Object-Oriented Integration of Legacy Systems - Maintaining Global Integrity through Mediators

Andreas Loew, Thomas Kudrass

Technische Hochschule Darmstadt, FB Informatik, FG Datenverwaltungssysteme I
Frankfurter Str. 69 A, D-64293 Darmstadt (Germany)

E-Mail: {kudrass | loew}@dvs1.informatik.th-darmstadt.de

ABSTRACT

This paper presents an overview of the Persistence product, an objected-oriented interface to relational databases, that has been tested on the integration of multiple heterogeneous relational systems. We describe a sample active multidatabase application and propose extensions of the active properties offered by Persistence in order to use it as a mediator capable of managing global integrity constraints.

1. INTRODUCTION

Intelligent cooperative information systems require powerful tools to make existing applications interoperable. After several years of research, some commercial products have appeared on the market. Among them, we have evaluated *Persistence*[1], an object-oriented interface to relational database management systems (RDBMSs).

Global integrity maintenance between heterogeneous data repositories in a federation of legacy systems is a frequently discussed problem. Our aim is to utilize an active, object-oriented system for the implementation of an active mediator component that is used to achieve interoperability (mutual accessibility, a common query language and data model as well as global dependencies) between the heterogeneous local database systems and enforce all types of global semantic integrity constraints. In a first step, we want to prove the feasibility of this approach by implementing a prototype for this mediator based on *Persistence*, which is described in some more detail in the following section.

2. *PERSISTENCE*: AN OBJECT-ORIENTED INTERFACE TO MULTIPLE RELATIONAL DATABASES

Although object orientation comes to be the leading software development approach, being especially capable of integrating the heterogeneous data, tasks and systems by the concepts of inheritance and polymorphism, relational databases seem to remain the dominant data repository for the next couple of years. The mismatch between both data models must therefore be resolved, most likely by implementing an object-oriented database access layer (a kind of application programming interface - API) between its programming language and the relational database, a largely complex, time-consuming job. In particular, technical solutions for object identity, persistence, complex data types, the mapping of classes to tables, polymorphism and object oriented database queries must be found and the basic object relationships like inheritance, association and aggregation mapped to relational equivalents [HaTW95, Catt94].

Application specific C++ methods to create, read, update and delete instances (and the corresponding tuples), managing foreign key information to enforce referential integrity, can be implemented, but any major, subsequent change to the object model will lead to significant changes in the code of the affected classes. In addition, another critical issue is application reliability, as far as object data integrity, locking and transaction management are concerned.

[1] *Persistence* is a product of Persistence Software, Inc.

The *Persistence* product provides development tools that enable object oriented, active applications to use the well-understood, fast and reliable industry standard relational database management systems (RDBMSs), such as Sybase, Oracle, Informix or Ingres, as the storage base for their persistent data. *Persistence* consists of two major functional components: the *Relational Interface Generator* (RIG) which generates portable, database independent C++ classes mapped to relational data and the *Relational Object Manager* (ROM), implementing database access and maintenance (object integrity and transaction management) [KeJA94]. Building an application with *Persistence* involves a three-step, top-down process: defining the application object model, generating the C++ class interfaces using the *Persistence* RIG, and implementing the application code using the generated class interfaces and the *Persistence* ROM database access methods. (A bottom-up alternative where object model information is reverse-engineered from existing is described in section 3.3.)

Given the appropriate object model, the *Persistence* Relational Interface Generator creates a C++ class for each object in the model. Every class implements its own set of methods for database interface, such as create persistent or transient object instances, set or update object attributes or relationships, query using attribute values or ANSI SQL and delete objects from memory or database. Inheritance of attributes, methods and relationships, especially propagating superclass queries to subclasses, the usage of virtual methods to support polymorphism and inheritance from additional, user defined classes, is also supported. *Persistence* maps associations to foreign keys in the database, offering methods to access associated objects through the defined relationship, to ensure referential integrity between related classes and to specify delete constraints (delete propagation, set relationship to NULL, block). The application programmer can transparently navigate class associations or aggregations without knowing how they are implemented in the database.

The most important issues the Relational Object Manager has to address are enforcing object constraints, enabling object concurrency and providing high-performance access to relational data. It is critical that certain integrity constraints are enforced at object level: Objects may comprise information from multiple tables, and several copies of the same object may be held in memory at the same time. The ROM ensures that each object must be unique by registering each object read from the database and using smart pointers to allow several different objects to point to the same data.

When a transaction is committed, the data for all the objects in the object cache is flushed, the cache is cleared and all locks held in the database are released. But, to prevent the object in memory and its underlying data from differing, the smart pointers of all flushed objects are retained, so that the next time data of any such object is requested, the appropriate lock(s) are reacquired, and data is reread from the database (and cached). Methods implemented by the ROM include connecting to and disconnecting from a database, mapping classes to (possibly multiple) database connections (see section 3), beginning, committing and aborting transactions, setting savepoints and managing cache contents explicitly.

3. SPECIAL *PERSISTENCE* FEATURES: FOCUS ON...

3.1. Accessing Multiple Heterogeneous Databases At A Time

The key feature that predestinates *Persistence* for the implementation of a mediator is its flexibility as far as accessing multiple databases is concerned. *Persistence* does not only support database connections to Oracle, Sybase, Informix and Ingres RDBMSs one at a time, but through the design of its class hierarchy, each *Persistence* class may be mapped to a particular database connection and table - the mapping may be even changed dynamically while an application is running. For example, an application could open connections to an Oracle and a Sybase database, mapping the **`Customer`** class to an Oracle table while mapping the corresponding **`Account`** class to Sybase [KeJA94].

If the object model on which the generated classes of an application are based contains any relations (aggregations or associations) between two or more classes mapped to different databases at runtime, we will get some sort of a most simple "mediator" application, which enforces the appropriate constraints as interdatabase constraints between the participating databases.

3.2. Active Processing: Hooks

Persistence offers the possibility to define notification hooks on classes, attributes and relationships. A notification hook is a C++ method defined on the respective class which is called automatically whenever the relating event will take place next (*pre* hook) or has just taken place (*post* hook). Hooks are available on the creation, removal, query and modification of objects in a class, the change of an attribute's value or the change of a relationship or query of related objects.

When the *Persistence* RIG generates the code for an object model, empty stubs for each of the defined hooks are generated that can easily be extended to execute every piece of code the user implements for the method. In particular, it is of course possible to check whether a set of conditions holds before taking a specific action. The *Persistence* hooks can therefore be used to implement the functionality of the standard ECA (event, condition, action) rules [Daya88] or the triggers in relational systems, respectively. Although the coupling modes for simple C++ hooks are *immediate* (for example, the actions in an **`if`**-block are executed immediately after the condition has been checked), it is nevertheless possible to realize even the more complex *deferred* or *detached* coupling modes using threads and interprocess communication features.

3.3. Reverse Engineering of the Persistence Object Model: Persistence Dictionary Readers

While the ordinary, "top-down" usage of *Persistence* usually starts from a given object model, *Persistence* can also re-engineer (read) object model information directly from the RDBMS's data dictionary or system tables, allowing to automatically create C++ classes that correspond to the existing tables of, for example, legacy systems. In the evaluated release of *Persistence* (2.354), an officially supported dictionary reader exists only for Sybase. An (at least currently) unsupported, preliminary version of an Oracle reader is delivered with *Persistence*, and the release of an Informix dictionary reader is just forthcoming.

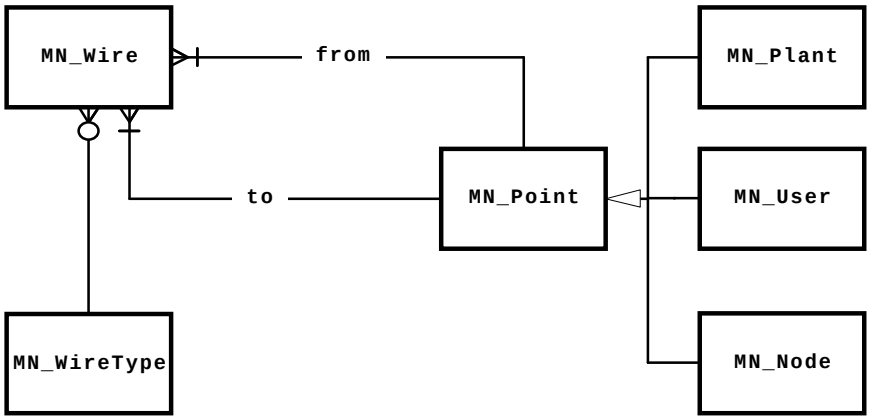
The data dictionary reader filters information in the RDBMS's system tables into a *Persistence* project structure: For all user tables in the database, a corresponding *Persistence* class with the table's columns as attributes of the appropriate data types is created. Primary key information is used to determine the *Persistence* key object definition, and - where the RDBMS supports it - foreign key information to set up relations between the participating classes. Unfortunately, the current version of the Sybase dictionary reader is not able to determine the correct cardinalities for the resulting relations: Regardless of e.g. **`unique`** or **`not null`** definitions and intermediate tables, every relation is set up to be a zero-or-one (source class) to zero-or-many (destination class) relationship. (For example, each **`Department`** (source class) can be related to zero to many **`Employee`**s, and each **`Employee`** can be related to zero or exactly one **`Department`**.) If the foreign key in the RDBMS table is allowed to be NULL, the data dictionary reader sets the default delete action for the destination class to *remove*, which results in simply removing the relation when a related object is to be removed; if the foreign key is not allowed to be NULL, the delete action will be to *propagate* the removal and delete the formerly related object, too.

4. *PERSISTENCE* AS AN ACTIVE MEDIATOR SYSTEM

4.1 A simple mediator example application

As a first evaluation example for active multidatabase access through *Persistence*, we decided to choose the example presented in [CeWi93]. Data describing power distribution networks - power supplies, transportation and usage - is stored in two different relational databases, potentially managed by different RDBMSs.
The first database, called MN (*Milano_Network*), implements the regional power network for the area of Milano (Milan). It stores data about the power produced by several plants, required by several users and the power loss at each intermediate node. Directed wires exist, connecting plants, users and nodes, each of a certain type and carrying a certain voltage and power. The second database, IN (*Italy_Network*), contains national-wide data about the power network throughout Italy. Plants, users and intermediate nodes are represented in the single table **`IN_Node`**, distinguished by the value of the attribute function (**`'p'`**, **`'u'`** or **`'n'`**, respectively). The (simplified) relational schemes and their "translation" to the equivalent object model supported by *Persistence* are shown in figure 1.

```
MN_Plant(id, location, power)
MN_User(id, location, power)
MN_Node(id, location, power)
MN_Wire(id, from, to, type, voltage, power)
MN_WireType(type, max_voltage, max_power, cross_section)
```




```
IN_Node(id, region, location, function, power)
IN_Connection(id, from, to)
IN_Wire(id, connection_id, type, voltage, power, age)
IN_WireType(type, max_voltage, max_power, wire_size)
```

Figure 1: Relational schemes and Persistence Object Model for databases IN and MN

We decided to introduce an additional (virtual) superclass into the object model, called **`MN_Point`**, comprising the plant, user and node classes in database MN, in order to model an equivalent to the **`IN_Node`** class. In our test environment, the *Persistence* database connections for the **`IN_`** and **`MN_`** classes are set to a Sybase or Informix connection, alternatively, as described in section 3.1. Clearly, it is desirable that the data describing the power network in the Milano region is consistent within the two databases.

For that reason, Ceri and Widom [CeWi93] introduced two notions of inter-database dependencies: *existence dependencies*, where the presence of data in one database implies that related data must be present in another, and *value dependencies*, where the value of data in one database determines (in some way or another) the value of related data in another database. Additionally, a database dependency is either *directional*, so that values in one database are treated as primary (or source of the dependency relation), while values in the other are treated as secondary (destination of the dependency relation) and are subjected to changes because of the dependency rules, or *nondirectional*, describing mutual interdependency.

We implemented the example inter-database dependency rules proposed in [CeWi93] using the *Persistence post* hooks described in section 3.2, and tested the correctness of the implementation using various examples. As an example, the hook implementation for a simple directional inter-database value dependency is shown in figure 2:

Directional Value Dependencies ([CeWi93]):

```
IN_Wire.voltage ⇐ select 'low'
                  from MN_Wire
                  where MN_Wire.id = IN_Wire.id and
                        MN_Wire.voltage < 15000
IN_Wire.voltage ⇐ select 'high'
                  from MN_Wire
                  where MN_Wire.id = IN_Wire.id and
                        MN_Wire.voltage >= 15000
```

Implementation of **`MN_Wire::didChangeVoltage()`** hook (abridged):

```
void MN_Wire::didChangeVoltage(int newValue) {

  IN_Wire *corrIN_Wire;

  corrIN_Wire = IN_Wire::queryKey(this->getWire_id());

  if (corrIN_Wire == NULL) {
     cerr << "FATAL: Inconsistency in database: Wire " << this->getWire_id() << "
not in IN_Wire" << endl;
    exit(1);
  }

  if (newValue < 15000) {
    if (strcmp(corrIN_Wire->getVoltage(),"low") != 0) {
        cout << "\tPropagating change to database IN: setting IN_Wire.voltage =
\"low\"" << endl;
      corrIN_Wire->setVoltage("low");
    }
  } else {
    if (strcmp(corrIN_Wire->getVoltage(),"high") != 0) {
        cout << "\tPropagating change to database IN: setting IN_Wire.voltage =
\"high\"" << endl;
      corrIN_Wire->setVoltage("high");
    }
  }

  delete corrIN_Wire;

}
```

Figure 2: Directional value dependency and Persistence `didChangeVoltage()` hook implementation

4.2 Mediators for General Databases: The Concept of a "Mediator Generator"

The *Persistence* "top-down" approach proceeds on the assumption that every interaction between the end user and the relational database is done exclusively through applications based on the generated C++ class interface, which is responsible for the mapping to equivalent commands. This is particularly true for database integrity maintenance: The only active features detected are calls to user defined C++ hook methods. By way of contrast, in a database federation we have to allow for local updates (for example through existing applications or even dynamic SQL frontends) which may also violate global interdatabase consistency conditions.

For that reason we plan to implement a database independent *mediator generator*, a tool that will take a *Persistence* object model - which can, in turn, be generated by a *Persistence* dictionary reader as described in section 3.3 - and output the C++ code for a SQL-to-*Persistence* translator background process. Linked with the *Persistence* generated classes for the object model and a set of user defined C++ hooks as described in section 3.2, this demon process shall take SQL DML commands executed against a specific RDBMS's SQL server - intercepted and preprocessed by a special RDBMS specific gateway [KuKr95] - as input, map them to the

equivalent calls of the *Persistence* C++ **`create()`**, **`remove()`**, **`get()`** and **`set()`** methods, and use *Persistence* functionality to maintain global database consistency.

One important restriction remains: any modification of the underlying database scheme (or object model) requires a regeneration of the mediator C++ source and recompilation, so dynamic SQL DDL statements can not be handled satisfactory. On the other hand, for the (hopefully) very stable and static legacy systems, this is not as bad as it may seem at first view.

5. CONCLUSION AND FURTHER WORK

While there remain a lot of yet unsolved questions as far as the precise design of the automatic C++ mediator generator is concerned - in fact, the design phase for the code generator has just been started - , evaluation results for the *Persistence* product are encouraging: The implementation of the complete Italian network example application has been completed in less than a one-man month, and *Persistence* itself (the RIG as well as the ROM and the generated classes) has shown to be highly reliable. Furthermore, the effects of the additional overhead for the object oriented interface on application execution time are neglectable (or even unlikely to be detected at all). Work will go on to implement the database model independent mediator and manage to intercept the SQL commands in a manner similar to that of the existing Sybase gateway for Informix and possibly other RDBMSs.

The evaluation of Persistence and the implementation of the mediator prototype is embedded in the REACH (*RE*al-time *AC*tive *H*eterogeneous system) project at TH Darmstadt [BZBW95]. The REACH system provides rules as objects of an OODBMS, a powerful event composition algebra and coupling modes which permit a flexible expression of execution models. While, in this paper, we have focused on the issue of global integrity maintenance, REACH goes beyond that by applying rules to a broader range of applications, such as workflow management or access control. It will be examined to what extent the already completed REACH implementation can be migrated to *Persistence*.

REFERENCES

[BZBW95] A. Buchmann, J. Zimmermann, J.A. Blakeley, D.L. Wells, *Building an Integrated Active OODBMS: Requirements, Architecture and Design Decisions*, in: Proc. 11th Intl. Conf. on Data Engineering, Taipeh, 1995.

[Catt94] R.G.G. Cattell: *Object Data Management - Object-Oriented and Extended Relational Database Systems, Revised Edition*, Addison-Wesley, 1994.

[CeWi93] St. Ceri, J. Widom: *Managing Semantic Heterogenity with Production Rules and Persistent Queues*, Proc. 19th VLDB Conference, Dublin, 1993.

[Daya88] U. Dayal, *Active Database Management Systems*, Proc. 3rd Intl. Conf. on Data and Knowledge Bases, Jerusalem, 1988.

[HaTW95] W. Hahn, F. Toenniessen, A. Wittkowski: *Eine objektorientierte Zugriffsschicht zu relationalen Datenbanken* (in German), Informatik Spektrum 18 (1995), 3.

[KeJA94] A. Keller, R. Jensen, Sh. Agarwal, *Enabling the Integration of Object Applications with Relational Databases*, *Persistence* Technical Overview, *Persistence* Software, Inc., 1994.

[KuKr95] Th. Kudrass, R. Krayer, *Active Mechanisms in Interoperable Systems*, TH Darmstadt, FG Datenverwaltungssysteme I, 1995 (submitted to the RIDE-NDS Workshop, New Orleans, Feb. 1996).