The Impedance Mismatch in Light of the Unified State Model

Piotr Wiśniewski, Marta Burzańska, Krzysztof Stencel

Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Toruń Poland
pikonrad,quintria,stencel@mat.umk.pl

Abstract. In this paper the authors discuss the misunderstanding that arose over the years around the broadly defined term of the object-relational mismatch. It occurs in many different aspects of programming. There are three concerns considered the most important: mismatching data models, mismatching binding times and mismatching object lifecycle. This paper focuses on the data model mismatch. The authors propose a theoretical approach to this mismatch through the introduction of the common state theory, which is a proposal of a unified language for the models of objects of popular programming languages as well as for basic database models. Using this theory and the discussion around it, the authors show that the mismatch of data models is no longer ans issue.

1 Introduction

In 1970 E. F. Codd described the relational data model in his seminal paper [1]. In a few years after that relational database management systems appeared at the software market. Parallel to the increased development of the RDBMSs, the programming language community developed and adopted object-oriented techniques. Then, it has turned out that the OO methods encounter nontrivial difficulties with the persistent data storage. Relational databases that were used to persist objects, came with a number of concerns known under a collective term *impedance mismatch* [2]. Those problems manifest themselves as (1) the mismatch of the data models - there is no single common data model to express the differences between programming entities and the relational model; (2) the mismatch of the binding time - programs are usually compiled and all the names are bound at the compile-time, while database entities are late-bound at the run-time; (3) the mismatch of the lifecycle - program's variables are destroyed together with it, while database entities survive a program's termination. The impedance mismatch is not only related to relational databases, but it also occurs when trying to connect OO programming languages and other database systems like XML documents [3].

In this paper we focus on the mismatch of models. Although the relational data model is based on the relational algebra, in practice it is implemented using multi-sets contrary to the original Codd's idea [4]. On the other hand, the data model in object-oriented programming languages is based on a less formal approach. Especially, the data model of classic programming languages like Pascal, C and C++ has a feature that prohibits smooth mapping to flat relational structures, namely the unlimited nesting of objects. We will elaborate on the mismatch of C++ and SQL models in section 2.

Important attempts to address those mismatches were the first-generation object database standard developed by ODMG [5] and the AS0 model that became the seed of the second generation standardization [6]. Those were just exemplary models that *possibly* could address the issue. However, in our opinion the sole model is not enough. One must also provide mappings of popular models onto the platform of a common data model. Without it, any discussion is void. In order to present an appropriate solution in section 3 we introduce the *Unified State Model* to describe the states of the modeled objects. Subsequent sections show how this model can be used to define the data models of various programming languages and databases. Especially, section 5 shows the first formalization of the data model AS0 described in [7] and a number of Subieta's papers.

2 Mismatch of the mapping between C++ and SQL

The problem of impedance mismatch between relational databases and programming languages has been around since the very beginning of the relational databases [8]. With the dawn of the object-oriented programming this problem became more apparent and was given a specific name of the object-relational (O/R) impedance mismatch. When asked, people associated with programming, have very different views on this problem. Some say it is a trivial matter, other say that it is a bottleneck for modern software development. To fully understand this problem we have to realize that it is actually a full set of problems among which some are fairly easy to overcome, and some pose more difficulties.

The first problem lies with the direct mapping of data types, objects, classes and inheritance. For some data types a database may require length constraint whereas programming language does not. Some data types in one of the worlds may allow for bigger values than in the other world. Also, the concept of public and private fields or methods is difficult to map onto the database. However, what is mostly brought up during the discussion on the impedance mismatch is the problem of inheritance, which is not supported by relational databases. There are also voices that by mapping objects into the database a programmer may stumble upon problems with checking objects identity and equivalence.

Other issue is connected with the structure of objects. In the OO paradigm an object may be composed of other objects and methods, which may be modeled with the use of pointer. In the pre-SQL-1999 RDBMS data model complex objects cannot be directly stored in a single table, and also the pointers are not allowed.

Also the mechanism for data retrieval may pose some problems. In the OO paradigm a programmer uses some form of object methods to gather information about it. Currently, there are three main approaches to map SQL's retrieval mechanisms onto the OO paradigm: Query-by-Example (QBE), Query-by-API

(QBA) and Query-by-Language (QBL). In the QBE approach a programmer has to fill out some template object of the type he wishes to acquire. However, while it is an efficient method for simple data extraction, it is insufficient for more complex queries. QBA queries are constructed using special query objects or methods. The predicates are written in a form of arguments for query functions (see Python's SQLObject syntax for examples). In this form of notation it is hard to include more advanced aspects of queries like outer joins, subqueries etc. QBL approach offers yet another SQL-like language implemented for (and similar to) the host programming language. However, this approach is only a direct translation of SQL's syntax into a more user-friendly style and does not offer actual object-oriented approach. And for all three approaches there is always the problem of Null values and three-valued logic used by RDMBSs.

Completely different aspect of the O/R impedance mismatch is dealing with user privileges and authorization. By mapping database tables onto objects a programmer may lose information on specific restrictions placed on the schema by database administrator.

The last case is the issue of object manipulation and transactions. When we acquire an object from the database we actually acquire its copy on which we work. The same object may be retrieved by others and also modified by them while we still have the old "out-of-sync" version.

There are, of course, other, smaller problems coming from those briefly mentioned above. Some of them are easy to bypass through considerate choice of programming style and data model for objects. Other aspects may require other means. The key problem here is to find a way to overcome this whole set of issues in one coherent approach that is less costly and less painful than those problems.

3 Unified State Model

In this section we introduce basic definitions of the Unified State Model (USM).

Definition. Let us use the following symbols:

- Let V be a set of atomic values.
- Let $N \subset V$ be a set of external names with chosen value $\eta \in N$.
- Let $I \subset V$ be a *set of identities* with chosen value $\mathbf{nil} \in I$ and assumption $N \cap I = \emptyset$.

For any set S we denote by $S^* = \bigoplus_{t \in \mathcal{N}} S$, i.e. the set of all finite sequence of elements of S.

Definition. *State of an object of level* 0 is a triple $(i, n, v) \in I \times N \times (V \cup I)$. By O_0 we mean the set of all states of level 0. This set is also denoted by S_0.

Next we introduce an induction definition of states of objects of level t for a natural number t.

Definition. Let $t \in \mathcal{N}$ be a positive natural number and assume that we have defined a state of object of the level k for all $k < t$ and assume that S_{t-1} denote the set of all states of object of the level lesser than t. A state of object of the level t is a triple $(i, n, L) \notin S_{t-1}$ where $i \in I, n \in N$ and $L \in S^*_{t-1}$ is a finite sequence of elements of S_{t-1} . A set of all states of objects of level t we denote by O_t and by $S_t = O_t \cup S_{t-1}$ we denote a set of all states of object of the level not grater than t.

Let us denote by S_∞ set of all states of objects of all natural levels. Let us denote $Ref(o = (i, n, x)) = i$ for any $o \in S_\infty$. Let us introduce an inductive definition of a function $\mathcal{S} : S_\infty \to 2^{S_\infty}$ as follow:

a. If $o \in O_0$ we put $\mathcal{S}(o) = \{o\}$.
b. Let us assume that $t > 0$ and we have defined $\mathcal{S}$ for all $o' \in S_{t-1}$ and let $o = (i, n, (o_1, ..., o_k)) \in O_t$. We define

$$\mathcal{S}(o) = \{o\} \cup \bigcup_{j=1}^{k} \mathcal{S}(o_j).$$

Another function we need is $\mathcal{I} : S_\infty \to 2^I$ defined as follow:

$$\mathcal{I}(o) = \bigcup_{o' \in \mathcal{S}(o)} \{Ref(o')\}.$$

1 Remark. *For any $o \in S_\infty$ the sets $\mathcal{S}(o), \mathcal{I}(o)$ are finite.*

A state of an object $o \in S_\infty$ is called *proper*, if:

a. $o \in O_0$ or
b. $o = (i, n, (o_1, ..., o_k)) \in O_t$ for some $t > 0$, all o_j are proper and

$$Ref(o) \notin \mathcal{I}(o_j), \text{ for } j = 1, ..., k \text{ and}$$

$$\mathcal{I}(o_i) \cap \mathcal{I}(o_{j'}) = \emptyset, \text{ for } j, j' = 1, ..., k; j \neq j'.$$

This means that in a proper state no identity is repeated. The set of all proper state is denoted by $\mathbf{S} \subset S_\infty$.

4 Relational Model

The relational model has been defined in [1]. However, the mathematical definitions originating from the set theory that were used in this paper do not fully correspond to the real model implemented in the RDBMSs. It soon became obvious that the classic definition of a set is too troublesome to be used to describe the result of a query. Because the projection of data on a hiperplane (in terms of a database language — not the mathematical terminology — meaning the selection of certain columns) often results in repeated tuples. From the point of view of a data model based on the set theory such results should be presented as only one tuple. However, in practical applications tuples tend to be repeated. Attempts to resolve this problem have led to the description of the relational algebra in the terms of the multisets. The following work focuses on the practical data models, thus when discussing the relational model as a starting point we have chosen the relational algebra based on multisets in terms of the Grefen's paper [4].

In this section we model the terms from the mentioned paper based on the USM. The terms *domain* and *relation's schema* are meta-terms and thus do not need to be modeled in USM. Let us start with the definition of a relational tuple and then with the definition of the state of a relation.

First we will remind the definition of a domain and a tuple described in the aforementioned paper:

Definition. Domain of a relation A is a subset $\mathcal{A} \in \mathcal{V}$. Domain $\mathcal{A}$ is also denoted by $dom(A)$. The relational schema, also called the type of a relation, is a set $\mathcal{R}$ denoted by $dom(R)$ such that $dom(R) = dom(A_1) \times ... \times dom(A_n)$.

A tuple of a relation of type $dom(R)$ is a state of an object $o = (i, R, o_1, ..., o_n)$ such that for $k = 1, ..., n$ $o_k = (i_k, A_k, v_k)$ is an atomic object such that $v_k \in dom(A_k)$. Let $Dom(R)$ denote the set of all of the tuples of a type $dom(R)$. We may now describe a natural function $Rel : Dom(R) \to dom(R)$ defined as follows:

$$Rel(o = (i, R, o_1, ..., o_n)) = (v_1, ..., v_n), \text{ where } o_k = (i_k, A_k, v_k).$$

Definition. Tuples $o = (i, R, o_1, ..., o_n)$, $o' = (i', R, o'_1, ..., o'_n)$ shall be called *relationally equal* if for all $k = 1, ..., n$ holds $v_k = v'_k$.

2 Corollary. *The relation of relational tuple equivalency is an equivalence relation. Also, there exists a direct correspondence between the equivalence classes of the $Dom(R)$ set and the set $dom(R)$.*

Next we deal with relations, which are defined by Grefen as multisets of tuples. The state of a relation R is any object (i, R, T), where T is a sequence of tuples of a type R.

Definition. States $o = (i, R, T) \approx_{\mathcal{R}} o' = (i', R, T')$ are called relationally equal iff $|T| = |T'|$ and there is a permutation p of the sequence T, such that for each $t = 1, ..., |T|$ the states of the tuples $p(T)_t$ and T'_T are relationally equal.

3 Corollary. *The relation of relational state equivalency is an equivalence relation.*

Let $\mathcal{R} = (i, R, T)$ be a state of a relation R of a type $dom(R)$. For any tuple $v = (v_1, ..., v_n) \in dom(R)$ we define

$$\Psi_{\mathcal{R}}(v) = |\{o \in T; Rel(o) = v\}|,$$

as a number of tuple objects corresponding to the tuple v.

Such function establishes the multiset of tuples of a relation type $dom(R)$. Such multiset is called the instance of a relation of a type $dom(R)$ in [4]. By using this function we may characterize the relational states:

4 Lemma. *States $\mathcal{R} = (i, R, T), \mathcal{R}' = (i', R, T')$ of a relation R are relationally equal iff $\Psi_{\mathcal{R}}(v) = \Psi_{\mathcal{R}'}(v)$ for any $v \in dom(R)$*

Such characteristics leads us to the following conclusion:

5 Theorem. *There exists a mutual correspondence between instances of relations defined in [4], and the equivalence classes of relations' states relative to the relational equivalency of the states of relations.*

The above considerations show that the relational model in terms of the multiset theory is expressible through the relational equivalence classes in the USM. The corresponding states of the matching relation instances are included in the S_2, while the states corresponding to the tuples are included in the S_1.

5 The M0 / AS0 Model

In the 1990s K. Subieta initiated his work on an object-oriented query language integrated with a programming language. The basic assumptions of this concept have been presented in the works [9, 10]. One of the key assumptions of this approach is a data model called there the $M0$ model. Later this model has been renamed to $AS0$.

Subieta's assumptions in [9] about the AS0 objects are:

a. A triple $\langle i, n, v\rangle$ is an object that we call atomic object. The object has the internal identifier i, the external name n and the atomic value v.
b. A triple $\langle i_1, n, i_2\rangle$ is an object that we call pointer object. The object has the internal identifier i_1, the external name n and internal identifier i_2 which is understood as a pointer (or reference) to another object.
c. A triple $\langle i, n, T\rangle$, where T is a set of objects, is an object that we call complex object. Note this rule is recursive, thus allows one to create complex objects with an unlimited number of sub-objects and many hierarchy levels.

In addition, the correct set of objects it is a set satisfying the assumptions:

a. Different objects have different identifiers.
b. For any pointer object $\langle i_1, n, i_2\rangle$ in this set, its also contains an object with the identifier i_2.

AS0 objects are the most natural for modeling in the USM. For any set of objects satisfying the above conditions, the atomic objects and pointers are modeled by states from the set S_0. Objects containing them directly are modeled by the states from the set S_1, etc. Furthermore, the assumption that two different database objects always have different identifiers (this assumption may be added to the definition above) ensures that the states representing a correct set of AS0 objects form a proper state of a set of objects.

The state of a database in the AS0 model is state of a complex object (i_c, η, T) with the following properties:

a. T is a correct set of objects' states.
b. If $p = (i_p, n, i) \in \mathcal{S}(T)$, then there exists an object $o = (i, n_o, x) \in \mathcal{S}(T)$ pointed by p.
c. $L = \{(i_1, n_1, j_1), \ldots, (i_t, n_t, j_t)\}$ is a list of pointer objects such that there exist states $o = (j_k, n_o, x) \in \mathcal{S}(T)$ for each $k = 1, 2, \ldots, t$ pointed by those objects.

6 XML Data Model

The XML data model is best formalized using its Document Object Model (DOM), since we need a way to cater for the mixed content (i.e. text interleaved with tags). DOM maps each text fragment into a distinguished TXT node. Since XML has also attributes we need a way to differentiate them and XML elements. An XML element may consequently contain a nested element and an attribute with same name. We make this distinction assuming a new structure of the set of names N (see section 3). Let N be an union of three disjoint sets N_e, N_a, T, where N_e is the set of names of elements (tags), N_a is the set of names of attributes and T is the singleton of the special name of TXT nodes. In practice, the separation of N_e and N_a is done by prefixing names of attributes with @.

With this assumption on the set N we can use the Unified State Model from section 3 to represent the set of instances of XML documents. We will assume the universe S_∞ and define an equivalence relation in it. Observe that the Unified State Model assigns object identity to each object state, while constituents of XML documents have no identity.

Definition. Assume $f : I \to I$ being an injective function. The function rmp_f remaps the object identifiers according to the f function and is defined as follows:

$$\begin{aligned} rmp_f &: S_\infty \to S_\infty \\ rmp_f((i, n, x)) &= (f(i), n, x) \\ rmp_f((i, n, (o_1, \ldots, o_k))) &= (f(i), n, (rmp_f(o_1), \ldots, rmp_f(o_k))) \end{aligned}$$

Definition. Two object states o_1 and o_2 will be called *XML equivalent* if and only if there exists an injective function f such that $rmp_f(o_1) = rmp_f(o_2)$.

6 Corollary. *XML equivalence is an equivalence relation.*

7 Corollary. *We can define XML documents as equivalence classes of some relation on S_∞.*

7 C++ Object Model

The object model of C++ is similar to the Unified State Model presented in section 3. The difference lies in some restrictions on the fields nested in an object—their names must be pair-wise distinct. The ordering of particular fields in not important. However, among those fields there may be arrays. They can be modeled as a *sequence* of objects with repeating names.

Definition. A state of a C++ object is any member of S_∞.

Let us assume that a linear order $<_N$ on the set of names N is given. In case of strings it could be the natural lexicographic order.

Let $sort : S_\infty \to S_\infty$ denotes the function that given a complex object state stably sorts its subobjects by their names using the order $<_N$. The stability means that the ordering of objects with the same name is preserved. Thus, the semantics of C++ arrays is observed. Object states of S_0 are returned by the function *sort* intact.

Definition. The *C++ evaluation* denoted *cpp* is the function

$$\begin{aligned} cpp \ &: \ S_\infty \to S_\infty \\ cpp(o = (i, n, x)) &= o \\ cpp(o = (i, n, (o_1, o_2, \dots, o_k))) &= sort(i, n, (cpp(o_1), cpp(o_2), \dots, cpp(o_k))) \end{aligned}$$

The function *cpp* is thus the identity for object states of S_0. For all other object states, i.e. complex, the functions is first applied for all subobjects and then the subobjects are stably sorted by their names.

Definition. Two object states o_1 and o_2 will be called *C++ equivalent* if and only if $cpp(o_1) = cpp(o_2)$.

8 Corollary. *C++ equivalency is an equivalence relation.*

9 Theorem. *There is a direct correspondence between states of C++ objects and equivalence classes of the C++ equivalence relation.*

8 Data models of modern programming languages

Data modeling in modern programming language is achieved is through the classes and their objects. However, after looking closer at the object modeling, we see that the class diagram or the entity-relationships diagram are focused mostly on metadata description. In particular, after examining programming aspects such as inheritance we observe that they do not affect the state of the object, but serve only the purpose of simplification of the class modeling, defining and coding. From the objects' states' point of view each class is a separate collection of entities. Analysis of the state of an object of the given class `C` reveals that it is a collection of fields with names that are unique in the context of that object. It does not matter whether these fields are defined directly in the class `C`, or are they derived from the ancestor classes of `C`.

Let us have a look at a single object. As it has been noted, in terms of this object's state its membership in any class is not important, neither is the structure of classes. The only thing that matters are fields comprised within that object.

A field of an object can be of two kinds. It is either a field with an atomic value such as `int` or `char`, or it is a reference pointing to another object. Crucial for further considerations is the observation that in modern programming languages such as Java, Python, C# there are no subobjects which are complex objects.

In this context, the state of a single field can be represented as an object's state $(i, n, v) \in S_0$, where n is the field's name and v is its value. Note that if the field is a pointer to another object, this representation is valid as long as we manage to identify a representation for that object. Let u be an arbitrary object of a fixed programming language and let T be an arbitrary sequence containing objects' states representing the fields of the u. Then, any state of the form $(i, \eta, T) \in O_1$ is a good representative of the object u.

For any class `C` the set of all selected representatives of all its objects (existing in a given software state — not all of the possible objects) will be denoted by TC. Then, the object $(i, C, TC) \in O_2$ is a representative of the set of entities of the class `C`.

Suppose that the state $o_1 = (I_1, \eta, T_1)$ represents the object u, and is relationally equivalent to a $o_2 = (i_2, \eta, T_2)$ in the sense of relational equivalency of tuples. Then, o_2 also represents the object u. Of course, one can observe that two objects with exactly the same data, are still two different objects, and may be still be representatives of a completely different real objects. This is true, but on the other hand, if we cannot distinguish objects based on their characteristics, we are not able to distinguish which of them is the representative of the actual object. This leads to the following conclusion:

10 Corollary. *Objects of the modern programming languages are in the mutual correspondence with the equivalence classes of objects' states $(i, \eta, T) \in S_1$ due to the relational equivalency of tuples.*

Now let $o = (i, C, U = o_1, ..., o_t)$ be a representative of an entity of the class `C`. It is clear that the order of the elements of the set U does not matter. Thus, the following conclusion:

11 Corollary. *If the state of o' is in a relational equivalence with the state of $o = (i, C, U = \{o_1, ..., o_t\})$ representing the set of entities of the class `C`, is a state o' represents the same set of entities of class `C`.*

12 Corollary. *If the state o' is relationally equivalent to a state $o = (i, C, U = \{o_1, ..., o_t\})$ representing the set of entities of the class `C`, then the state o' represents the same set entities of the class `C`.*

The above corollaries clearly correspond with the conclusions of section 4.

9 Conclusion

The above considerations allow us to distinguish two classes of data models:

flat models This class includes the relational model and object models of most modern programming languages.

nested models This class contains the models of object-oriented databases, the XML model, the C++ language model and a model of nested relations. However, the nested relations model is quite specific in the terms of the USM and significantly differs from the other nested models.

This classification shows that at the time when the most popular and the key programming language was C++ and the key data model was the relational model, the problem of impedance mismatch between data models was a crucial obstacle. Currently, the leading programming languages for business applications belong to the same class of models as the relational model. This shows that, contrary to the common believes, object-relational mapping systems for languages such as Java, Python or C# have solid grounds based on the data models. On the other hand, this study explains why the mapping systems for C++ develop much more slowly and are more complicated.

References

1. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6) (1970) 377–387
2. Neward, T.: Avoiding the quagmire. Online (May 2007)
3. Lmmel, R., Meijer, E.: Revealing the x/o impedance mismatch. In Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J., eds.: Datatype-Generic Programming. Volume 4719 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2007) 285–367
4. Grefen, P.W.P.J., de By, R.A.: A multi-set extended relational algebra - a formal approach to a practical issue. In: ICDE, IEEE Computer Society (1994) 80–88
5. Cattell, R.G.G., Barry, D.K.: The Object Data Standard: ODMG 3.0. Morgan Kaufmann (2000)
6. Card, M.: OMG next-generation object database standardization white paper. Online (September 2007)
7. Subieta, K.: Theory and Construction of Object Query Languages [in Polish]. Publishers of the Polish-Japanese Institute of Information Technology (2004)
8. Smith, K.E., Zdonik, S.B.: Intermedia: A case study of the differences between relational and object-oriented database systems. SIGPLAN Not. **22** (December 1987) 452–465
9. Subieta, K., Kambayashi, Y., Leszczylowski, J.: Procedures in object-oriented query languages. In Dayal, U., Gray, P.M.D., Nishio, S., eds.: VLDB, Morgan Kaufmann (1995) 182–193
10. Subieta, K., Beeri, C., Matthes, F., Schmidt, J.W.: A stack-based approach to query languages. In: East/West Database Workshop. (1994) 159–180