to the specified capability of the source. The upper left window of Figure 5 displays some of the toolkit features. Typically the following sequence will be followed to specify the wrapper:

- Specify *Input attributes* and *Input restrictions*.  
Input attributes are defined, and if available, the domains of these attributes are identified. The *input restrictions* are those *input attributes* which must be bound in the *submitTerm*. The upper right window of Figure 5 displays some of the input attributes, for example, `StateName`, and indicated that the domain is enumerated.
- Specify *Output types*.  
Using the set, list and bag constructors, and the built-in types, the *Output type* is specified. The second-row window of Figure 5 shows some output types for this weather source.
- Specify the *Wrapper Capability Table*.  
An association of *Input attributes*, *URLConstructor*, *Output type* and *HTMLExtractor* specifies a row of the *Wrapper Table*. These associations are stored in the catalog and can be directly tested. They are also used in the wrapper generation process. The third-row window of Figure 5 shows some associations of the *Wrapper Capability Table*. The bottom window of Figure 5 shows the *Query Interface* that is generated to test the capability of one row of the *Wrapper Capability Table*. In this example, it is testing the last row of the displayed table. The input attributes are `StateName` and `USCityName`, and the output is `bag(USCity)`.

Source Name: *Weather*  
URL: *http://www.weather.com/twc/home*  

Modify the Output Types  
Modify the Input Attributes  
Modify the Wrapper Capability Table  
Exit Generate Wrapper Server Generate Client Applet

Input Attributes  
CityName:String  
StateName:String domain:Enumerated  
Done  
Add a new Attribute

Output Types  
Forecast: {day:String, forecastImage:URL, forecast:String, lowTemperature:Integer, highTemperature:Integer,}  
City: {name:String, Forecast:Bag, CityName:String,}  
State: {name:String, City:Bag, StateName:String,}  
Done Add a new Type Generate Types

Input Type	URL Constructor	Output Type	HTML Extractor
[]	Processor.URLWeather	Bag(Input.StateName)	Processor.WeatherStateDomain
[CityName]	Processor.URLWeatherCity	Bag(City)	Processor.WeatherForecast
[StateName]	Processor.URLWeatherState	Bag(State.City)	Processor.WeatherStateCities
[CityName, StateName]	Processor.URLWeatherCityState	State.City	Processor.LocalWeatherStateCities

Done New Row

☐ CityName

☐ StateName

Result Type

OK Cancel

Figure 5. Setting Input Attributes; Output types; Wrapper Capability Table; and a Query Interface using the Toolkit

Attribute list Constants Operators Index

"http://www.weather.com/city\_search.pl?city\_destination"+\$USCityName

Generate Load Test Done

Figure 6. Programming the URL constructor

- Specify the *URLConstructor* expression.

A *URLConstructor* expression is used to generate a URL. Its evaluation will produce a string representing the URL. This expression is constructed using a set of built-in operators that manipulate strings. During the specification of a new *URLConstructor* expression, a previously specified *URLConstructor* from the catalog may be selected or an expression can be constructed. If the expression is to be specified, then the graphical interface of Figure 6 is used.

*Attribute List* provides the bindings of attributes from the *Input attributes*. *Constants* provides any constants that are particular to the source. These must have been previously entered into the catalog. *Operators* provides all of the built-in operators and any *Index lookup* which is particular to the source. The Index lookup is also obtained from the catalog. A process similar to domain construction is used to specify the Index lookup. In our prototype Java native methods are also supported.

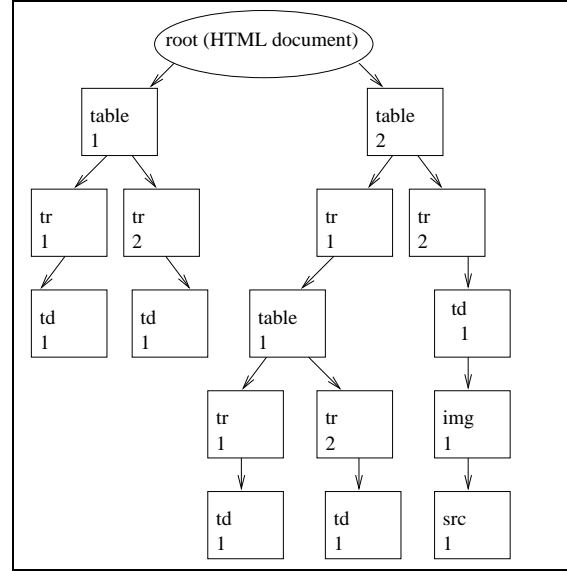
The *URLConstructor* expression specified in Figure 6 provides the correct URL for a query which has a binding on *USCity-Name*. The value for the parameter *city\_destination* is the value of the attribute *USCity-Name*. The function “+” is the string concatenation operator and \$USCityName represents the binding for the attribute *USCityName* in the query. The symbol “?” indicates that the parameter of a CGI-script follows in the usual manner. Details of this interface and language are in [12].

- The final step is specifying either simple or complex extractors. This is described in the next section.

## 5. Functionality of Simple Extractors

A simple extractor processes an HTML document and constructs objects corresponding to some output type specified in the Wrapper Capability Table. The contents of the HTML document and its structure (placement of the HTML elements) are represented as an ordered tree with the HTML element *document* as root. The nodes represent HTML elements or their components, e.g., table, list, img, etc., or table row, list element, source, etc. The arcs of the tree represent the structure (placement) of these elements within the document. The *preorder* enumeration of the ordered tree corresponds to the top-down placement of elements in the document. A data model similar in spirit to the data model of [1], is presented in [12].

**Example 5.1** Consider an *HTML* document whose ordered tree representation is in Figure 7. It comprises of two *HTML*



**Figure 7. An ordered tree representation of an HTML document**

*tables*, each of which is a node. The components of these elements are also nodes, e.g., table rows, (*tr*), and table data (*td*). One *td* element comprises of an *img* element, which is also a node. The integer in each node (rectangle) represents the placement of this object in the HTML document w.r.t. the HTML element that is the *parent node* and w.r.t. the *sibling nodes of the same type*.

A *Qualified path expression Extractor Language, QEL*, is used to specified simple extractors. It provides qualified-path-expressions to specify declarative queries that (a) identify the HTML elements based on their contents or their placement in the document; (b) extract data from the selected HTML elements. We also provide built-in operators to manipulate the extracted data. Finally, we provide a *construct* operator to construct structured objects. A qualified path expression, defined in [9], differs from an OQL like path expression in two ways. First, a predicate can be specified to limit the objects that satisfy the expression. Second, a qualified path expression is able to traverse relationships that are defined to be a collection of objects. Features similar to *QEL* have previously been presented in XSQL [16]. Details of *QEL* in [12].

**Example 5.2** Consider an HTML document with a table whose title is Maryland. Each row of the table corresponds to a city in this state, and the second element (column) of each row contains the name of the city. The following *QEL* expression *MarylandCities*, is used to construct the domain of all cities in Maryland:

*MarylandCities*: Root.child[Name=table & Title="Maryland"].

child.child[Name=td & Occurrence=2].Data

This query first identifies the correct TABLE object using parameters *Name* and *Title*. The expression

Root.child[Name=table & Title="Maryland"].child

is a list of objects that are the rows of this table. The expression

Root.child[...].child.child[Name=td & Occurrence=2]

is a list of objects corresponding to the second column of each row of that table. Next, *.Data* extracts the name of city from each column object. Thus, this query extracts a list of all the city names for the state of Maryland in the same order that they appear in the HTML document.

**Example 5.3** Consider the HTML document in Figure 2 with a table storing the weather forecast for Baltimore. A simple extractor will construct a bag of Forecast objects, where Forecast is defined in section 3.2. We define five *QEL* queries, ListDay, ListForecastDesc, etc. to construct a List(String) corresponding to the values for day, a List(String) for values of forecastDesc, etc. We then use a query BagForecast to construct Forecast objects as follows:

```
Forecast: construct(day: ListDay,
                  forecastDesc: ListForecastDesc,
                  lowTemperature: ListLowTemp,
                  highTemperature: ListHighTemp,
                  forecastImg: ListForecastImg)
```

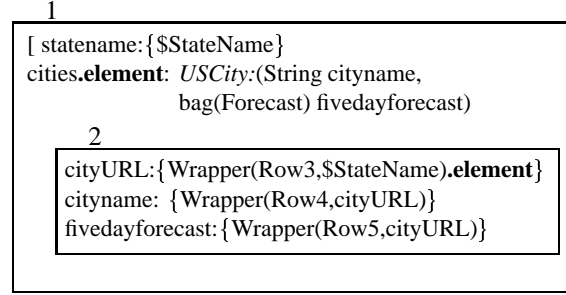
## 6. Specification of Complex Extractors

Simple extractors parse a single HTML document and extract structured objects. In many cases simple extractors are inadequate. We require complex extractors that are able to use the output of other extractors to construct output objects. They are also needed for domain construction where the values of some domain are enumerated within some documents and for index construction. Complex extractors support a conditional operator which test the type of page that was returned. They also support an iterative capability.

*CESEL* is a Complex Extractor Specification Language to specify complex extractors. *CESEL* specifies how a complex object is constructed. A *CESEL* expression is used to assign a value to each attribute of an object. The specification can be nested. The value that is assigned can be an input attribute or the output of an extractor. It can be a singular value or multiple values. The value could be conditionally specified. *CESEL* supports an iterative capability, when some extractor must be applied to multiple documents. Details of *CESEL* in [5].

**Input:** *StateName* , **Output:** *State*

*State:* (String statename, bag(USCity) cities)



1: Object Constructor for *State*

2: Object Constructor for *USCity*

**Figure 8. State Extractor Program in CESL**

### 6.1. An Example of a Complex Extractor

Consider the problem of constructing a state object given a statename as input. A complex extractor to answer this query needs to extract the names of all the cities in the state, and extract the forecast for each of these cities. A *CESEL* program to do this is given in Figure 8. The input is a *StateName* string, and the output is an object of type *State*. An object constructor is used to indicate that the complex extractor will extract the different attributes of the state type, then use them to construct a state object. The object constructor for the *State* object is specified inside the outer box of Figure 8. The definition of the state object is given in parentheses above the box. Inside the object constructor box, rules are given to specify how to extract *statename* and *cities*, the two attributes of the type *State*.

The complex extractor does not need to extract the *statename* value, since the statename is given as input. Therefore, an attribute definition is used to assign this value to *statename*.

To assign a value to *cities* is more complicated. This attribute is a bag of USCity objects, but the complex extractor will construct these objects one at a time. To specify this, the *.element* operator is used.

Since a USCity is a complex object, the program has another object constructor to specify how to extract each attribute of a USCity object. This object constructor is specified inside the inner box in Figure 8. In Figure 4, a Wrapper Call to Row3 returns the URLs of all the cities in one state. Note that the value specified by CityURL is not part of the USCity schema; it is used as an intermediate value by the complex extractor to extract the cityname and forecast of each city. Therefore, Row3 of the capabilities table is to be used internally by the wrapper, and is kept



hidden from the external interface to the wrapper.

The attribute definition *CityURL* tells how to get the URL for a single city. Since the extractor specified in Row3 of the table returns a bag of cityURLs, the complex extractor needs to process the values in this bag one at a time. Therefore, the *.element* operator is used. This indicates that the complex extractor should construct each *USCity* object by iterating over the bag of cityURLs.

To construct each *USCity* object, the complex extractor needs the get the cityname and five day forecast corresponding to each *CityURL*. Row4 of the table gives us a URL-Constructor and HTML extractor pair that extract the name of a city. Row5 of the table extracts the city's forecast.

Note that although Row5 of the table returns a bag of Forecast objects, the *.element* operator is not needed here. This is because a five day forecast is also a bag of Forecast objects, so the iterative capability is not needed in this case. The attribute definition simply assigns the output of Row5 to the *fivedayforecast* attribute.

## 7. Prototype Implementation

The wrapper generation toolkit has been implemented in `jdk1.1.5`. Within the toolkit, each URLConstructor and extractor (simple or complex) can be generated and tested. Each row of the Wrapper Capability Table can also be tested. The bottom level window of Figure 5 shows the *query interface* of a *client applet* that can be generated to test this wrapper. Once the wrapper specification is complete, the information is stored in a catalog so that it can be modified. In addition, a *wrapper server* object may be generated and registered with an RMI server. Extracting meta-information such as domains, statistics, etc., is seamlessly integrated within the toolkit, since these processes are defined in a manner similar to the extraction process.

Each generated *wrapper server* will support a *connect* and a *submit.Selection* method. In response to the *connect* call, the wrapper will return the capabilities of a source. In addition to the input, output types, and input-output relationship, the protocol object will support the following:

- A method *domain()* to obtain the domain of some input attribute(s).
- A method *parser*. This method tests the correctness of queries sent to the wrapper. It uses the input attributes, the input-output relationship, and the domain definitions to test the query.
- A method *meta* to obtain meta-information such as statistics about the distribution of values (cardinality, minimum, maximum, histogram, etc.), of some input attribute for which there is a domain definition.

- A method *cost\_Estimator*. It uses a Multi Dimension Table (MDT) [11] to learn from past queries. It gives back a cost estimation of an input query and a confidence number.

## 8. Related Work

We now consider the variety of wrapper implementations and related projects on wrappers for Web sources. We also consider data models and query languages to handle Web documents and data. An architectural approach that has been advocated for the task of providing uniform access to data residing in heterogeneous data sources (files, databases and legacy servers) is one of *mediators* and *wrappers*; see [24] for a detailed discussion. In this paradigm, the wrapper interface for each data source defines the source schema and capabilities using some common data model and query language. The relational data model and relational operators are used to define the wrapper interface in systems such as DISCO [23, 15], Information Manifold (IM) [4] and Web-Semantics [19]; an object model is used in Garlic [13] and IRODB [8]; and a semi-structured data model, OEM, is used in TSIMMIS [22, 21].

A variety of wrapper implementations are currently available, ranging from the powerful and widely accepted JDBC [18] drivers to wrappers specific for DISCO, Garlic, IRODB, WebSemantics, etc. The generic WebSemantics JDBC wrapper [19] is capable of connecting to any data source that is a DBMS, for which a JDBC driver is available. In contrast, our Web wrappers are wrappers tailored to the capability of a specific source.

Related projects on Web wrappers are described in [2, 14]. They are passive wrappers that do not answer queries on the sources; instead they extract data statically from pages. A language for specifying an extractor for semi-structured OEM objects is described in [14]. The extraction language is based on the syntactic recognition of HTML structures. As a result, it is more sensitive to syntactic changes in the HTML document. Since we support semantic recognition of HTML documents using built-in operators, our extractors are more robust. The wrapper retrieves all (relevant) data in OEM format and executes the wrapper query against these structures. [2] proposes a tool that generates extractors for Web sources where the data follows a hierarchical structure. Syntactical recognition of HTML structures, as well as heuristics about HTML syntactic structures, are used to automatically generate a program to extract the data. They are limited in that the extracted data must exactly reflect the structure within the document. In contrast, we extract structured objects of some fixed output type. Neither project [14, 2] provides a toolkit for specifying capabilities for sources and defining the wrapper functionality. There has been much work recently on

models and query languages for semi-structured Web data. [6, 20, 3, 7] propose data models to represent unstructured or semi-structured data. They also provide languages to query these schemas. These projects provide data models and query languages, but do not consider supporting the functionality of a wrapper.

A recently emerged proposal for a standard for exchange of types and structured data, XML-Data [17], if widely accepted, can be used as an alternative for the JDBC protocol, for object representation. However, XML-Data does not support a query and answer protocol. The acceptance of the XML-Data standard would simplify the process of extracting data from Web sources. However, the need to generate customizable wrappers to provide a standard interface to application programs will remain.

In future work, we plan to implement Web wrappers for XML-Data sources. We also plan to extend the capability of the wrappers beyond the extraction task. Instead of extracting an answer from a page, a wrapper may otherwise test the contents of the page, or test when the page was last updated, etc. With the added capability, the Web wrapper could support the functionality of a programmable robot, whose crawling behavior and actions could be programmed using the capabilities of the Web wrappers described in this paper.

## 9 Acknowledgments

We acknowledge the programming support of Meera Mahabala and Tao Zhan on this prototype.

## References

- [1] G. Arocena and A. Mendelzon. Weboql: Restructuring documents, databases, and webs. *Proceedings ICDE*, 1998.
- [2] N. Ashish and C. Knoblock. Wrapper generation for semi-structured internet sources. *Proceedings of the Workshop on Management of Semi-structured Data*, 1997.
- [3] P. Atzeni, G. Mecca, and P. Merialdo. Semistructured and structured data in the web: Going back and forth. *Proceedings of the Workshop on Management of Semi-structured Data*, 1997.
- [4] A.Y. Levy et al. Querying heterogeneous information sources using source descriptions. *VLDB*, 1996.
- [5] L. Bright. Cesi: A tool for handling complex queries to web sources. *Technical Report, Department of Computer Science, University of Maryland*, 1998.
- [6] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. *Proceedings of ICDT*, 1997.
- [7] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language and processor for a web-site management system. *Proceedings of the Workshop on Management of Semi-structured Data*, 1997.
- [8] G. Gardarin et al. Iro-db : A distributed system federating object and relational databases. *VLDB*, 1994.
- [9] G. Gardarin et al. Cost-based selection of path expression in object-oriented databases. *VLDB*, 1996.
- [10] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), June 1993.
- [11] J. Gruser, L. Raschid, and M. Vidal. Learning from query feedback to estimate performance of web sources. *Technical Report, UMIACS, University of Maryland*, 1997.
- [12] J. Gruser, L. Raschid, M. Vidal, and L. Bright. A wrapper generation toolkit to specify and construct wrappers for web accessible data. *Technical Report, UMIACS, University of Maryland*, 1998.
- [13] L. Haas. Optimizing queries across diverse data sources. *VLDB*, 1997.
- [14] J. Hammer, H. Garcia-Molina, R. Cho, R. Aranha, and A. Crespo. Extracting semistructured information from the web. *Proceedings of the Workshop on Management of Semi-structured Data*, 1997.
- [15] O. Kapitskaia et al. Dealing with discrepancies in wrapper functionality. *Technical Report INRIA*, 1997.
- [16] M. Kifer et al. Querying object-oriented databases. *ACM SIGMOD*, 1992.
- [17] A. Layman et al. Xml data. <http://www.w3.org>.
- [18] S. Microsystems. The jdbc database access api. <http://splash.javasoft.com/jdbc>.
- [19] G. Mihaila, L. Raschid, and A. Tomassic. Equal time for data on the internet with websemantics. *EDBT*, 1998.
- [20] S. Nestorov, S. Abiteboul, and R. Motwani. Inferring structure in semistructured data. *Proceedings of the Workshop on Management of Semi-structured Data*, 1997.
- [21] Y. Papakonstantinou et al. Capabilities-based query rewriting in mediator systems. *Proceedings of the Intl. Conference on Parallel and Distributed Information Systems*, 1996.
- [22] Y. Papakonstantinou et al. A query translation scheme for rapid implementation of wrappers. *Proceedings of the Intl. Conference on DOOD*, 1996.
- [23] A. Tomasic et al. Scaling heterogeneous databases and the design of disco. *Proceedings of the Intl. Conf. on Distributed Computing Systems*, 1996.
- [24] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, March 1992.