Semistructured Data: The TSIMMIS Experience

Joachim Hammer, Jason McHugh, and Hector Garcia-Molina

Department of Computer Science
Stanford University
Stanford, CA 94305–9040
U.S.A.

{joachim,mchughj,hector}@db.stanford.edu
http://www-db.stanford.edu

Abstract

In this paper we discuss the management of semi-structured data, i.e., data that has irregular or dynamically changing structure. We describe components of the Stanford TSIMMIS Project that help extract semi-structured data from Web pages, that allow the storage and querying of semi-structured data, and that allow its browsing through the World Wide Web. A prototype implementation of the TSIMMIS system as described here is currently installed and running in the database group testbed.

1 Introduction

At a recent workshop on management of semistructured data [15], the workshop attendants defined *semistructured* data as data that does not have a regular and static structure like data found in a relational database but whose schema is dynamic and may contain missing data or types. For example, if we look at weather forecasts on the Web, the "fields" and their structure may differ across sites. Even at a single site, some forecasts may be missing information, or may have extra information depending on the geographical location of the affected region (e.g., cities in the Rocky Mountains usually include ski reports in the winter months whereas forecast for tropical resorts do not). However, semistructured data is not just limited to the World Wide Web (WWW), but is also found in many other interesting sources including file systems, news wires, electronic mail systems, etc. just to name a few. In addition to occurring natively in the above-mentioned classes of sources, semistructured data is often a "by-product" of the integration process when multiple heterogeneous schemas are involved. In those cases, semistructured (rather than "fully structured") data arises because the integrated objects may be based on complimentary, sometimes conflicting, and often dynamic information from multiple sources, forcing the integrator to filter, merge, or omit certain fields when performing the integration.

The goal of the TSIMMIS project at Stanford [4, 7, 11, 13] is to provide integrated access to a wide variety of heterogeneous data sources (e.g., databases, object stores, knowledge bases, digital libraries) including sources containing semistructured data (e.g., WWW, file system). In this paper, we present the TSIMMIS approach to managing semistructured data. In particular, we discuss three critical aspects semistructured data management: (1) extracting the intended content from its native source (how to get it?), (2) reading the extracted data (how to query it?), and (3), exploring the result in a graphical, easy-to-understand manner (how to browse it?). In TSIMMIS we have developed components that address all of the above issues and together provide an integrated solution to the problem of managing semistructured data. Several other recent projects have similar goals (e.g., LORE [10], Garlic [3], Information Manifold [9], Rufus [14]), but we do not survey them here.

2 Representing Semistructured Data in TSIMMIS

For the TSIMMIS project we have adopted a simple *self-describing* (or *tagged*) object model. Similar models have been in use for years; we call our version the *Object Exchange Model*, or OEM [4]. OEMis a flexible model that is particularly well suited for representing semistructured data. Data represented in OEM constitutes a graph, with a unique root object at the top and zero or more nested subobjects. The fundamental idea is that all objects, and their subobjects, have *labels* that describe their meaning. For example, the following object represents a Fahrenheit temperature of 80 degrees:

temp-in-Fahrenheit, *int*, 80

Here, the string "temp-in-Fahrenheit" is a human-readable label, "int" indicates an integer value, and "80" is the value itself. If we wish to represent a *complex* object, then each component of the object has its own label. For example, an object representing a set of two temperatures may look like:

set-of-temps, *complex*, {
  temp-in-Fahrenheit, *int*, 80
  temp-in-Celsius, *int*, 20 }

OEM is very simple, while providing the expressive power and flexibility needed for representing semistructured data from a wide range of heterogeneous sources. Our primary reason for choosing a simple model is to be able to accommodate a wide variety of external data models and to facilitate integration. As pointed out in [2], a simple model such as OEM has an advantage over complex models when used for representing and integrating heterogeneous data, since the

```
1 <HTML>
2 <HEAD>
3 <TITLE>INTELLICAST: europe weather</TITLE>
4 <A NAME="europe"></A>
5 <TABLE BORDER=0 CELLPADDING=0 CELLSPACING=0 WIDTH=509>
6 <TR>
7 <TD colspan=11><I>Click on a city for forecasts</I><BR></TD>
8 </TR>
9 <TR>
10 <TD colspan=11><I>temperatures in degrees celsius</I><BR></TD>
11 </TR>
12 <TR>
13 <TD colspan=11><HR NOSHADE SIZE=6 WIDTH=509></TD>
14 </TR>
15 </TABLE>
16 <TABLE CELLSPACING=0 CELLPADDING=0 WIDTH=514>
17   <TR ALIGN=left>
18     <TH COLSPAN=2><BR></TH>
19     <TH COLSPAN=2><I>Tue, May 06, 1997</I></TH>
20     <TH COLSPAN=2><I>Wed, May 07, 1997</I></TH>
21   </TR>
22   <TR ALIGN=left>
23     <TH><I>country</I></TH>
24     <TH><I>city</I></TH>
25     <TH><I>forecast</I></TH>
26     <TH><I>hi/lo</I></TH>
27     <TH><I>forecast</I></TH>
28     <TH><I>hi/lo</I></TH>
29   </TR>
30   <TR ALIGN=left>
31     <TD>Finland</TD>
32     <TD><A HREF=weather/hel/>Helsinki</A></TD>
33     <TD>rain</TD>
34     <TD>16/4</TD>
35     <TD>shwrs</TD>
36     <TD>10/4</TD>
37   </TR>
38   <TR ALIGN=left>
39     <TD>France</TD>
40     <TD><A HREF=weather/bor/>Bordeaux</A></TD>
41     <TD>fair</TD>
42     <TD>26/13</TD>
43     <TD>fair</TD>
44     <TD>29/15</TD>
45   </TR>
      .
      .
   </TABLE>
      .
```

Figure 1: A section of the HTML source file

operations to transform and merge data will be correspondingly simpler. Meanwhile a simple model can still be very powerful: advanced features can be "emulated" when they are necessary (e.g., subclass/superclass relationships, inheritance, etc.). For additional information on OEM, please refer to [12].

3 Extracting Data

Continuing with our weather example, let us assume that we have an application that needs to process weather data, such as temperature and forecast, for a given city. As one of its information sources, we want to use a Web site called Intellicast, which reports daily weather data for most major cities across the world. Since this site cannot be queried directly from within another application (e.g., "What is the forecast for Helsinki for May 7, 1997?") we first have to extract the contents of the weather table from the underlying HTML page[1] which is displayed in Figure 1.

3.1 The Extraction Process

Our *configurable extraction* program parses this HTML page based on the specification file shown in Figure 2. The specification file consists of a sequence of *commands*, each defining one extraction step. Each command is of the form

$$[\ variables,\ source,\ pattern\ ]$$

where *source* specifies the input text to be considered, *pattern* tells us how to find the text of interest within the source, and *variables* are one or more extractor variables that will hold the extracted results. The text in variables can be used as input for subsequent commands. (If a variable contains an extracted URL, we can also specify that the URL be followed, and that the linked page be used as further input.)

[1] The line numbers shown on the left-hand side of this and the next figures are not part of the content but have been added to simplify the following discussion.

After the last command is executed, some subset of the variables will hold the data of interest. Later we describe how the contents of these variables are packaged into an OEM object.

Looking at Figure 2, we see that the list of commands is placed within the outermost brackets '[' and ']', and each command is also delimited by brackets. The extraction process in this example is performed by five commands. The initial command (lines 1-4) fetches the contents of the source file whose URL is given in line 2 into the variable called *root*. The '#' character in line 3 means that everything (in this case the contents of the entire file) is to be extracted. After the file has been fetched and its contents are read into *root*, the extractor will filter out unwanted data such as the HTML markup commands and extra text with the remaining four commands.

The second command (lines 5-8) specifies that the result of applying the pattern in line 7 to the source variable *root* is to be stored in a new variable called *_temperature*. The pattern can be interpreted as follows: discard everything until the first occurrence of the token $\langle /TR \rangle$ ('*' means discard) in the second table definition and save the data that is stored between $\langle /TR \rangle$ and $\langle /TABLE \rangle$ ('#' means save). The two $\langle TABLE$ tokens between the '*' are used as navigational help to identify the correct $\langle /TR \rangle$ token since there is no way of specifying a numbered occurrence of a token (i.e., discard everything until the third occurrence of $\langle /TR \rangle$). After this step, the variable *_temperature* contains the information that is stored in lines 22 and higher in the source file in Figure 1 (up to but not including the subsequent $\langle /TABLE \rangle$ token which indicates the end of the temperature table). The underscore at the beginning of the name *_temperature* indicates that this is a temporary variable; its contents will not be included in the resulting OEM object.

The third command (lines 9-12) instructs the extractor to split the contents of the temperature variable into "chunks" of text, using the string $\langle TRALIGN = left \rangle$ (lines 22, 30, 38, etc. in Figure 1) as the "chunk" delimiter. Note, each "chunk" represents one row in the temperature table. The result of each split is stored in a temporary variable called *_citytemp*. The split operator can only be applied if the input is made up of equally structured pieces with a clearly defined delimiter separating the individual pieces. If one thinks of extractor variables as lists (up until now each list had only one member) then the result of the split operator can be viewed as a new list with as many members as there are rows in the temperature table. Thus from now on, when we apply a pattern to a variable, we really mean applying the pattern to *each* member of the variable, much like the `apply` operator in Lisp.

In command 4 (lines 13-16), the extractor copies the contents of each cell of the temporary array into the array *city_temp* starting with the second cell from the beginning. The first integer in the instruction $\_citytemp[1:0]$ indicates the beginning of the copying (since the array index starts at 0, 1 refers to the second cell), the second integer indicates the last cell to be included (counting from the end of the array). As a result, we have excluded the first row of the table which contains the individual column headings. Note, that we could have also filtered out the unwanted row in the second command by specifying an additional $*\langle /TR \rangle$ condition before the '#' in line 7 of Figure 2. The final command (lines 17-20) extracts the individual values from each cell in the *city_temp* array and assigns them into the variables listed in line 17 (*country*, *c_url*, *city*, etc.).

```
[["root",
  "get('http://www.intellicast.com/weather/europe/')",
  "#"
 ],
 ["_temperatures",
  "root",
  "*<TABLE*<TABLE*</TR>#</TABLE>*"
 ],
 ["_citytemp",
  "split(temperatures,'<TR ALIGN=left>')",
  "#"
 ],
 ["city_temp",
  "_citytemp[1:0]",
  "#"
 ],
["country,c_url,city,w_tody,hgh_tody,low_today,w_tomorrow,hgh_tomorrow,low_tomorrow",
  "city_temp",
  "*<TD>#</TD>*HREF=#>#</A>*<TD>#</TD>*<TD>#/#</TD>*<TD>#</TD>*<TD>#/#*"
 ]]
```

Figure 2: A sample extractor specification file

```
root     complex   {
         city_temp complex   {
                   country        string    "Finland"
                   city_url       url       http://www...
                   city           string    "Helsinki"
                   weather_today            string    "rain"
                   high_today               string    "16"
                   low_today                string    "4"
                   weather_tom              string    "shwrs"
                   high_tomorrow            string    "10"
                   low_tomorrow             string    "4"
         }
         city_temp complex   {
                   country        string    "France"
                   city_url       url       http://www...
                   city           string    "Bordeaux"
                                  ...
         }
         ...
}
```

Figure 3: The extracted information in OEM format

After the five commands have been executed, the variables hold the data of interest. This data is packaged into an OEM object, shown in Figure 3, with a structure that follows the extraction process. Notice that this sample object reflects the structure of our extractor specification file. That is, the root object of the OEM answer will have a label root because this was the first extracted variable. This object will have children objects with label *city_temp* and so on. Notice that the variables *_temperature* and *_citytemp* do not appear in the final result because they are declared as temporary variables.

3.2 Additional Capabilities

In addition to the basic capabilities described in our example, the extractor has components for automatic handling of HTML tables, for conditional parsing, and other services. The extractor can also follow URLs in the process, extracting data from multiple Web pages into a single OEM object. Overall, we believe the extractor provides natural facilities for extracting data, as well as for structuring it in different ways into OEM objects. For more details on the extractor, please refer to [8].

4 Querying Semistructured Data

In this section we introduce the LOREL query language, primarily through examples. LOREL is an extension of OQL and a full specification can be found in [1]. Here we highlight those features of the language that have an impact on the novel aspects of the system—features designed specifically for handling semistructured data. Many other useful features of LOREL (some inherited from OQL and others not) that are more standard will not be covered.

4.1 Simple LORE Examples

Our first example query introduces the basic building block of LOREL: the simple path expression, which is a name followed by a sequence of labels. For example, *Root.City.Location* is a simple path expression. Its semantics consists of the set of objects that can be reached starting with the *Root* object, following an edge to objects labeled *City*, then following an edge to objects labeled *Location*. Range variables can be assigned to path expressions, e.g., "*Root.City.LocationX*" specifies that X ranges over the set of locations.

Continuing with our European weather example, the following example query retrieves the locations of all cities located in England when evaluated over the sample OEM database shown in Figure 4.

Query 4.1 (LOREL)

```
 SELECT Root.City.Location
WHERE Root.City.Country = "England"
```

At a high level, the query execution engine will find all objects which satisfy the path *Root.City.Location* and for each of these will check whether the where clause is satisfied. The result of Query 4.1 is shown here:

> **answer** *complex* {
>   **location** *string* "Southern"
>   **location** *complex* {
>   **longitude** *float* -0.167
>   **latitude** *float* 51.5 }
>   }

The database over which this query is evaluated presents a number of irregularities, as discussed earlier. A guiding principle in LOREL is that, to write a query, one should not

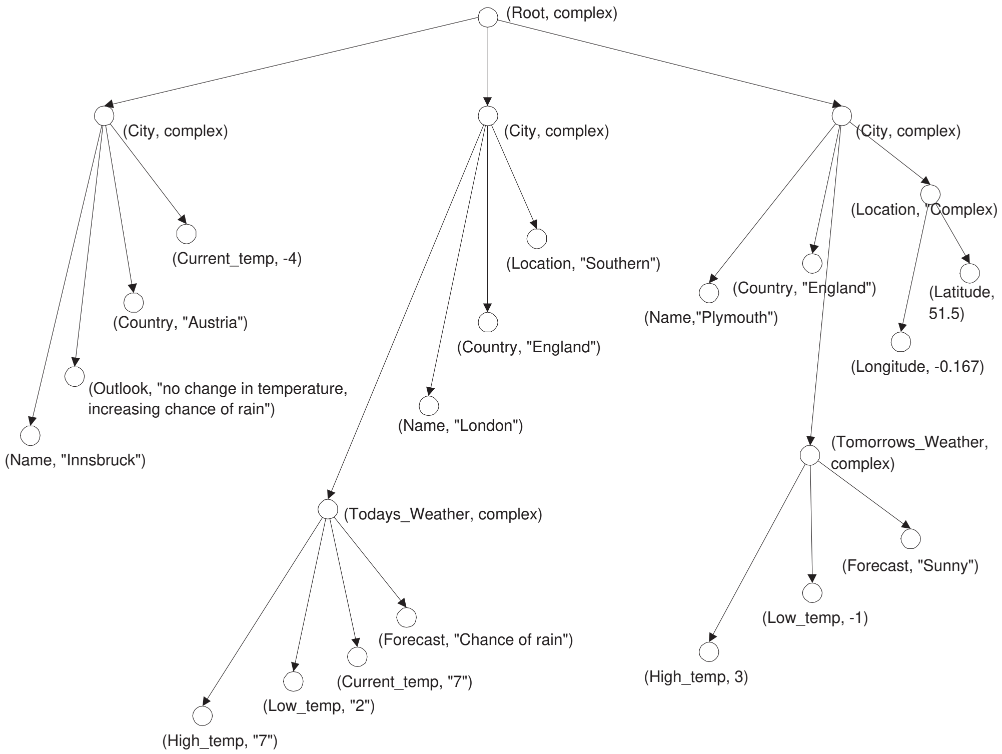

Figure 4: Sample OEM database

have to worry about such irregularities or know the *precise* structure of objects (e.g., the structure of location objects), nor should one have to bother with precise types. This query will not yield a run-time error if a *Location* object has a string value or is complex, or if *Country* objects are single-valued, set-valued, or even absent for some cities. Indeed, the above query will succeed no matter what the actual structure of the database is, and will return an appropriate answer. Of course, this query was written with some obvious knowledge of how the graph is laid out within our database. In Sec. 4.2 we discuss how an end user can discover the structure of the database.

Value comparisons are made after two objects have been coerced into comparable types. That is, if two objects do not have the same type then attempts will be made to coerce the values into comparable types before applying the comparison operator. Any types which cannot be coerced for comparison will not return type errors, but will simply evaluate to false. This reinforces our underlying principle that LOREL does not require precise knowledge of the data and is most useful when dealing with semistructured data.

The system will in fact translate all LOREL queries into OQL-like queries for evaluation. This is done for two reasons: first, LOREL is based on OQL and thus OQL gives us well defined semantics for our queries, and second it allows a user familiar with OQL to directly enter an OQL query to be evaluated over the semistructured data. In some sense, LOREL can be viewed as shorthand for OQL, however LOREL also introduces *generalized path expressions* not present within OQL. Generalized path expressions offer a richer form of "declarative navigation" in OEM databases than simple path expressions. Intuitively, the user loosely specifies a desired pattern of labels in the database: one can specify patterns for paths (to match sequences of labels), patterns for labels (to match sequences of characters), and patterns for atomic values. A combination of these three forms of pattern matching is illustrated in the following example:

Query 4.2 (LOREL)

```
 SELECT Root.City.Name
WHERE Root.City(.%weather)?(.Forecast|.Outlook)
        grep "rain"
```

Here the expression *%weather* is a label pattern that matches all labels ending with the string *weather* (e.g., *weather*, *Todays_weather*, or *Tomorrows_weather*). For path patterns, the symbol "|" indicates disjunction between two labels, and the symbol "?" is applied to the parenthesized expression to the left and indicates that the label pattern is optional. The complete syntax is based on regular expressions, along with syntactic wildcards such as "#", which matches any path of length 0 or more. Finally, *grep*"*rain*" specifies that the data value should contain within it the string "*rain*. The *grep* operator is similar to the Unix `grep` command. We also support *like*, based loosely on the SQL `like`, and *soundex* for phonetic matching. In English this query is asking for the names of all cities where the forecast (or outlook) of the weather contains the word "rain". The result of Query 4.2 applied over the database in Figure 4 looks like this:

answer *complex* {
  name *string* "Innsbruck"
  name *string* "London"
  }

During preprocessing, simple path expressions are eliminated by rewriting the query to use variables, as demon-

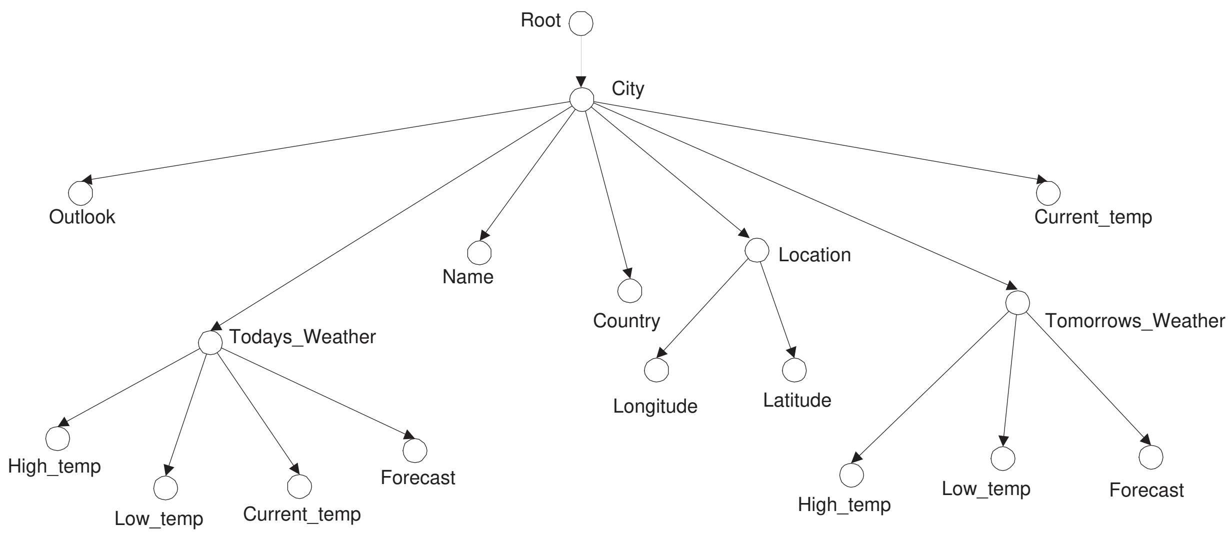

Figure 5: Sample DataGuide

strated in our first example. It is not possible to do so with general path expressions, which require a run-time mechanism. Indeed, note that if the database contains cycles, then a general path expression may match an infinite number of paths in the data. When trying to match a general path expression against the database, we match through a cycle at most once, which appears to be a reasonable simplification in practice.

We conclude with an example that illustrates advanced features of the language. The following query illustrates subqueries and constructed results. For every city in the database that satisfies the (bottom) where clause, we will select out the name of the city along with the current temperature, but only if the current temperature satisfies the `WHERE` clause.

Query 4.3 (Lorel)

```
 SELECT C.Name,
    ( SELECT X
    FROM C.Todays_weather.current_temp X
    WHERE X < 10 )
FROM Root.City C
WHERE C.Country = "England"
```

The result is shown below. Notice that each city which provides a binding for the C variable and satisfies the where clause appears within the answer. Of particular interest is the fact that *Plymouth* does not have a *current_temp* field within the answer. This is filtered out as a result of the subquery appearing within the *SELECT* clause. Specifically, the *Plymouth* object does not have a subobject labeled *Todays_weather*.

answer *complex* {
  city *complex* {
    name *string* "London"
    current_temp *integer* 7 }
  city *complex* {
    Name *string* "Plymouth" }
  }

4.2 Query Formulation with the DataGuide

Since our data does not have an explicit schema, query formulation and query optimization are particularly challenging. Without some knowledge of the structure of the underlying database, writing a meaningful Lorel query may be difficult, even when using general path expressions. One may manually browse a database to learn more about its structure, but this approach is unreasonable for very large databases. Further, without information about the structure of the database, the query processor may be forced to perform more work than necessary. For example, consider Query 4.1 that finds the locations of all cities whose country is England. Even if no cities have a country subobject, the execution engine would still needlessly examine every city in the database.

A *DataGuide* is a concise and accurate summary of the structure of an Oem database, stored itself as an Oem object. Each possible path expression of a database is encoded exactly once in the DataGuide, and the DataGuide has no path expressions that do not exist in the database. As an example Figure 5 shows a DataGuide for the sample database shown in Figure 4. (Note that atomic values are usually not stored within the leaf nodes of the DataGuide since it is primarily concerned with the structure of the database.) In typical situations, the DataGuide is significantly smaller than the original database. A DataGuide plays a role similar to metadata in traditional database systems. The DataGuide may be queried or browsed, enabling user interfaces or client applications to examine the structure of the database. Assuming the role of the missing schema, the DataGuide can also guide the query processor. Of course, in relational or object-oriented systems the schema is explicitly created before any data is loaded; here, DataGuides are dynamically generated and maintained over all or part of an existing database.

In [5], formal definitions for DataGuides are provided as well as algorithms to build and incrementally maintain DataGuides that support annotations. Also given is a discussion of how DataGuides aid query formulation in practice and their use for query optimization.

5 Browsing OEM Results through MOBIE

The main idea behind our browsing tool centers around the need for displaying semistructured objects in a way that makes it easy for the user to grasp their structure and explore their contents when viewing the result of a TSIMMIS query. OEM results are typically irregular in structure and nested, containing a top-level (root) object and zero or more subobjects (sometimes referred to as children). Each subobject may itself be a nested object. In general, nested objects are structured like trees (or graphs if we allow cycles). Anybody who has worked with nested objects before can attest to the fact it becomes increasingly difficult to understand the contents of a nested object the more its structure increases in complexity (i.e., the larger the number of subobjects and the deeper the level of nesting).

For this reason, we have built a system that transforms OEM results into a "web" of hyperlinked documents that can be viewed using any WWW browser. An object that is selected for viewing is formatted as an HTML document. If the object is a complex object, the document may also include hyperlinks pointing to some or all of the object's substructure depending on the user's preferences. If the object is atomic, it will be displayed by itself. In addition, each document always contains a link to the parent object, unless the selected object is the root of the entire structure. The main contribution of our system is that it gives the user the option to decide which information is to be displayed, how much of the chosen information he or she wants to see, and when. Information is presented one screen at a time, allowing the user to browse complex objects, which may be too large to view all at once, in a "cafeteria-style " (pick-and-choose) fashion. This approach to browsing nested objects is analogous to how one uses the table of contents to explore the individual chapters of a book.

An important part of the functionality of our browser focuses on the layout of information on individual pages. Since this is a process that depends heavily on each user's individual preferences as well as the data that is being displayed, we have paid careful attention to design a system that is flexible enough so that it can be tailored to satisfy many different needs. Our goal was to provide users with choices as to how information is to be displayed: from the overall layout of a screen down to the format of an individual object. Initially, the system uses default settings that maximize the amount of information that can be displayed within the given real-estate of the window. The result can then be improved upon by changing the values of *session variables*, which control the document layout, the level of nesting per screen, the number of subobjects per level, etc. By default, session variables control the formatting for the complete object hierarchy. However, by using the label names that refer to a particular object in the hierarchy, the scope of session variables can be limited: from the entire hierarchy, to a specific substructure, to one object. Although customization of the object display may be time consuming in certain cases, the state of the session variables can be saved on a per-user basis and re-used during subsequent sessions.

We have implemented a fully functional prototype system called MOBIE (Multimedia OBject and Information Explorer), which currently provides the graphical interface to TSIMMIS data sources. However, MOBIE is not limited to browsing only data from TSIMMIS but can be tailored for displaying and formatting structured information from any object-based database/footnoteOne can either use a translator for converting data into OEM or modify our algorithm

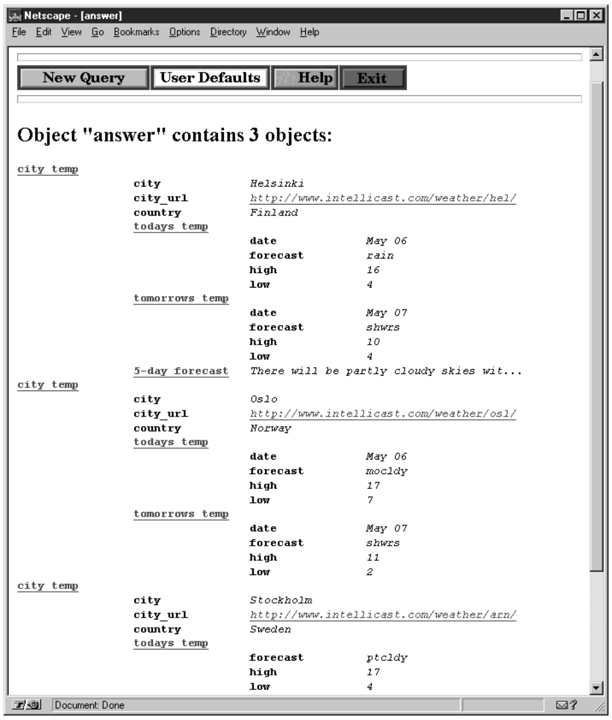

Figure 6: Result to Query 5.1

to work with other object-based data models.. Since a complete description of our browser is beyond the scope of this paper, we invite the user to obtain the details from [6]. Instead, we will briefly demonstrate some of MOBIE's functionality using screen snapshots from a sample interaction with a TSIMMIS wrapper connected to the Intellicast weather source (via the above mentioned Web extractor). We start our description when the result is returned from the database, omitting such details as how to connect to the database server, transmission of the query and its results, etc. When displaying data, we use the following conventions. Object labels are displayed in **bold**, object values are *italicized*. Underlining indicates the existence of a hyperlink.

5.1 Sample Screen Snapshot

Let us assume that we have submitted the following LOREL query asking for all cities in Europe where tomorrow's forecast calls for showers:

Query 5.1 (LOREL)

```
 SELECT city_temp
FROM intellicast:i
WHERE i.city_temp.tomorrows_temp.forecast = "shwrs"
```

Let us also assume that the answer to this query consists of three cities that are displayed together under one root object, labeled **answer**. Figure 6 shows the **answer** object as it is displayed in MOBIE. Each object labeled **city_temp** is a complex object exhibiting additional substructure underneath: the objects labeled **city**, **city_url**, **country**, **todays_temp**, and **tomorrows_temp**. Note that the first subobject (the city of Helsinki) has one additional subobject labeled **5-day_forecast** that is not present in the other results. The **city**, **city_url**, and **country** subobjects are *atomic* meaning they contain no further substructure. In those cases, the value of the object is displayed. (If there

Figure 7: Query result—subobject "Helsinki"

is not enough room for the value, a hyperlink is provided.) The **todays_temp** and **city_url** subobjects on the other hand, are *complex* objects that contain additional subobjects: **forecast**, **high**, **low**, and **date**. Labels belonging to complex objects are underlined meaning that a hyperlink exists that will take the user to the document containing only those subobjects. (Those subobjects are displayed in a similar fashion.) Also note, the value of the **city_url** subobjects is a standard URL that is part of the answer and has been activated by Mobie for loading.

5.2 Formatting Options

As mentioned before, the user can control the formatting of objects through various control parameters. These parameters are called *session variables* and can be accessed from the **User Defaults** menu. Formatting options fall into two categories: "Global Settings", which apply to the whole object structure, and "Label-Based Settings" for which the scope can be specified based on object labels. (See [6] for details and other options.)

The following parameters are available for controlling global settings:

- *Maximum levels of sub-objects* controls the number of visible levels of subobjects for each object that is displayed.
- *Sub-object indentation* controls the amount of indentation used for subobjects.

The following parameters are available for controlling label-based settings:

- *Layout* controls the overall "look-and-feel" of the output when it gets displayed in the browser window. The two options currently available are *table* and *list* layout.
- *Number of displayed sub-objects* controls the number of subobjects that are displayed on a screen.
- *Label size* and *value size* control the length of labels and values respectively.

Using these options one can format data in the way that best suits it. For example, Figure 7 shows some data formatted as a table. Labels are shown on the left side. If the subobject is an atomic object (e.g., the subobjects labeled **city**, **city_url**, **country**, and **5-day_forecast**) the first column starting form the left will contain the subobject's value. If the subobject is a complex object, e.g., the subobjects labeled **todays_temp** and **tomorrows_temp**, the first column will be empty, and subsequent columns will contain the values of its immediate subobjects. In the latter case, the column headings are the labels of the lower-level subobjects. Note, if there are several complex subobjects with different substructure, the table will display the union of all possible headings.

As mentioned before, label-based settings apply to objects. In order to format an object, a formatting choice associated with its label must be defined. Thus it is possible, for example, to display three or more levels of nesting for the root object, and then reduce the number of visible levels to just one when viewing its subobjects. As another example, one can display the part of a result that contains numerical values as a table but leave the part that is mostly textual in list format.

6 Conclusion

In this paper we have presented an overview of the Tsimmis approach to accessing and managing semistructured data. In particular, we have described how semistructured data can be obtained from Web pages, how it can be manipulated in a database system, and how it can be browsed. We believe that semistructured data exists in many applications, and flexible tools like the ones we have described can be very helpful for managing it.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Journal of Digital Libraries*, 1(1), November 1996.

[2] C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[3] M.J. Carey, L.M. Haas, P.M. Schwarz, M. Arya, W.F. Cody, R. Fagin, A. Flickner, A.W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E.L. Wimmers. Towards heterogeneous multimedia information systems: the Garlic approach. In *In Proceedings of the Sixth International Conference on Data Engineering*, pages 123–130, Los Angeles, California, February 1995.

[4] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The tsimmis project: Integration of heterogeneous information sources. In *Proceedings of the Tenth Anniversary Meeting*, pages 7–18. Information Processing Society of Japan, Tokyo, Japan, October 1994.

[5] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the Twenty-Third International Conference on Very Large Database*, Athens, Greece, September 1997.

[6] J. Hammer, R. Aranha, and K. Ireland. Browsing object databases through the Web. Technical report, Department of Computer Science, Stanford, California, February 1997.

[7] J. Hammer, M. Breunig, H. Garcia-Molina, S. Nestorov, V. Vassalos, and R. Yerneni. Template-based wrappers in the TSIMMIS system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 532, Tucson, Arizona, May 1997. Association of Computing Machinery.

[8] J. Hammer, H. Garcia-Molina, Y. Cho, R. Aranha, and A. Crespo. Extracting semistructured information from the Web. In *Proceedings of the First Workshop on Management of Semistructured Data*, pages 18–25, Tucson, Arizona, May 1997.

[9] T. Kirk, A. Levy, J. Sagiv, and D. Srivastava. The information manifold. Technical report, AT&T Bell Laboratories, 1995.

[10] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. LORE: A database management system for semistructured data. *SIGMOD Record*, 26(3):50–61, 1997.

[11] Y. Papakonstantinou, H. Garcia-Molina, and S. Abiteboul. Object fusion in mediator systems. In *Proceedings of the International Conference on Very Large Databases*, pages 234–245, Bombay, India, September 1996.

[12] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260. Computer Society of the IEEE, Taipei, Taiwan, March 1995.

[13] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *International Conference on Deductive and Object-Oriented Databases*, pages 97–107, August 1995.

[14] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, and J. Thomas. The RUFUS system: Information organization for semi-structured data. In *Proceedings of the International Conference on Very Large Databases*, pages 97–107, Dublin, Ireland, August 1993.

[15] D. Suciu. Proceedings of the workshop on managemnt of semistructured data. Tucson, Arizona, May 1997. Los Angeles. (Workshop papers are available electronically at `http://www.research.att.com/~suciu/workshop-papers.html`.)