ELSEVIER

The Journal of Systems and Software 54 (2000) 185–199

The Journal of Systems and Software

www.elsevier.com/locate/jss

# Modularized design for wrappers/monitors in data warehouse systems

Jorng-Tzong Horng [*], Jye Lu

*Department of Computer Science and Information Engineering, National Central University, Jungli 320, Taiwan, ROC*

Received 20 August 1998; received in revised form 27 May 1999; accepted 25 July 1999

**Abstract**

To simplify the task of constructing wrapper/monitor for the information sources in data warehouse systems, we provide a modularized design method to re-use the code. By substituting some parts of wrapper modules, we can re-use the wrapper on a different information source. For each information source, we also develop a toolkit to generate a corresponding monitor. By the method, we can reduce much effort to code the monitor component. We also develop a method to map the object-relational schema into relational one. The mapping method helps us make an uniform interface between wrapper and an integrator. © 2000 Elsevier Science Inc. All rights reserved.

## 1. Introduction

A data warehouse system (Hammer et al., 1995) is a repository of integrated information, which can be provided for query or analysis. Data warehouse systems can collect and maintain information from multiple distributed, autonomous, or heterogeneous information sources. When initializing of information sources or modifications occur in information sources, related data retrieved from the information source can be processed and transformed into internal types available for data warehouse systems. The related data may be integrated with other information already existing in a data warehouse system. Queries and analyses will respond efficiently by a data warehouse system, for part of results responded to user has been calculated and restored in a data warehouse system.

The main functions of data warehouse systems are retrieving, filtering, integrating related information required by complex queries. In contradiction to an on-demand approach (extract data only when processing queries) of traditional databases, data warehouse systems provide an in-advanced approach (interested data are retrieved from information sources in advance). It is not necessary to re-calculate the whole query result, because some parts of the result has been calculated and restored in repository (data warehouse), and these parts can be used directly by data warehouse system. Therefore, required time can be reduced by submitting complex queries to data warehouse system.

Because processed information has been stored in data warehouse, there exists inconsistency between data warehouse and underlying information sources. According to the warehouse information project at Stanford (WHIPS) architecture (Hammer et al., 1995), we can use monitor/wrapper components to detect modification in information sources and to maintain the consistency between information sources and data warehouse system.

The WHIPS system uses the relational model to represent the warehouse data: materialized views are defined in the relation model and the warehouse stores relations (Wiener et al., 1996). All the information sources convert the underlying data to the relational model. The integrator component integrates the information retrieved from information sources. The integrator sends queries to the information source; these queries are handled by *wrappers* which translate the queries to internal representation.

The integrator component also maintains the consistency between the information sources and the data warehouse system. Any update information will be sent to the integrator by the monitor component of the information source.

When modifications, such as insertion, deletion and updates, occurred in underlying information sources, monitor must detect the situation and notify the integrator. The integrator then asks the view managers

[*] Corresponding author. Tel.: +886-3-4227151 ext. 4519; fax: +886-3-4222681.
*E-mail address:* horng@db.csie.ncu.edu.tw (J.-T. Horng).

related to the modification to respond to this modification. The view manager sends the requests to query processor which propagates the query request to a wrapper. At last, the wrapper retrieves query results from the information source and returns the results to the query processor (Wiener et al., 1996).

According to the degree of support by information sources, we can classify these information sources into four types (Widom, 1995):

*Cooperative sources*. Information sources provide notification capabilities, e.g. triggers. Modification can be concretely notified by cooperative information sources. When coding a monitor, programmers can use this notification facility to simplify their job.

*Logged sources*. Log files maintained by an information source are available for monitor. This will be extravagant if log files are very huge. There is a more serious problem when using logged sources, because it often needs administrator privileges to access the log files.

*Queriable sources*. Information source allows monitor to query the data in the information source. We can use periodic polling to detect the change in information source. If the polling frequency is too high, the performance will degrade; if the polling frequency is too low, data warehouse system responds the modification inefficiently.

*Snapshot sources*. If information source does not provide the above methods, we compare the snapshot with an earlier one. This method requires most cost; it is a problem to reduce the cost when implementing the snapshot comparison (Labio and Garcia-Molina, 1996).

Monitor/wrapper is germane to underlying information sources, so we code different monitors/wrappers for different information sources. It is a wasteful cost to re-write the monitor/wrapper for each information source. We can divide the wrapper into several modules. When new protocols are applied or new information sources occupied, we can substitute some modules and reuse other modules to construct wrapper/monitor rapidly. By this method, we can decrease onerous coding.

The job of a wrapper is to accept query request sent by query processor and obtain results from the information source. After obtaining the results, wrapper transforms the results into the internal form of the data warehouse system. All wrappers offer query processors the uniform interface to hide the complexity at information sources.

For example, data warehouse system uses relational algebra and information sources use SQL. Wrapper translates the query which is requested by the query processor from relational algebra into SQL, sends it to the information source, and returns the results to query processor. If there are other query forms in new information sources, wrapper must be re-written to the corresponding form. In WHIPS system, query processor sends the query by a view tree (which resembles relational algebra) (Wiener et al., 1996).

Wrappers need to be re-written not only for the different forms of information sources, but also for the same type of sources (e.g., relational database) with different interfaces or different degrees of supporting.

There are problems in how to divide wrapper into modules, what function each component should be responsible for, how to communicate to another component, how to represent the result, and how to transform the representation between the information source and the data warehouse system.

In the past, a wrapper was emphasized in heterogeneous information sources and neglected to heterogeneous data warehouse systems. After dividing wrapper into modules, we hope modules can be re-used not only in heterogeneous information sources, but also in heterogeneous data warehouse systems to reduce the cost. By changing some parts of the wrapper, we can re-use wrapper/monitor to a new information source or a new data warehouse system.

This paper focuses on how the modularized design is applied on a wrapper, and discusses how to solve the mismatch between the query processor and the information source.

The remainder of the paper is organized as follows. In Section 2, we overview the related work. In Section 3, we propose the architecture and modules for designing a wrapper/monitor. We show some examples in Section 4 and conclude in Section 5.

## 2. Related work

The goal of WHIPS (Hammer et al., 1995) is to develop algorithms and tools for the efficient collection and integration of information from heterogeneous and autonomous sources, including legacy sources. There are three main components in WHIPS architecture: data warehouse, integrator, and monitor/wrapper.

*Data warehouse* stores integrated information available for applications. The concept in data warehouse system is to retrieve, and process relevant information before queries sent by applications. When queries arrive at the data warehouse, we can check whether this query is associated with those information in data warehouse or not. We can directly use the stored information, available for the query, without extra effort to process the query. The data warehouse in the WHIPS architecture may be any relational database (Wiener et al., 1996). The data warehouse in WHIPS also stores metadata of view definitions in the internal format. The warehouse-wrapper insulates other data warehouse system components from the data warehouse, so we can use different kinds of database system.

*Integrator* receives update notification sent by monitor. If this update affects integrated information in the data warehouse, integrator must take appropriate actions, including retrieving more information from information sources. The detail modules of data integration component in Hammer et al. (1995) contain view manager modules, a query processor module, and an integrator module (Wiener et al., 1996). A *view manager module* is responsible for maintaining its view. We can use different maintaining algorithms to achieve different degrees of consistency in individual cases. In WHIPS architecture, view managers can maintain different degrees of consistency by different algorithms (Zhuge et al., 1995). The *query processor module* distributes the query, whose irrelevances have been pruned off, to wrappers. Query processor will integrate the results of each individual queries into a global query result. The main role of the *integrator module* is to facilitate view maintenance. The integrator judges whether an update sent by a monitor is relevant to a view or not. If necessary, the update will pass to the view managers being interested in that modification.

*Monitor* component detects the modification applied to the information source. These modifications will be passed to integrator module. *Wrapper* component translates queries propounded by query processor from internal representation used by data warehouse system to native query language used by information sources.

The goal of the TSIMMIS (Chawathe et al., 1994) project is to develop tools that facilitate the rapid integration of heterogeneous information sources that may include both structured and unstructured data. TSIMMIS project uses a common information model *object exchange model* (OEM) to represent the underlying data. OEM retains the simplicity of relational model while allowing the flexibility of object-oriented models (Papakonstantinou et al., 1995). Translator in TSIMMIS converts the query language in OEM into the native query language, and converts the results into OEM. *Mediator specification language* (MSL) can be seen as a view definition language that is targeted to the OEM data model and the functionality needed for integrating heterogeneous source. Therefore, MSL allows the declarative specification of mediators in TSIMMIS (Papakonstantinou et al., 1996).

University of Maryland has proposed an architecture of an *interoperability module* (IM) to process queries on heterogeneous databases (Chang, 1994; Chang et al., 1994; Hammer et al., 1997). Query transformation can be made by using an F-logic to express the mapping information among different schemata. The IM resolves the conflicts among different databases by two kinds of parameterized canonical representations (CR). Chang (1994) proposed two kinds of parameterized canonical form to resolve two kinds of heterogeneity, query language and different schema, respectively. The first CR resolves the heterogeneity caused by different query languages. We can extract and represent query semantics by the first CR form. The second CR serves the mapping among different schemata by corresponding algorithms. The mapping information will be stored in the global knowledge dictionary. When a query transformation is performed, IM can retrieve mapping information from that dictionary.

For some non-cooperative information sources, we need to detect modifications by periodically comparing snapshots. Labio and Garcia-Molina (1996) proposed a *Window Algorithm* to solve the snapshot differential problem. Although window snapshot algorithm reduces a lot of cost, this is still an expensive operation to compare two huge snapshots.

Ashish and Knoblock (1977) provides database-like querying for semi-structured WWW sources by building wrappers around these sources. Gruser et al. (1998) presents technology to define and (automatically) generate wrappers for web accessible sources. However, these approaches focus on the semi-structured web data, which differ from our information sources based on the databases. The CQ project aims at developing a scalable toolkit and techniques for an update monitoring and event-driven information delivery on the net (Liu et al., 1998). Liu et al. (1998) is just adequate for queriable information source (Widom, 1995).

## 3. System design and implementation

The concept of this paper is to reduce the onerous job when we encounter to develop a wrapper/monitor. Our wrapper/monitor has the following advantages.

*Reduce developing time*. We propose a modularized design to solve the variable problem of information sources or query processors. When the information source or the protocol between wrapper and query processor is changed, wrapper can be made ready quickly to work by interchanging the modules.

*Re-use the code*. Most of codes can be re-used when we design a wrapper. This situation is especially obvious when information sources have the same data model. We want to increase the ratio of code-reuse by modularized design and object-oriented technology.

*Increase the portability*. Information source may reside on many types of platforms, therefore we should take the portability into account. For example, PostgreSQL may be run on FreeBSD or Solaris; if we can reuse the codes, the developing time will be slashed.

### 3.1. System architecture

#### 3.1.1. Functions of each module

We first briefly describe functions of each module in our system architecture shown in Fig. 1. *Driver* is

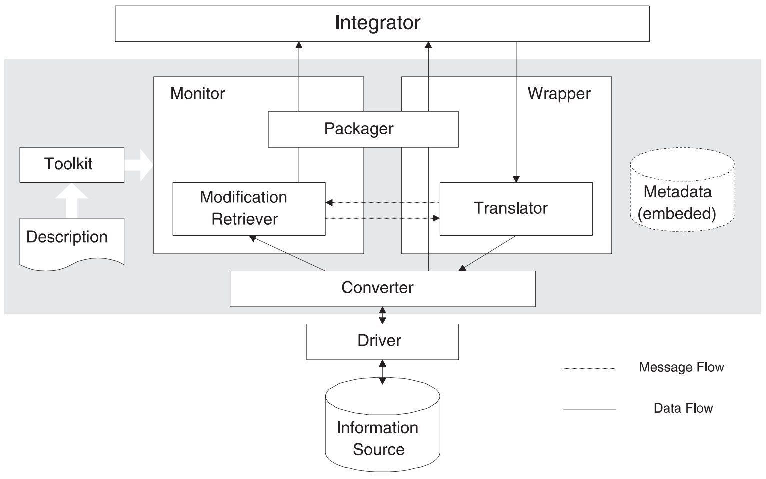


Fig. 1. System architecture.

responsible for retrieving data requested by other modules. *Converter* resolves the representation conflicts among the information source and wrapper/monitor. *Modification retriever* detects the changes in information source and propagates the messages to notify integrator. *Packager* transforms data from internal form into the form recognizable by integrator. *Translator* resolves the schema conflicts between the information sources and the integrator.

### 3.1.2. *The Interaction between monitor/wrapper and integrator*

*System initialization*. We initialize each wrapper of information sources when the integrator is started (Fig. 2(a)). Wrapper will start up the monitors which belong to the information source (Fig. 2(b)). The wrapper notifies the integrator what relations it handles, and monitors send the corresponding relation schema to the integrator. All schema information will be registered at integrator (Fig. 2(c)). Finally, monitor module checks the update message. If monitor finds any update, it will notify the integrator (Fig. 2(d)).

*Periodically update detection*. In our approach, updates will be stored until a modification retriever module detects these changes. The period can be determined by the administrator of data warehouse systems.

Every update will be stored in a table before it is sent (Fig. 3(a)). Administrator can set the period determined by the update frequency. At the stated intervals, monitor sends the updates to the integrator (Fig. 3(b)) and cleans the table (Fig. 3(c)).

Then, when another update applied on the relation (Fig. 3(d)), the update will be recorded in the table (Fig. 3(e)). The update will be sent to the integrator at the next predetermined time (Fig. 3(f)).

*Query arrived at wrapper*. When a query arrived at the wrapper component, the wrapper first notifies the monitor to detect the updates (Fig. 4(a)). The monitor detects the modifications from the information source (Fig. 4(b)) and sends the updates to the integrator

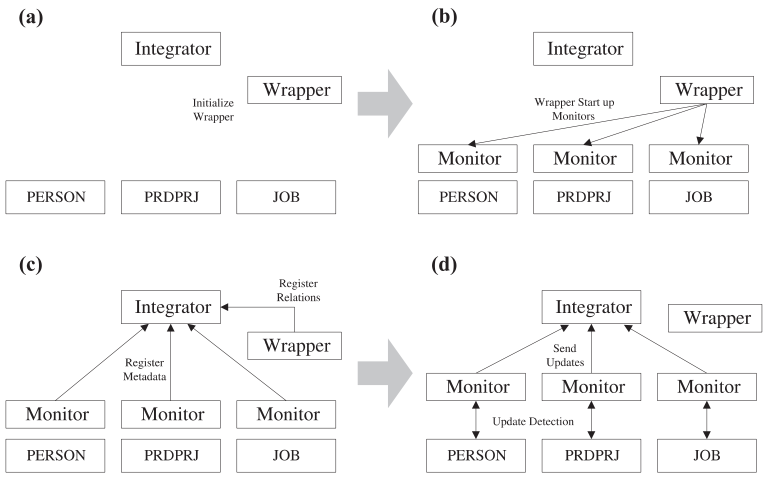


Fig. 2. System initialization.

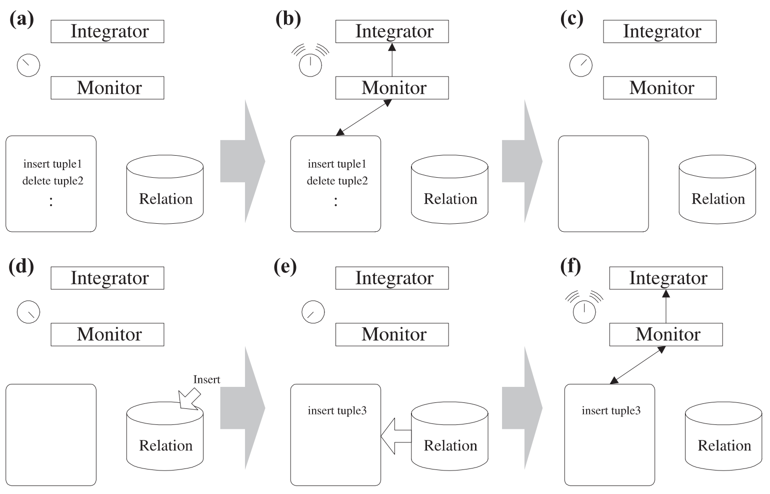


Fig. 3. Periodically update detection in Oracle.

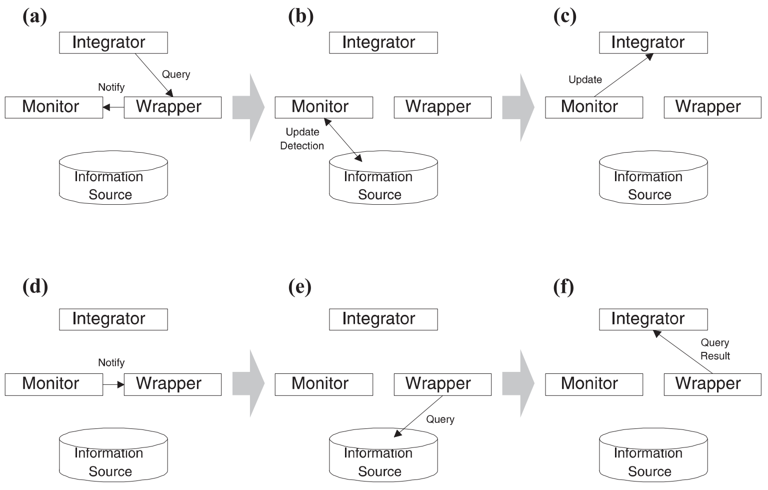


Fig. 4. Query arrived at wrapper.

(Fig. 4(c)). Wrapper does not send the query to the information source until the update detection completes.

When monitor completes its detection, it will notify the wrapper to continue the query (Fig. 4(d)). Then, wrapper sends the query to information source and gets the results (Fig. 4(e)). Finally, the results will be sent to the integrator (Fig. 4(f)).

## 3.2. The design of each module

In this subsection, we introduce a design of each module in system architecture.

### 3.2.1. Packager

Internal data type used by wrappers and monitors is defined for efficiency or simplicity, but the type may differ from the type used between the integrator and packager. Packager module is responsible for transforming internal data type into one that an integrator can recognize.

For example, if the information source is a relational database, modification retriever module may detect an update tuple containing a `CHAR` type attribute. The `CHAR` attribute, represented as `character` type in internal type, could be represented as an attribute of an object in integrator. Packager module is like the packager in wrapper of TSIMMIS (Papakonstantinou et al., 1995), which is responsible for transforming data into OEM objects.

### 3.2.2. Modification retriever

Each modification retriever listens to its information source. Any change on the information source must be detected by the modification module. The modification retriever module can retrieve data from converter, filter the interested data, then ask the packager to propagate these data to integrator.

If the information source provides sufficient support, it will reduce a lot of time to develop the module. When the module works with a non-cooperative information source, it can use other detecting methods to achieve update detection. There are three general update approaches (Zhuge et al., 1995):

*Immediate update*. The update that occurred in information source will be propagated immediately.

*Deferred update*. The update that is propagated when a query is arrived.

*Periodic update*. The update that is propagated periodically.

Our approach is the combination of a deferred update and a periodic update.

#### 3.2.3. Translator

Translator module provides mappings of query language and schema between the integrator and an information source. Among heterogeneous systems, the main job is to code the translator module when a wrapper is developed. A translator module may provide a different scene, which differs from the information source. It means that we can provide the integrator with a different data model or a different schema. If the integrator uses the same data model and query language of the information source, the translator module is just responsible for extracting the semantic of the query. In homogeneous environment, we can even degrade this module from translating to forwarding a query.

#### 3.2.4. Converter

Converter module resolves conflicts in data representation. For example, we may use different representations of the same DATE data in databases, e.g., '1975-03-19', so the module takes responsibility of transforming into the same style that the integrator has, e.g., '19-MAR-75'. Converter serves not only the wrapper but also the monitor, because the detected information also needs to be transformed to the style belonging to the integrator.

#### 3.2.5. Driver

Driver module processes the query forwarded by the converter module. Driver module should be provided by an information source vendor, or be coded by the programmer in the worse case. The interface of a database driver tends to be unified or use multi-tier architecture. This makes it easier to develop a Converter module quickly.

#### 3.2.6. Miscellaneous modules

The Toolkit uses a description file, which describes the schema of the information source, to generate the corresponding monitor. We can rapidly develop a monitor by using the toolkit. For different information sources, we should develop corresponding toolkits to match the demands.

Metadata provides the processing information about the information source to execute query, to transform data and to retrieve data. It may be embedded in the code of each module.

### 3.3. Implementation of the tool and its environment

The whole system is implemented by Java Language. Java gives benefits when we code the wrapper/monitor, but it also has drawbacks. The main benefit of Java is its portability. Once we compile the code, the compiled code can be run everywhere. It will obviously reduce the developing time. During the implementation, this characteristic lets us transfer our programs to other platforms without re-compiling our source code. The main drawback is its speed because the Java program is interpreted by Java Virtual Machine. The efficiency may be solved by 'just-in-time' compiler.

We distribute the Java-to-Java applications by *remote method invocation* (RMI). Java program can call methods of remote objects once it gets the reference to the remote object. It is much simpler to communicate with objects in other sites. In the WHIPS, they use CORBA (ILU) to hide the low-level communication. RMI can achieve the same goal but it is simple when we use Java language. By the way, Java also provides an IDL to support CORBA.

*Java database connectivity* (JDBC) is an access interface of relational databases. It provides a uniform way to access different relational databases by Java. Because of the unified interface, it is much simpler to develop a wrapper/monitor. JDBC uses objects to send query and retrieve data. We can re-write related classes to intercept the result set and apply operations to it.

BYACC/Java is a YACC-compatible parser generator. The BYACC can generate Java source codes available for us to develop a translator module. The JLex utility is based upon the Lex. The input file resembles what is accepted by Lex. JLex takes this input file and generates the source code of a lexical analyzer.

In this paper, we use two database systems, PostgreSQL and Oracle, as the information sources when we implement our system. In the early time of our implementation, we use PostgreSQL 6.2 and Oracle7 on Solaris. We use PostgreSQL 6.3 on FreeBSD and Oracle8 on Solaris later in our implementation. These changes are easily overcome because we use Java to implement our system.

### 3.4. Detailed implementation of each module

#### 3.4.1. Driver module

We use the JDBC driver provided by DBMS vendors as the driver module. The JDBC driver uses SQL as the query language, and the result set can be retrieved by a cursor. JDBC also supports the transactions. So the ACID requirement can be achieved by JDBC drivers.

### *3.4.2. Converter module*

The main job of converter module in our approach is to resolve the conflicts in data representation. The converter module provides an interface like JDBC. If integrator use a relational schema, it can directly communicate with converter module without the translator module.

When a query arrived at converter module, we use a parser to find the conflicts and between different data formats and convert them to one that information source has. Then the translated query will be sent to driver module.

A `ResultSet` maintains a cursor pointing to its current row of result data. Because the query result is retrieved through the `ResultSet` class, we can intercept the retrieving action by re-writing the `ResultSet` class. For every conflicting type, we also write a corresponding class to resolve the conflicts. Once the result is retrieved through the `ResultSet`, our converter will convert the data representation.

We provide the same interface as translator module, so integrator can directly communicate with convert.

### *3.4.3. Packager module*

The packager module in our approach is responsible for translating the internal data into strings. The data between the integrator component and package module is communicated through string.

The reason to use a string as a data type is simple because the integrator component can directly form a query by using these strings and transform them into another type. The actual type of the result can be looked up in the metadata registered by the monitor.

Besides, each object has `toString()` method in Java, so the transformation can be directly applied on every object. We can also define our class type, which uses the same method or overrides the method, to support new data types.

### *3.4.4. Modification retriever module*

In our approach, we combine the deferred update and periodic update to maintain the view consistency. It means we detect updates periodically or when a query sent to wrapper. The period can be adjusted by administrator of data warehouse system.

For cooperative information sources, such as Oracle, we use triggers to detect the modification in the information source. We record these information in another table, and periodically retrieve data from this table. Thus, we can use these information to notify integrator.

For non-cooperative information sources, such as PostgreSQL, we use a snapshot algorithm to retrieve the modification. We duplicate a copy of original table, and periodically compare two tables. The snapshot algorithm is undoubtedly inefficient, but we may have no other choice in some cases.

As a result of the Strobe algorithm (Zhuge et al., 1996), modifications are reported as `INSERT` or `DELETE`. All contents are reported as strings by the packager module. The update action is represented as the delete–insert pair.

In our approach, the source code of this module is generated by the toolkit module.

### *3.4.5. Toolkit module*

If there are a lot of tables in one database, it will be very helpful to use a toolkit to generate the modification retriever module. It will be painful to re-write the similar code many times.

Since the monitor should send the metadata to an integrator, we must get the information from either the information source or the administrator of the data warehouse. The metadata contains name of the relation, key information and type information. If the converter module takes care of some types, we must provide the converting information.

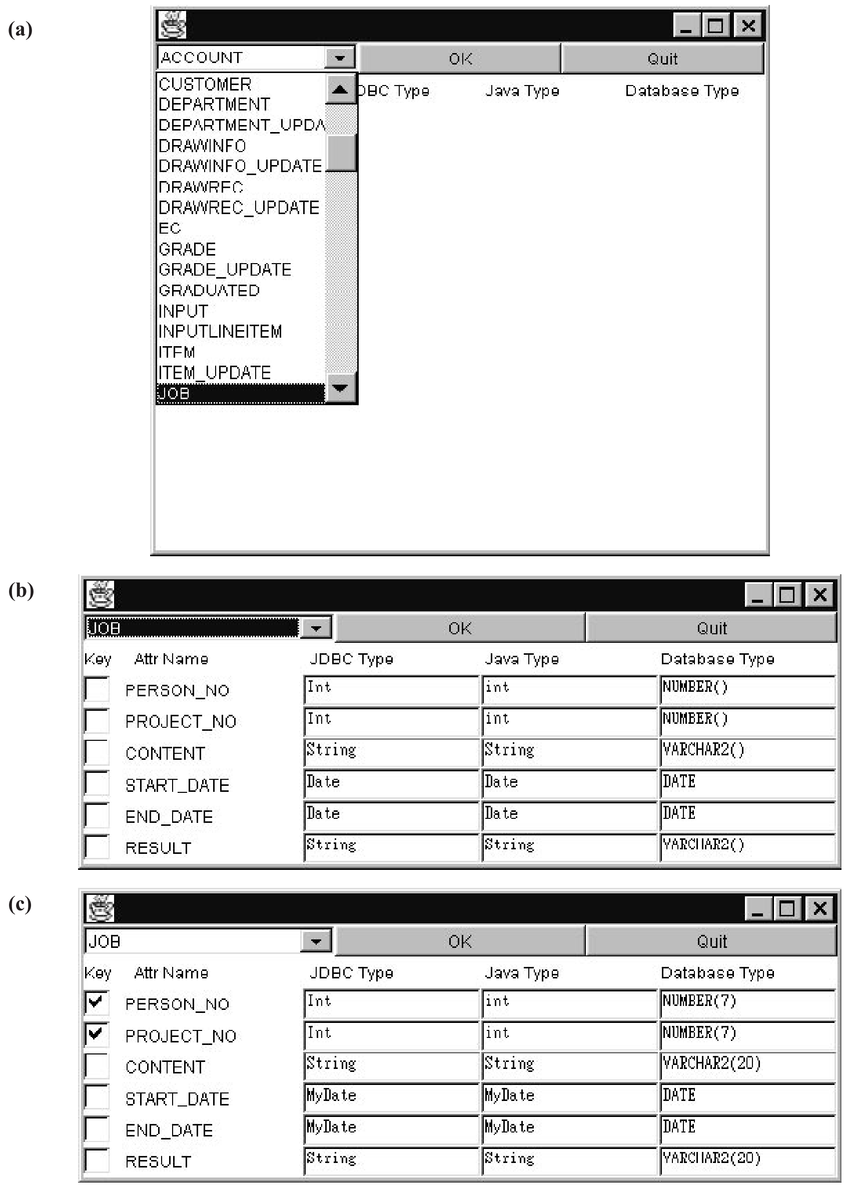
This module takes the description file and generates the corresponding source code of the modification retriever module. The input file describes the relational name, attributes, types, primary key attributes, and internal types. The description file can be generated by any text editor or by our toolkit interface (Fig. 5(a)).

The upper window of Fig. 5 displays the situation when we choose a relation in the menu on upper left corner. After we specified the relation, the toolkit interface retrieves the metadata from the information source and displays the default types (Fig. 5(b)).

Then, we select the key attributes and adjust the database types to appropriate ones (Fig. 5(c)). In general, we do not have to modify the 'Java types' and 'JDBC types'. If there exist some types, we want to convert, e.g., `DATE` type, we can re-write the Java types or JDBC types and provide the corresponding classes. When we re-write the Java types and JDBC types, the modification retriever module will invoke the corresponding `get`*`Type`*`()` method provided by the converter.

After generating the description file, we can create a monitor by the description file. Our toolkit generates the monitor component and the modification retriever module. In our approach, we can save much effort when we develop the monitors in information sources with vast amount relations.

When the information source changes, we can modify the description file and generate modification retriever quickly. In different information sources, we must develop corresponding toolkits to meet the different demands.



(a)

ACCOUNT | OK | Quit

CUSTOMER
DEPARTMENT
DEPARTMENT_UPDA
DRAWINFO
DRAWINFO_UPDATE
DRAWREC
DRAWREC_UPDATE
EC
GRADE
GRADE_UPDATE
GRADUATED
INPUT
INPUTLINEITEM
ITEM
ITEM_UPDATE
JOB

DBC Type | Java Type | Database Type

(b)

JOB | OK | Quit

| Key | Attr Name | JDBC Type | Java Type | Database Type |
|---|---|---|---|---|
| ☐ | PERSON_NO | Int | int | NUMBER() |
| ☐ | PROJECT_NO | Int | int | NUMBER() |
| ☐ | CONTENT | String | String | VARCHAR2() |
| ☐ | START_DATE | Date | Date | DATE |
| ☐ | END_DATE | Date | Date | DATE |
| ☐ | RESULT | String | String | VARCHAR2() |

(c)

JOB | OK | Quit

| Key | Attr Name | JDBC Type | Java Type | Database Type |
|---|---|---|---|---|
| ✓ | PERSON_NO | Int | int | NUMBER(7) |
| ✓ | PROJECT_NO | Int | int | NUMBER(7) |
| ☐ | CONTENT | String | String | VARCHAR2(20) |
| ☐ | START_DATE | MyDate | MyDate | DATE |
| ☐ | END_DATE | MyDate | MyDate | DATE |
| ☐ | RESULT | String | String | VARCHAR2(20) |

Fig. 5. Toolkit interface.

### *3.4.6. Translator module*

Translator resolves the conflicts of query language and schema between the integrator and the information source. In our approach, we provide a relational schema to the integrator component. Therefore, we should map the schema between the underlying information source and the integrator.

Oracle8 is an object-relational database system. When we use Oracle8 as one of the information source, we must provide some mappings. In the following, we introduce these mappings. The direction of the mapping is from the translator to the integrator.

*3.4.6.1. OID. We map an OID into a primary key.* We need to identify the primary key to satisfy the demand of Strobe algorithm (Fig. 6). Oracle8 can offer an oid for

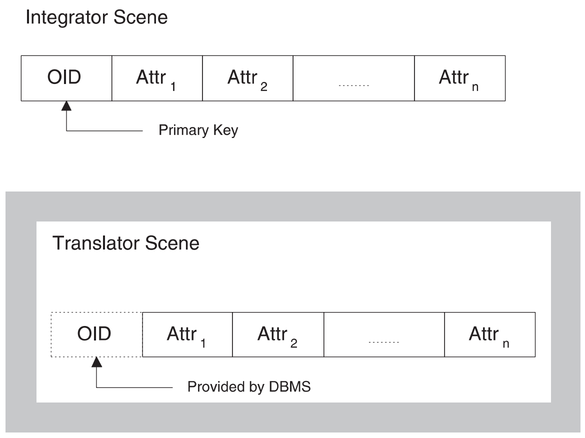


Fig. 6. Mapping between primary key and OID.

each object, so we can directly use oid as the primary key.

To retrieve the primary key using the project operation is meaningless because the primary keys are OIDs. Once OID appears in query string, it has special meanings that will be discussed later.

*3.4.6.2. CLASS. We map the class into a relation.* An integrator will see the relation we provide. When the integrator retrieves data from the relation, we map it to retrieve data from a class (Fig. 7).

A leaf-level attribute is either a built-in type or a collection type. The leaf-level attributes which are not collection types are called the leaf-level scalar attributes of the object type. A leaf-level scalar attribute will be mapped into an attribute of a relation. A collection type attribute will be discussed below.

REF is an oid referencing to another object. REFs will be mapped into foreign keys that refer to primary keys, i.e., OIDs.

Querying on leaf-level scalar attributes result in returning the attributes. There is an assumption that query results will not contain the REFs. It is meaningless to query a foreign key made by an OID.

*3.4.6.3. Set. We map the set into another table.* There are two kinds of collection types, *nested tables* and *VARRAYs*. We just handle the nested table. We map the nested table into another relation, which has a foreign key referencing to the original relation (Fig. 8).

When queries are applied on the mapped relation, we can translate them into the native form.

*3.4.6.4. Relationship. Association. We map references into foreign keys.* As discussed above, the REF will be mapped into a join between two relations.

*Nested attribute. We map the nested attributes into another relation.*

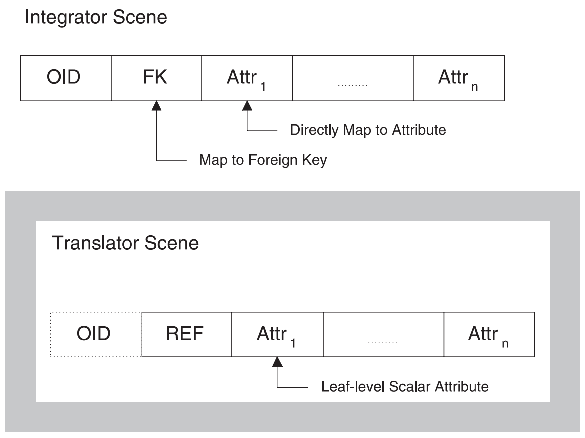


Fig. 7. Mapping between relation and class.

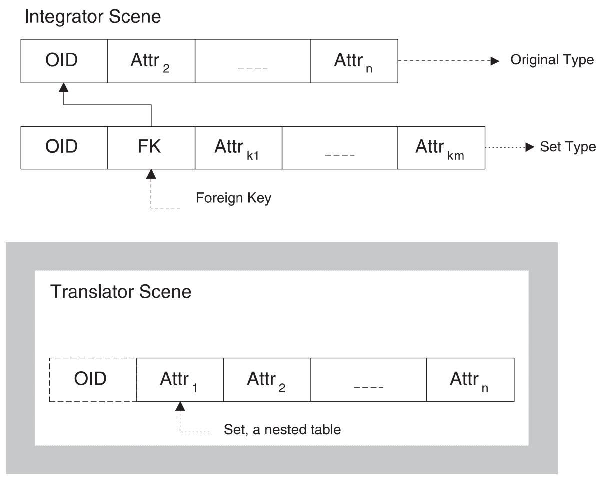


Fig. 8. Mapping between set and another relation.

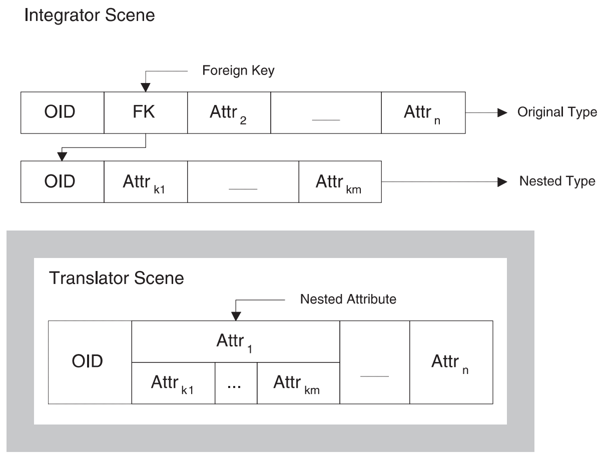


Fig. 9. Mapping between nested attribute and another relation.

We map the column object (nested attribute) into a foreign key which refers to an instance of the additional relation (Fig. 9).

Querying on the additional relation will result in translating the query.

## 4. Examples

### 4.1. Construction of information source

In this section, we construct an information source based on a modified PDM system in Chang and Trappey (1996) (Fig. 10). The database uses a new ballpoint pen project to demonstrate the PDM system execution procedure. We assume relations in the database are distributed to two sites, Oracle and PostgreSQL, and the integrator uses the dialect of Oracle. We briefly describe the relations used instead of introducing the whole database.

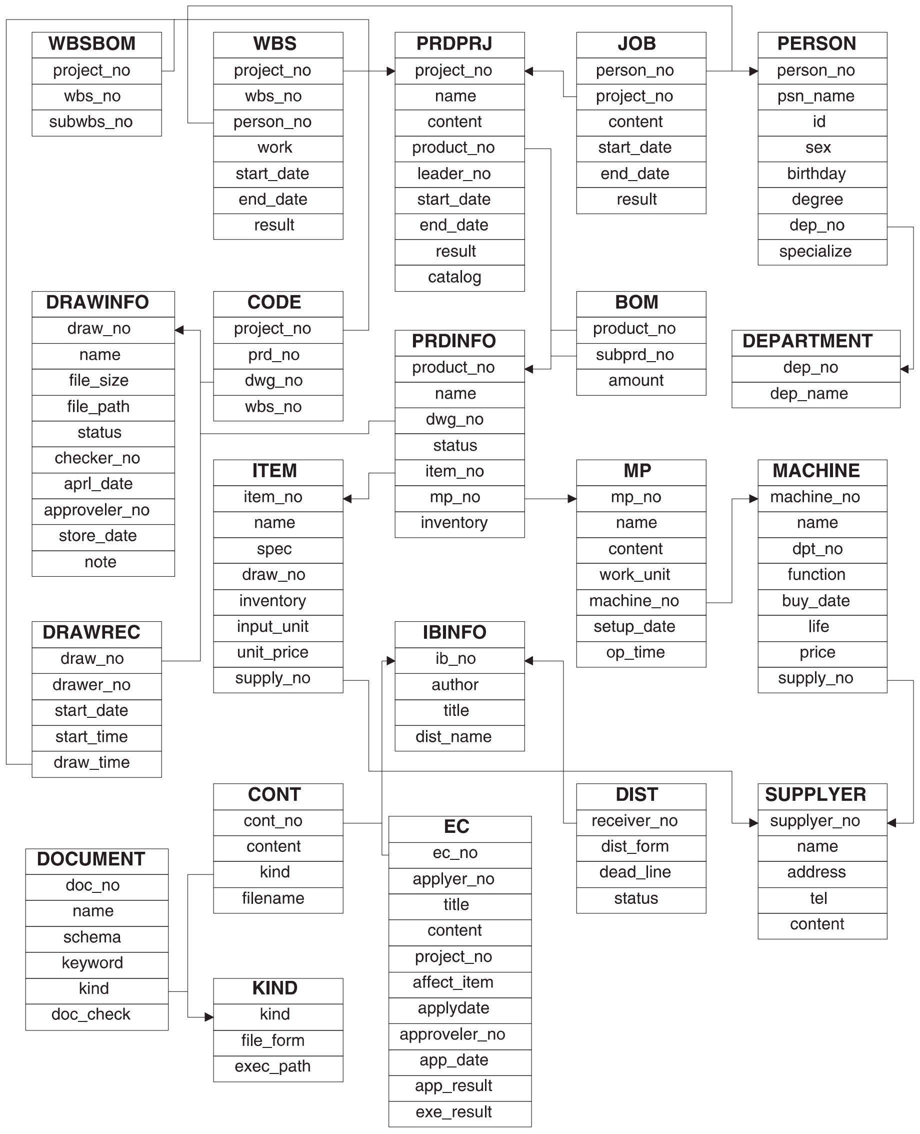


Fig. 10. PDM schema.

1. JOB. The relation JOB records every staff's responsible works and his work status. If project leader assigns some works, the works will be recorded in this relation. The responsible person can access this table to know what the works should do. The JOB relation resides on the Oracle.
2. EC. The relation EC contains the engineering change request data, including applicant, title, content, belongs to which project, affect item, apply date, approver, apply result and executive result. The EC relation resides on the PostgreSQL.

### 4.1.1. Wrapper

In this PDM example, we use relational data model. So, SQL can be directly sent to underlying database system. The most serious problem when we develop a wrapper in PDM schema is the data representation conflict.

*Oracle*. Since the integrator uses the same dialect, the wrapper only forwards the query string to the converter module. And this string can be handled by JDBC directly. In contrast to PostgreSQL, the development of wrapper component in Oracle seems to be more straightforward.

*PostgreSQL*. The main difference between PostgreSQL-wrapper and Oracle-wrapper is the data representation in query strings. The style used by integrator in Oracle needs to be translated into what is used in PostgreSQL.

We write a parser to translate the query string. Therefore, the date string will be replaced before sending the SQL to PostgreSQL.

### 4.1.2. Monitor

As we see in Section 3, we use the toolkit to generate the monitor components in both DBMSs. We use the

JOB relation in PDM schema to show how we generate the monitor. The description file generated by our toolkit interface is given below.

```
JOB
0 1
PERSON_NO Int int NUMBER(7)
PROJECT_NO Int int NUMBER(7)
CONTENT String String VARCHAR2(20)
START_DATE MyDate MyDate DATE
END_DATE MyDate MyDate DATE
RESULT String String VARCHAR2(20)
```

When the monitor component registers its information at the integrator, it will send metadata to integrator. The first line is the relation name; the second line is the keys of this relation. The remaining lines are the attributes of this relation. The first column is attribute name; the second column is JDBC type; the third column is internal type which can be handled by Java. The last column is the type reported to integrator. These information let our toolkit to generate the corresponding monitor component.

*Oracle*. Since Oracle is a cooperative information source, we can create triggers to record the changes. Therefore, we create triggers and an additional table for each relation.

For example, we record the updates of JOB table in JOB_UPDATE table:

```
 SQL> describe JOB_UPDATE

 Name              Null?          Type
------------------ --------       ------------------
SQ                                NUMBER(8)
TYPE                              NUMBER(2)
PERSON_NO                         NUMBER(7)
PROJECT_NO                        NUMBER(7)
CONTENT                           VARCHAR2(20)
START_DATE                        DATE
END_DATE                          DATE
RESULT                            VARCHAR2(20)
```

The SQ is a sequence number, we can sort the updates by this attribute, and TYPE records event type (delete or insert).

*PostgreSQL*. Unlike Oracle, PostgreSQL 6.2 does not provide the notification service. Therefore, we use a snapshot algorithm to detect the updates. We create a table to store the snapshot, then periodically compare them.

### *4.1.3. Generating monitors with a toolkit*

In our PDM example database, we briefly show how much effort we saved in Table 1. The description file can be created by our toolkit interface, then our toolkit will process the file and generate the monitor component. This means that the users can generate monitors without any additional coding when they use our toolkit interface.

For example, we use the JOB relation in PDM schema to show how we generate the monitor. The description file generate by our toolkit interface is given in (B). Our toolkit then processes the description file and generates the monitor source code. The user uses the Java compiler to compile the source code. The corresponding monitor(a Java application) is generated.

The approach we use to generate the monitor source code is like that of macro processor in assembler. We get the information from the description file and put it into the appropriate place in the source code. Fig. 11 shows the different steps of generating a monitor for a relation.

As we see in Table 1, there are totally 1942 lines in six monitors on the PostgreSQL, while 1322 lines in 15 monitors on the Oracle. In the example, we can save about 80% effort to code our monitors (1322 lines) by the toolkit (200 lines) on Oracle. We can also save about 65% effort to code our monitors (1942 lines) by the toolkit (629 lines) on PostgreSQL.

Unlike C/C++ programming language, Java does not provide a preprocessor or template features, so we must develop a toolkit module to simplify the development of monitors. The toolkit module plays the role of preprocessor in C/C++ programming language.

Table 1
Generating monitors with a toolkit

| | Toolkit file (lines) | Tables | Generated files (lines) | Description files (lines) |
|---|---|---|---|---|
| Oracle | 200 | 15 | 1322 | 107 |
| PostgreSQL | 629 | 6 | 1942 | 41 |

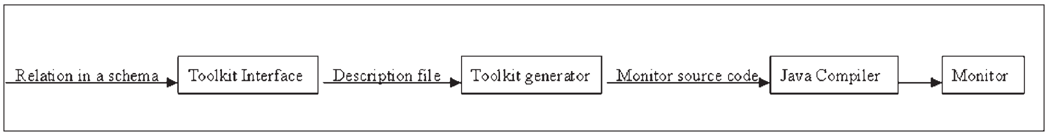


Fig. 11. Steps of generating a monitor.

There are two situations in which we can save the effort to develop a monitor by toolkit: vast amount tables and non-cooperative information sources.

The average size of a monitor is usually shorter than the toolkit module, once there are few tables in the database, the cost is even higher when we develop a toolkit. The more tables on a information source, the more benefit we can acquire. In our PDM example, Oracle database is a good paradigm of this situation.

Coding a monitor on a non-cooperative information source (PostgreSQL) is more complex than on a cooperative one (Oracle). By our toolkit, all the details of a monitor can be hidden when users develop their monitors.

The toolkit is also adequate to the situation when the schema may be changed.

## 4.2. Query on the object-relational DBMS

In this subsection, we show how translator module processes the SQL over an object-relational database, Oracle8. As we see in Section 3, there are conflicts between the integrator and the database. Integrator sends the pure relational query to the translator module, but the Oracle8 may use additional features which the query cannot handle.

Therefore, we provide pseudo-tables for the integrator so that the semantic of the SQL applied on these tables can be easily retrieved.

### 4.2.1. The schema

The database contains four types: employee, department, location and phone type (Fig. 12).

*Employee type*. `dept` refers to the department object, which employee works in; `supv` refers to the supervisor object of employee type; `position` is a nested attribute, contains `building` and `city` attributes; `phone` is a set of `phone` type.

*Department type*. `mgr` refers to the manager object of employee type.

*Location type*. This type can hold office-information for employees.

*Phone type*. The `phone` attribute is the phone number of an employee.

### 4.2.2. Queries sent by integrator

Since the integrator uses relational schema, we map the object-relational schema into relational one (Fig. 12). Queries based on relational model will be translated into what can be processd by the information source, which uses an object-relational model. By mapping the schema and translating the query, we provide pseudo-tables and ability to use the SQL language which acts on these tables.

We use an example to demonstrate the processing of a query between integrator and information source. In this example, the integrator uses a relational model and the information source uses an object-relational data model. The sample database of an information source is shown in Fig. 13.

When integrator sends a query:

```
SELECT x.phone,c.name, b.name
FROM phone x, employee a, department b, em-
ployee c, location d
WHERE a.position_fk = d.oid and d.city =
'Taipei'
  and a.dept_fk = b.oid and b.oid =
  c.dept_fk
  and c.oid = x.fk
```

In this query, we retrieve employees whose office is located in Taipei. We retrieve the phone numbers and the

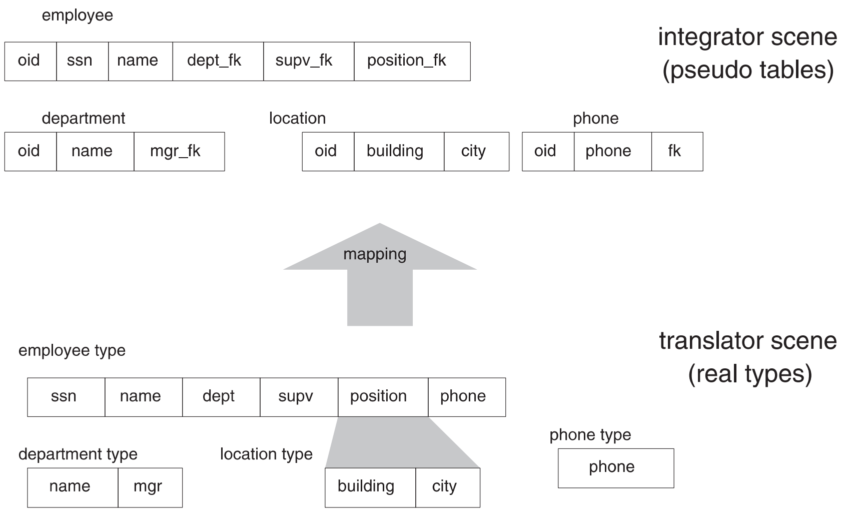


Fig. 12. Mapping schema.

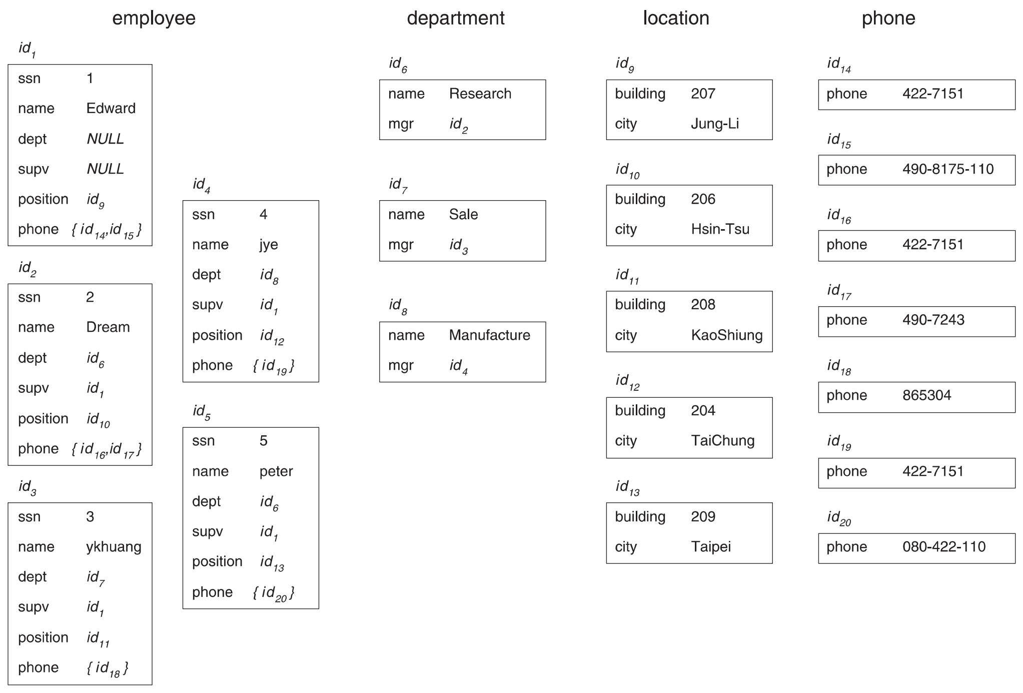


Fig. 13. Database instance based on the schema in Fig. 12.

name of employees who work in the same department. This query will be translated by the wrapper. Then, the wrapper sends the translated query to the information source. Finally, the result will be sent back to the integrator.

The results of the query are given below.

```
Result:
  422-7151
  Dream
  Research
Result:
  490-7243
  Dream
  Research
Result:
  080-422-110
  peter
  Research
```

From the results above, we find `peter`'s ($id_5$) office is in `Taipei` ($id_{13}$) and he works in `Research` ($id_6$) department. The other employee working in `Research` department is `Dream` ($id_2$). Now, all the phone numbers of `peter` and `Dream` will be retrieved.

### 4.2.3. Directly communicate with converter module

If an integrator and an information source use the same data model, the communication between them is straightforward and the translator module can be skipped. In this situation, the integrator can directly communicate with converter module. For this reason, the translator module and the converter module inherits the same interface.

For example, if the integrator can handle the additional features of Oracle8, query can be delivered to converter module directly. We use another example to show how to achieve it.

Consider the following query based on the relational model, and the query will be sent to translator module.

```
SELECT a.name, b.name
FROM employee a, department b
WHERE a.oid = b.mgr_fk
```

The query results are shown as follows:

```
Result:
  Dream
  Research
Result:
  ykhuang
  Sale
Result:
  jye
  Manufacture
```

The equivalent query of an information source based on object-relational model is as follows:

```
SELECT b.mgr.name, b.name
FROM department b
```

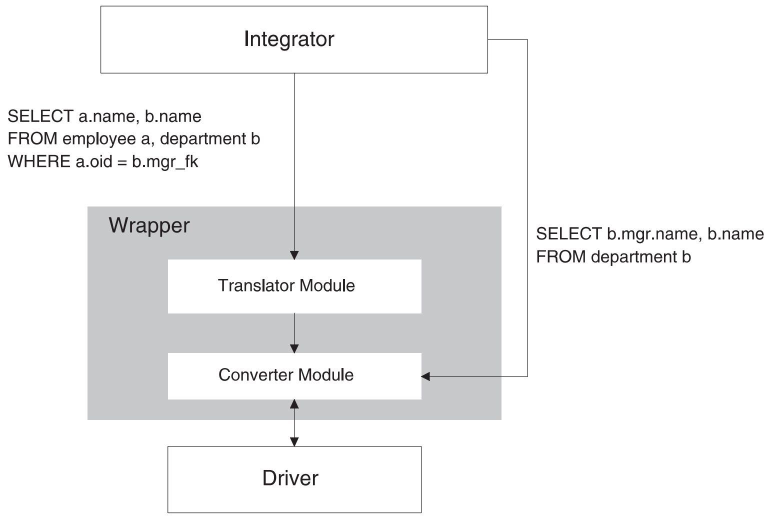


Fig. 14. Communicate with converter module.

The query is then sent to converter directly. That is, the integrator can directly communicate with the converter module and skip the translator module if integrator and the information source use the same version of database, e.g., Oracle8 (Fig. 14). We use the JDBC Drvier for Oracle7 now, so only a subset of the new features in Oracle8 can be provided via the converter module.

## 5. Conclusions and future works

In this paper, we first describe a modularized design for the wrapper/monitor in relational databases. We also show the architecture and functions of each module. Because we use Java to develop our system, a large part of code can be easily ported to another platform without re-compiling. We demonstrate how a monitor component works on a non-cooperative information source as correctly as a cooperative one. Besides the snapshot algorithm (Labio and Garcia-Molina, 1996), there are still other solutions to solve this problem. We create monitors and a wrapper for each site; every monitor is generated by the corresponding toolkit, which reduces a lot of onerous jobs. These components communicate with database via JDBC drivers. If a DBMS vendor does not provide the driver, we can write it by Java native interface (JNI), however this may reduce the feature of portability.

Next, we use several examples to show the flows of message passing when a system is initialized, update occured and query arrived. Because we use database as the information sources, we can use transactions to maintain the order among updates and queries. The sequence number will be sent to the integrator component, so integrator can determine whether an event is earlier than the others.

Finally, we demonstrate how a translator module works with integrator. An integrator can also communicate with the converter module directly, because both modules use the same interface.

Future work on our architecture includes developing monitors and toolkit on heterogeneous environments, designing a uniform interface between packager module and integrator component, enhancing the query capability and increasing the performance of our system. Our monitors can be run on relational databases only. Monitors on pseudo-tables are not ready in our approach, but this mapping job is similar to what translator does. In this paper, we just use `toString()` method to forward `Strings` to integrator. We can define an interface containing uniform methods. Every class returned to integrator will implement these methods. The packager module can invoke these methods to transform the representation; it will be easier to transform the data by this way. We just map a subset of the capability of Oracle8. The translator module can be easily extended by BYACC/Java and JLex, but the underlying driver should provide more support.

## Acknowledgements

A grant from the National Science Council, NSC882213E008011, partially supported this research. We are grateful to professor Baw-Jhiwne Liu for this contributions to this research. We would also like to thank our referees for their helpful comments and suggestions.

## References

Ashish, N., Knoblock, C.A., 1977. Wrapper generation for semi-structured Internet sources. SIGMOD Record 26 (4), 8–15.

Chang, Y., 1994. Interoperable query processing among heterogeneous databases. Technical Report 94-67, University of Maryland.

Chang , Y., Raschid, L., Dorr, B.J., 1994. Transforming queries from a relational schema to an equivalent object schema: a prototype based on F-logic. In: Proceedings of the International Symposium on Methodologies in Information Systems, 154–163.

Chang, C.-C., Trappey, A.J.C., 1996. A framework of product data management system – procedures and data model. Master's thesis, Department of Industrial Engineering, National Tsing Hua University, Hsinchu, Taiwan, ROC.

Chawathe, S.S., Garcia-Molina, H., Hammer, J., Ireland. K., Papakonstantinou, Y., Ullman, J.D., Widom, J., 1994. The TSIMMIS project: integration of heterogeneous information sources. In: Proceedings of IPSJ Conference, pp. 7–18.

Gruser, J.-R., Raschid, L., Vidal, M.E., Bright, L., 1998. Wrapper generation for web accessible data sources. In: Proceedings of the Third IFCIS Conference on Cooperative Information Systems (CoopIS '98) (also see ftp://ftp.umiacs.umd.edu/pub/louiqa/BAA9709/PUB98/CoopIS98.ps.

Hammer, J., Garcia-Molina, H., Nestorov, S., Yerneni, R., Breunig, M.M., Vassalos, V., 1997. Template-based wrappers in the tsimmis system. In: Proceedings of the 26th SIGMOD International Conference on Management of Data, pp. 532–535.

Hammer, J., Garcia-Molina, H., Widom, J., Labio, W., Zhuge, Y., 1995. The stanford data warehousing project. In: Proceedings of the IEEE Data Engineering Bulletin, vol. 18, no. 2, pp. 41–48.

Labio, W., Garcia-Molina, H., 1996. Efficient snapshot differential algorithms for data warehousing. In: Proceedings of VLDB Conference, pp. 63–74.

Liu, L., Pu, C., Tang, W., Buttler, D., Biggs, J., Benninghoff, P., Han, W., Yu, F., 1998. CQ: a personalized update monitoring toolkit. In: Proceedings of the ACM SIGMOD, May (also see http://www.cse.ogi.edu/DISC/CQ/papers/sigmod-demo.ps.

Papakonstantinou, Y., Gupta, A., Garcia-Molina, H., Ullman, J.D., 1995. A query translation scheme for rapid implementation of wrappers. In: Proceedings of the International Conference on Deductive and Object-Oriented Databases, pp. 161–186.

Papakonstantinou, Y., Garcia-Molina, H., Ullman, J.D., 1996. MedMaker: a mediation system based on declarative specifications. In: Proceedings of the IEEE International Conference on Data Engineering, pp. 132–141.

Papakonstantinou, Y., Garcia-Molina, H., Widom, J., 1995. Object exchange across heterogeneous information sources. In: Proceedings of the IEEE International Conference on Data Engineering, pp. 251–260.

Widom, J., 1995. Research problems in data warehousing. In: Proceedings of the Fourth International Conference on Information and Knowledge Management (CIKM), pp. 25–30.

Wiener, J.L., Gupta, H., Labio, W., Zhuge, Y., Garcia-Molina, H., Widom, J., 1996. A system prototype for warehouse view maintenance. In: Proceedings of the ACM Workshop on Materialized Views: Techniques and Applications, pp. 26–33.

Zhuge, Y., Garcia-Molina, H., Hammer, J., Widom, J., 1995. View maintenance in a warehousing environment. In: Proceedings of the ACM SIGMOD Conference, pp. 316–327.

Zhuge, Y., Garcia-Molina, H., Wiener, J.L., 1996. The strobe algorithms for multi-source warehouse consistency. In: Proceedings of the Conference on Parallel and Distributed Information Systems (also see http://www-db.stanford.edu/pub/papers/strobe.ps.

**Jorng-Tzong Horng** was born in Nantou, Taiwan, on 10 April 1960. He received the M.S. and Ph.D. degrees in Computer Science and Information Engineering from National Taiwan University, Taipei, Taiwan, in June 1986 and April 1993, respectively. He is currently an Associate Professor of the Department of Computer Science and Information Engineering at National Central University, Chung-Li, Taiwan. His current research interests include object-oriented database systems, distributed database systems, query processing and optimization, hypertext systems, genetic algorithms, and bioinformatics.

**Jye Lu** received the M.S. degree in Computer Science and Information Engineering from National Central University, Jungli, Taiwan, in June 1998. Now he is a staff of the Telecommunication Labs., Chunghwa Telecom, Yang-Mei, Taoyuan, Taiwan.