Scientific Bulletin of the „Petru Maior" University of Tîrgu Mureş
Vol. 10 (XXVII) no. 1, 2013
ISSN-L 1841-9267 (Print), ISSN 2285-438X (Online), ISSN 2286-3184 (CD-ROM)

A MODEL FOR USER INTERFACE DESIGN IN DATABASE-DRIVEN INFORMATION SYSTEMS

Marius MUJI
Petru Maior University of Tirgu Mures
Nicolae Iorga street, no. 1, 540088, Tirgu Mures, Romania
[1]marius_muji@yahoo.com

Abstract

Current technologies employed for user interface development in database-driven information systems have a procedural approach – they need to specify HOW the business rules should be enforced, through a sequence of operations on data (e.g., create, retrieve, update, delete). Therefore, the development process is time-consuming and error prone. This paper proposes a logical data model – inspired by and complementary to the relational model – which allows a declarative approach in application development. We also present an example which specifies, in terms of the proposed model, WHAT data integrity constraints should be (automatically) enforced be the system.

Key words: Logical design, data models, declarative specifications

1. Introduction

A Database-driven Information System is a generical name for any information system built around an *integrated* and *shared* source of data, i.e., a database [1], and whose basic functions can be reduced, conceptually, to the following simple operations on data: create, retrieve, update, delete (CRUD).

Actually, almost all the Information Systems that we use daily are database-driven. From the so called ERPs (Enterprise Resource Planning systems), which ensures the data integration, through a common database, for large organizations, to any virtual store, all these systems are, basically, database-driven, regardless the various technologies employed for their development.

Architecturally, besides the database itself, these systems contain a presentation component, which exposes the appropriate data for every particular (category of) user(s). When the system provides information to a large community of users, there will be many different *user views* [2] (UV) on the system's data.

The technology of choice for the integrated, central, component of the system (i.e., the database), are the Database Management Systems, which follow the same theoretical model: "the relational model", proposed in 1969 by E.F. Codd [3]. This is a rigorous, mathematical model which provides the conceptual foundation for the most important database technologies.

On the other hand, the presentation component, composed by all the *user views* of the system, is developed using programming languages like C, Java, Delphi, etc. Those languages do not benefit from the same theoretical support as the database technologies. This is why the real world "business rules" related to the "presentation data" cannot be expressed directly, in a *declarative* manner, as part of the data structures' definition - like in the database case. The nature of the employed technology forces a *procedural* approach, in which the business rules are expressed through a sequence of elementary CRUD operations on data. Even though the great majority of programming languages have raised substantially their level of abstraction in the last decades, they are still procedural in nature, so that at a certain point in the development process, somebody still needs to write (procedural) code, in order to express all the required business rules of the system.

There is a plethora of presentation level (i.e., user interface) development technologies on the market, with the main objective of reducing the amount of code (programming work), through various declarative facilities, which allow the subsequent automatic generation of the procedural code. None of them provide a complete solution for the objective of "code elimination", due to the previously stated reason: the lack of a rigorous theoretical support.

This paper proposes a logical data model for the declarative specification of the presentation level in database-centric systems. As for any logical data model [4], we need to define the *data structures*, the *operators*, and the *integrity constraints* used to represent and to manipulate data in a consistent

manner. The corresponding definitions are covered respectively by the sections 2, 3, and 4 , while section 5 provides an example for the model usage in a common practical situation. Section 6 concludes and present some future projects.

2. Data Structures

There are two important requirements for data collections that need support at the presentation level: *ordering* and *current* position. The second relies on the first, and both are incompatible with the set theory (and thus with the relational model). However, these are the only *essential* data definition requirements needed at the UV level and not supported by the relational model.

On the other hand, the essentiality of the data model, i.e. the existence of a unique data constructor, is one of Codd's great ideas, and it should be considered for any data model definition.

The presentation model uses the ***array of tuples*** [5] as the only data constructor. The array is defined as an ordered pair, with the second element being a table which has a mandatory column, *seq_no*, and the first element indicating the *current element* in the sequence.

3. Operators

Since the only UV data structure is the *array,* we will need some ***array operators***. In order to take advantage of the power of the relational algebra, and also to eliminate the impedance mismatch with the RDBMSs data structures, it is necessary to define operators which perform a transformation from relations to arrays and vice-versa. Consequently, the presentation model needs two categories of operators:

1. Array operators;
2. Relational operators (relational algebra).

The array operators are defined as follows:

- *Cardinality* – returns the cardinality of the array's table;
- *Extract Attribute Val* – returns the value of a specified attribute of the current tuple of a specified array (if the array is 'empty', it will return a default value);
- *Extract Current Tuple* – self explained;
- *Get Cursor* – returns the sequence number of the current tuple;
- *Set Cursor* – changes the current position of the cursor in the specified array;
- *Array-to-Table* - extracts the second element of the ordered pair which defines the array;
- *Table-to-Array* – transforms a table into a set of arrays.

Note that, in the case of *Table-to-Array* operator, the cardinality of the obtained set of arrays is determined by the specified ordering criteria, and it can take a value from one to the cardinality of the considered table. At implementation, there is always a possibility to reduce to one this value, using the physical order of the table's tuples.

4. Integrity Constraints

There are two categories of integrity constraints which needs to be defined for a user view. The first category of constraints are meant to enforce the consistency of the UV data, as if the UV where isolated from the database. Their definition is almost identical with the definition of the database constraints [6]. Thus, we will have:

- *Attribute constraints* – contained in the *characterization of each array structure;*
- *Tuple constraints* – contained in the definition of the *tuple universes;*
- *Table constraints* – contained in the definition of the *table universes;*
- *User view constraints* – contained in the definition of the *user view universe;*
- *User view state transition constraints* – contained in the definition of the *user view's state transition universe.*

The second category of constraints should keep *the entire system* (DB + VU) consistent, enforcing the semantic synchronization between the user view and the database. This synchronization implies bi-directional *transformations/mappings* between the data structures defined at the database level and the data structures defined at the UV level. Thus, we will have:

- *System update constraints* – contained in the *update transitions universe of the system*;
- *User view refresh constraints* – contained in the *refresh transitions universe of the system.*

These mapping constraints are the key ingredient of the presentation model: they facilitate the declarative development and the automation of the User Interface.

5. Example

The following example is inspired from the chapter about presentation rules in reference [7].

Let us consider a *User View* that presents to the end user data about customers, orders, and order details. The user should have to be able to see at any time all the customers located in a specific region which have a credit limit less than a certain value. The displayed customers should be ordered by name, by credit limit, or by the total value of their orders (as indicated by the user).

Likewise, the user should be able to see the orders which belong to the current customer and their issuing date is in a certain period (say, after a start_date and before an end_date, specified by the user). The user may also choose the ordering sequence of the respective orders: by date, value-ascending, or value-descending.

When the user inspects a specific order, the system should provide all the order_details that belong to that particular order. The user should also

be able to track the current order's history and to insert a new status for the respective order.

Figure 1 shows an entity-relationship diagram [8] [9] of the *arrays* (represented by boxes) of our sample *User View*, end their mutual *relationships* (represented by arrows).

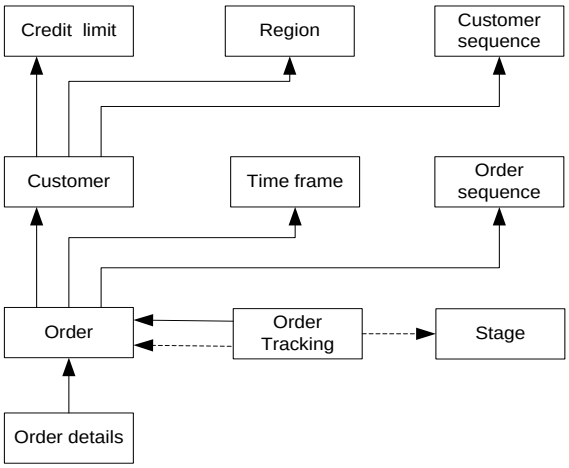

Fig. 1: The *user view* example in E-R representation

Each of these arrows indicates a *dependency relationship* between two arrays. There are two categories of *dependency relationships* [10] with graphical representation: the first, associated with the *refresh constraints*; the second, associated with the *update constraints.*

The dependency relationships associated with *refresh constraints*: the database queries specified for the refresh constraints of the dependent (children) arrays need some parameters, whose values are taken (at run time) from the current tuple of the referenced (parent) arrays. Since the entire content of the child array depends on the current tuple of the parent array, the dependency relationships of this category are *many-to-one relationships*, and have the same importance at the user view level, as the *foreign keys* at the database level. This category of relationships are used (at the physical/implementation level) to build the dependency graph which guides the automatic execution of all the database queries needed to refresh the entire user view. The graphical representation is done by a plain arrow, oriented from child to parent. Self reference relationships are not allowed, neither do cycles in the dependency graph.

The dependency relationships associated with *update constraints*: the database transactions specified for the update constraints of the dependent (children) arrays need some parameters, whose values are taken (at run time) from the current tuple of the referenced (parent) arrays. This category of relationships presents a secondary importance, compared with the first category, and their graphical representation is done by a dashed arrow, oriented from child to parent. Self reference relationships of this kind are allowed (being actually the most common), but do not have a graphical representation.

In terms of the proposed presentation model, using an SQL-like syntax, the *User View Schema* [11] can be specified as follows:

```
CREATE ARRAY Credit_Limit
(
Value          money_dom      DEFAULT 0
);
CREATE ARRAY Region
(
Region_ID              id_dom,
Region_Name            region_dom
CONSTRAINT region_refresh
      REFRESH_ARRAY get_region()
)
```

We assume that get_region() is a function without parameters, which returns a relation (table) with the same header as the array Region (it might be an SQL View, a stored procedure, or any other function, written in any language, which satisfies the above conditions).

```
CREATE ARRAY Customer_Sequence
(
Code           cust_seq_dom,
Description    descr_dom
CONSTRAINT customer_sequence_refresh
      REFRESH_ARRAY get_cust_seq()
)
```

We assume that get_cust_seq() is a function without parameters, which returns a relation (table) with the same header as the array Customer_Sequence (more specifically, the returned relation will have three tuples, corresponding to the required options: 'name', 'credit limit', and 'total value', e.g. { {'N', 'name'}, {'C', 'credit_limit'}, {'T', 'total value'} }).

The entities *Customer*, *Time_Frame*, *Order_Sequence*, *Stage*, and *Order*, respectively, have similar definitions. The logical schema for the entities *Order_Details* and *Order_Tracking* is specified as follows.

```
CREATE ARRAY Order_Details
(
Product_name           prod_name_dom,
Quantity               quantity_dom,
Unit_price             money_dom,
Value                  money_dom

CONSTRAINT order_detail_refresh
      REFRESH_ARRAY
      get_order_details(Order.Order_ID)
)
```

We assume that get_order_details() is a function (e.g., a stored procedure, a table-valued function) with one parameter (the order ID), which returns a relation (table) with the same header as the array *Order_Details*. At run-time, the enforcement of the refresh constraint will determine the execution of *get_order_details()*: its argument will take the value of the attribute Order_ID from the current tuple of the array Order.

```
CREATE ARRAY Order_Tracking
(
Date                    date_dom,
Stage                   stage_name_dom,
Comments                comments_dom

CONSTRAINT order_tracking_refresh
      REFRESH_ARRAY
      get_order_tracking(Order.Order_ID);
CONSTRAINT order_tracking_insert
      INSERT_TUPLE insert_order_tracking(
            Order.Order_ID,
            Stage.Stage_ID,
            Order_Tracking.Date,
            Order_Tracking.Comments)
)
```

We assume that insert_order_tracking() is an update procedure (e.g., a stored procedure) with four parameters: the order ID, the stage ID, the DATE in which the order reaches the stage with the specified id, and the optional comments associated with that stage of the considered order. At run-time, the enforcement of the INSERT TUPLE constraint – triggered by any insert of a tuple in the array Order_Tracking – will determine the execution of insert_order_tracking(): the first argument will take the value of the attribute Order_ID from the current tuple of the array Order; the second argument will take the value of the attribute Stage_ID from the current tuple of the array Stage; the third argument will take the value of the attribute Date from the current (inserted) tuple of the array Order_Tracking; the fourth argument will take the value of the attribute Comments from the current (inserted) tuple of the array Order_Tracking. If the insert will be rejected due to some violated database constraints, or to any business rule implemented at the insert_order_tracking() procedure level, the tuple insert should be rejected for the Order_Tracking array, as well. If the insert is accepted at the database level, the refresh constraint should be automatically enforced, causing the execution of the get_order_tracking() procedure.

6. Conclusions

Current practice in information systems development promotes technologies and languages which strive to provide a high level of abstraction, through nowadays popular approaches like the OMG's Model Driven Architecture [12] [13]. Our logical data model provides the conceptual foundation for declarative technologies that allows for a *data-model driven approach* in application development [14] [15] [16]. Declaring business rules in terms of data integrity constraints, their enforcement can be automated, eliminating the procedural code. This approach can lead to important changes for the entire industry, like:

- a dramatic reduction of the development and maintenance costs;

- the transformation of the *programmers* in *data designers* - switching the focus from *how* the system is developed to *what* data should be exposed to the user;

- eventually, the "democratization" of the Information Systems - allowing anyone to develop a personal-purpose database-driven system, without need for any programming skills.

Future work will concentrate on model implementation in various technologies. A common data description language (DDL) will also be specified, in order to provide a seamless integration of the implementation technologies, through a common metadata dictionary.

References

[1] C. J. Date, *An Introduction to Database Systems (8th edition)*.: Addison-Wesley, 2003.

[2] ANSI/X3/SPARC Study Group on Data Base Management Systems, "Interim Report," 1975.

[3] E. F. Codd, "A relational model for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377-387, 1970.

[4] Edgar F. Codd, "Data models in database management," in *The 1980 workshop on data abstraction, databases and conceptual modeling*, New York, February 1980.

[5] C. J. Date and Hugh Darwen, *Foundation for Future Database Systems: The Third Manifesto (2nd Edition)*.: Addison-Wesley, 2000.

[6] Lex de Haan and Toon Koppelaars, *Applied Mathemathics for Database Professionals*.: Apress, 2007.

[7] C. J. Date, *What Not How: The Business Rules Approach to Application Development*.: Addison-Wesley, 2000.

[8] Peter Pin-Shan Chen, "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9-36, 1976.

[9] David C. Hay, *Data Model Patterns: A Metadata Map*.: Morgan Kaufmann, 2006.

[10] C. J. Date, *Logic and Databases: The Roots of Relational Theory*.: Trafford Publishing, 2007.

[11] David C. Hay, "Data Model Views," *The Data Administration Newsletter - TDAN.com*, April 2000.

[12] OMG. (2000) MDA Specifications. [Online]. http://www.omg.org/mda/specs.htm

[13] Object Management Group. (2009) UML Resource Page. [Online]. http://www.uml.org/

[14] Bill Lewis, "Data-Oriented Application Engineering: An Idea Whose Time Has Returned," *The Data Administration Newsletter - TDAN.com*, January 2007.

[15] Bill Lewis, "Data-Driven Molecular Specifications, Part 1," *The Data Administration Newsletter - TDAN.com*, October 2010.

[16] Bill Lewis, "Data-Driven Molecular Specifications, Part 2," *The Data Administration Newsletter - TDAN.com*, December 2010.