Template-Based Wrappers in the TSIMMIS System*

Joachim Hammer, Hector Garcia-Molina, Svetlozar Nestorov, Ramana Yerneni, Marcus Breunig, Vasilis Vassalos

Department of Computer Science
Stanford University
Stanford, CA 94305-9040

E-mail: {joachim,hector,evtimov,yerneni,vassalos}@db.stanford.edu
http://www-db.stanford.edu/tsimmis

1 Overview

In order to access information from a variety of heterogeneous information sources, one has to be able to translate queries and data from one data model into another. This functionality is provided by so-called (source) *wrappers* [4,8] which convert queries into one or more commands/queries understandable by the underlying source and transform the native results into a format understood by the application. As part of the TSIMMIS project [1,6] we have developed hard-coded wrappers for a variety of sources (e.g., Sybase DBMS, WWW pages, etc.) including legacy systems (Folio). However, anyone who has built a wrapper before can attest that a lot of effort goes into developing and writing such a wrapper. In situations where it is important or desirable to gain access to new sources quickly, this is a major drawback. Furthermore, we have also observed that only a relatively small part of the code deals with the specific access details of the source. The rest of the code is either common among wrappers or implements query and data transformation that could be expressed in a high level, declarative fashion.

Based on these observations, we have developed a *wrapper implementation toolkit* [7] for quickly building wrappers. The toolkit contains a library for commonly used functions, such as for receiving queries from the application and packaging results. It also contains a facility for translating queries into source-specific commands, and for translating results into a model useful to the application. The philosophy behind our "template-based" translation methodology is as follows. The wrapper implementor specifies a set of templates (rules) written in a high level declarative language that describe the queries accepted by the wrapper as well as the objects that it returns. If an application query matches a template, an implementor-provided action associated with the template is executed to provide the *native query* for the underlying source[1]. When the source returns the result of the query, the wrapper transforms the answer which is represented in the data model of the source into a representation that is used by the application. Using this toolkit one can quickly design a simple wrapper with a few templates that cover some of the desired functionality, probably the one that is most urgently needed. However, templates can be added gradually as more functionality is required later on.

Another important use of wrappers is in extending the query capabilities of a source. For instance, some sources may not be capable of answering queries that have multiple predicates. In such cases, it is necessary to pose a native query to such a source using only predicates that the source is capable of handling. The rest of the predicates are automatically separated from the user query and form a *filter query*. When the wrapper receives the results, a post-processing engine applies the filter query. This engine supports a set of built-in predicates based on the comparison operators $=, \neq, <, >$, etc. In addition, the engine supports more complex predicates that can be specified as part of the filter query. The postprocessing engine is common to wrappers of all sources and is part of the wrapper toolkit. Note that because of postprocessing, the wrapper can handle a much larger class of queries than those that exactly match the templates it has been given.

Figure 1 shows an overview of the wrapper architecture as it is currently implemented in our TSIMMIS testbed. Shaded components are provided by the

*This work was supported by ARPA Contract F33615-93-1-1339, by the Anderson Faculty Scholar Fund, by the Center for Integrated Systems at Stanford University, and by equipment grants from Digital Equipment Corporation and IBM Corporation. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of the US Government.

[1] A native query is not necessarily a string of a well-structured query language, e.g., SQL. In general, the term "native query" may refer to any program used to access and retrieve information from the underlying source.

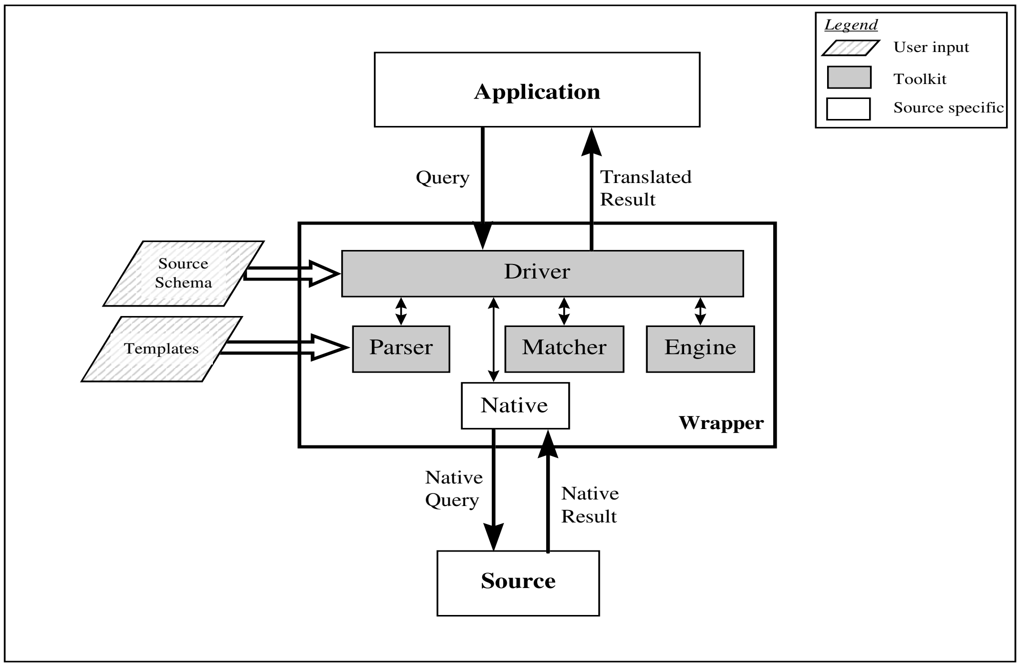

Figure 1: Wrapper architecture

toolkit, the white component is source-specific and must be generated by the implementor. The driver component controls the translation process and invokes the following services: the *parser* which parses the templates, the native schema, as well as the incoming queries into internal data structures, the *matcher* which matches a query against the set of templates and creates a filter query for postprocessing if necessary, the *native component* which submits the generated action string to the source, and extracts the data from the native result using the information given in the source schema, and the *engine*, which transforms and packages the result and applies a postprocessing filter if one has been created by the matcher. We now describe the sequence of events that occur at the wrapper during the translation of a query and its result using an example from our prototype system. The queries are formulated using a rule-based language called MSL that has been developed as a template specification and query language for the TSIMMIS project. Data is represented using our Object Exchange Model (OEM). We will briefly describe MSL and OEM in the next section. Details on MSL can be found in [5], a full introduction to OEM is given in [1].

2 An Example

OEM, is a self-describing object model with nesting and identity. Every object in OEM has an object identifier (OID), type, label, and value. The label carries semantic information about the object, i.e., describes the meaning of the object in a human readable form. The type is either atomic (e.g., integer, string, binary, etc.) or complex. The value of a complex object is the set of the object references to its subobjects. Certain OEM objects are chosen to be *top-level* or root objects, that have zero or more subobjects (also termed child objects). Top-level objects provide "entry points" into the object structure from which subobjects can be requested, as explained below. For example, the following structure represents an object labeled `answer` that consists of several `book` subobjects. Each `book` object has itself three subobjects labeled `author`, `year`, and `title`. Note that we will omit the type and OID from all OEM objects in order to improve the readability of the examples:

```
⟨answer {
    ⟨book {
        ⟨author "Jones"⟩
        ⟨year "1980"⟩
        ⟨title "Database Theory"⟩} ⟩
    ⟨book {
    ...     } ⟩
...} ⟩
```

OEM objects are created by the wrapper as the result of an MSL query. Since our example only uses a small subset of its syntax we will not describe the full functionality here. MSL is a rule-based language that extracts OEM objects (including their subobjects) by matching patterns in the query against existing OEM structures, for example, the wrapper templates. Each MSL query consists of a rule *head* and a rule *tail* separated by the `:-` symbol. *Variables* are represented by identifiers starting with capital letter, such as `P` and `Y`. The tail describes the search pattern, while the head describes the structure of the OEM objects that will be constructed. Intuitively, we match the tail pattern against the object structure exported by the wrapper, thereby binding the variables to object components of the wrappers' object structure. Note that when a field contains a constant (e.g., "Jones"), the pattern binds successfully only with OEM objects that have the same constant in the corresponding field (pattern matching may descend recursively through the object structure). The result of the query consists of all the objects (and their descendants) whose patterns match the query tail. For each of the matched objects, the object specified in the query head is returned.

For the remainder of this example, we are assuming that we have a relational database containing bibliographic information about papers and books for which we have developed a wrapper accepting MSL queries. Let us further assume that we have a user who is interested in all papers authored by "Jones" which have been published before 1984. A corresponding MSL query is:

$$P \;:-\; P :< book\ \{< year\ Y > < author\ \text{“}Jones\text{''} >\} > AND\ lt(Y, 1984). \qquad (1)$$

This query can be interpreted in plain English as

> *Find all publications labeled* `book` *which have subobjects* `year` *and* `author`*, and for which the author field has value "Jones" and the year field value is less than 1984.*

The `lt(Y, 1984)` predicate in the MSL query shown in (1) specifies that the $<$ comparison operator be used on the values of the `year` subobject rather than the default $=$ operator (as is the case with `author` subobject values).

Upon receipt by the wrapper, the query is sent to the driver component which invokes the parser. After the query is successfully parsed, the driver invokes the matcher to match the query against a set of template rules. These rules describe the queries that are accepted by the wrapper; in a sense, template rules are "parameterized MSL queries." Associated with each rule is an action string that describes the corresponding native query. Since we are assuming a relational DBMS as our source in this example, the action string is a parameterized SQL query. In order to demonstrate the postprocessing capabilities of the wrapper we have chosen not to include predicates on `year` in the templates[2].

[2] In a sense, we are assuming that our source does not support the $<$ predicate on `year`. However, in a "production-version" wrapper we always make use of all the natively supported predicates in order to maximize efficiency.

The following is an example of a template (without its action string) that matches the query shown in (1):

$$B \;:-\; B :< book\ \{< author\ \$X >\} > \qquad (2)$$

This template matches the input query because the substitutions $B \leftarrow P$ and $\$X \leftarrow$ "Jones" transform the template into an MSL expression that subsumes the input query, i.e., the resulting query returns a superset of the expected results. For details on the matching process as well as a discussion on query subsumption in MSL refer to [7]. Note that we could have designed a template that matches the input query exactly had we decided to let the source execute predicates on `year`.

Using the associated action

$$// \quad \$\$ = \text{“}select\ *\ from\ book\ where\ author\ =\text{''}\ \$X\ //. \qquad (3)$$

and the substitution $\$X \leftarrow$ "Jones", the matcher produces the following native SQL query:

```
select *
from book
where author = ''Jones''
```

The driver then invokes the query processing part of the native component which submits the native query to the source. When the result is returned, the driver invokes the query engine to perform postprocessing that is necessary in order to remove all those publications from the answer that were published on or after 1984 (since the original query was for publications before 1984). This is done by applying the following MSL filter query to the result:

$$B \;:-\; B :< book\ \{< year\ Y >\} > AND\ lt(Y, 1984). \qquad (4)$$

Specifically, the postprocessing engine takes each answer object in the native query result, extracts the `year` field of the object and checks if it is less than 1984. If so, the answer object is included in the result constructed by the engine. At the end, the engine returns only those entries whose `year` field is less than 1984. Finally, the driver passes the result back to the application that issued the original query.

3 Description of Demo

We demonstrate the latest version of our wrapper toolkit as described in the previous sections. Specifically, we show how to use our wrappers for accessing information from the following four different types of sources containing bibliographic data in heterogeneous formats.

1. A University-owned legacy system called FOLIO which is accessible through an interactive front-end (called INSPEC).

2. A Sybase relational DBMS which is accessible through SQL.

3. A collection of UNIX files which are accessible through a Perl script file.
4. A World-Wide Web source which is accessible through our WWW extraction utility [3].

Although all four sources support different access methods, the wrappers hide all source specific details from the application/end-user by exporting a common interface to the underlying data independently of where and how it is stored. By adding new templates or modifying existing ones, we show how one can quickly enhance the query capabilities of a wrapper or change the structure of the resulting answers without writing one line of code. Furthermore, we demonstrate the postprocessing capabilities of our wrappers.

As part of the TSIMMIS project we have also developed a graphical browsing tool called MOBIE [2] that lets users connect to TSIMMIS components using HTTP and the WWW. MOBIE provides mechanisms for submitting queries to wrappers and for navigating through OEM answer objects, zooming in on their nested substructures as necessary. In our demo we use MOBIE for submitting queries to the wrappers and for viewing the translated results as formatted, hyperlinked HTML pages.

References

[1] 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 10th Anniversary Meeting*, pages 7–18. Information Processing Society of Japan, Tokyo, Japan, October 1994.

[2] J. Hammer, R. Aranha, and K. Ireland. Browsing TSIMMIS data sources through the Web. Technical report, Department of Computer Science, Stanford, California, February 1997.

[3] J. Hammer, H. Garcia-Molina, R. Aranha, and Y. Cho. Extracting semistructured information from the Web. Technical report, Computer Science Department, Stanford University, Stanford CA 94305-9040, February 1997.

[4] J.W. Lewis. Wrappers: integration utilities and services for the DICE architecture. In *Proceedings of the Second National Symposium on Concurrent Engineering*, pages 445–457. Concurent Engineering Research Center, Morgantown WV, 1991.

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

[6] 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.

[7] 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.

[8] D. Wells. Wrappers. Survey. URL. http://www.objs.com/survey/wrap.htm.