University of Groningen

Declarative semantics of transactions in ORM

de Brock, E.O.

Published in:
Information Systems

DOI:
[10.1016/j.is.2016.03.005](https://doi.org/10.1016/j.is.2016.03.005)

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2016

[Link to publication in University of Groningen/UMCG research database](#)

Citation for published version (APA):

de Brock, E. O. (2016). Declarative semantics of transactions in ORM. *Information Systems*, 60, 85-94.
<https://doi.org/10.1016/j.is.2016.03.005>

Copyright

Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: <https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment>.

Take-down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): <http://www.rug.nl/research/portal>. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Declarative semantics of transactions in ORM

E.O. de Brock¹

University of Groningen, Faculty of Economics and Business, P.O. Box 800, 9700 AV Groningen, The Netherlands

ARTICLE INFO

Article history:

Received 6 March 2016

Accepted 10 March 2016

Available online 19 March 2016

Keywords:

Transaction modeling

Transaction language design

Semantics

Rollback

ORM-method

Transaction verbalization

ABSTRACT

In order to specify databases completely at the conceptual level, conceptual database specification languages should contain a data *definition* (sub)language (DDL), for specifying *data structures* (+constraints), a data *retrieval* (sub)language (DRL), for specifying *queries*, as well as a (declarative) data *manipulation* (sub)language (DML), for specifying *transactions*.

Object Role Modeling (ORM) is a powerful method for *designing* and *querying* database models at the conceptual level. By means of *verbalization* the application is also described in natural language as used by domain experts, for communication and validation purposes. ORM currently comprises a DDL and a DRL (ConQuer). However, the ORM-method does not yet contain an expressive DML for specifying transactions at the conceptual level.

In an earlier paper we designed a syntactic extension of the ORM-method with a DML for specifying transactions at the conceptual level in a purely declarative way. For all transactions we proposed syntaxes, verbalizations, and diagrams. However, we did not give a formal semantics then.

The purpose of this paper is to add a clear, formal and purely declarative semantics to the proposed ORM-transactions. The paper also formally defines rollbacks and illustrates everything with examples (including a solution to a well-known transaction specification problem). The extension of ORM with an expressive set of completely declaratively specified transactions makes ORM complete as a database specification method at the conceptual level.

© 2016 Elsevier Ltd. All rights reserved.

1. Introduction

In order to specify databases completely at the conceptual level, conceptual database specification languages should contain a data *definition* (sub)language (DDL), i.e., a part for specifying *data structures* (+constraints), a data *retrieval* (sub)language (DRL), i.e., a part for specifying *queries*, as well as a (declarative) data *manipulation* (sub)language (DML), i.e. a part for specifying *transactions*.

In the well-known database language SQL for example, the DDL typically contains CREATE-, DROP-, and ALTER-statements, the DRL typically contains SELECT-statements, and the DML typically contains INSERT-, DELETE-, and UPDATE-statements (see e.g. [1]).

Object Role Modeling (ORM) is a powerful method for *designing* and *querying* database models at the conceptual level. By means of *verbalization* the application can also be described in natural language which is easily understood by non-technical users (e.g., domain experts). Verbalization supports communication and validation with domain experts and future users of the system to be developed. Refs. [2] and [3] contain a lot of background information about ORM. ORM is also extensively described in [4]. ORM

E-mail address: E.O.deBrock@rug.nl

¹ Tel. +31 50 3637315.

started as a DDL. Later on the language was extended with a DRL, called ConQuer ([5,6]).

However, the ORM-method is incomplete in the sense that it does not yet contain a good and sufficiently expressive DML, for specifying and verbalizing transactions at the conceptual level, e.g., in order to discuss (standard, canned and complex) transactions with non-technical users. In this paper we will solve this incompleteness problem. One of our contributions is proposing the right set of primitive transactions, including a clear semantics.

When we look at the situation in comparable languages such as ER and UML, we first note that the Entity-Relationship (ER) conceptual modeling ([7]) does not have any notion of transaction at all. The Unified Modeling Language (UML, see [8]) does not have the notion of database nor does it have the notion of database transaction. The paper [18] tries to add transactions to UML-diagrams, but it leads to a restricted and cumbersome theory. UML is not very well suited for database modeling anyway. ORM provides a simpler, more accurate and more powerful approach to information modeling at the conceptual level than UML ([3,9,10]). Ref. [2] contains various articles comparing UML and ORM.

The basic notion of a *fact type* in ORM is more or less 'comparable' to the notion of a class in UML, an entity type (or relationship) in ER, and a table type in relational database theory.

The operations **add** and **del** in [4] only apply to one fact (instance) at a time. The operation **add** in [4] corresponds to our addition of an instance, treated in Section 3. The operation **del** in [4], removal of an instance, is a very special case of the removal of a subset treated in Section 5. However, only adding or deleting one particular fact at a time is not enough (e.g., deleting all order lines belonging to a given order is a natural counterexample). Moreover, the operation **del** in [4] requires the complete fact to be mentioned; e.g., with a fact type such as *Employee earns Salary* (see the example in Section 6) a removal would look like "**del**: Employee 123 earns Salary 4857". But maybe you only want to say that the salary fact on Employee 123 has to be removed; maybe you don't know the exact salary (or maybe are not even allowed to know the salary).

Balsters et al. mention transactions in [11] and [12], but actually they concentrate on dynamic rules, and do not give a syntax for data manipulation operations as such. In [11], adding actual operations to the ORM-language that explicitly model transactions is mentioned as future work, which it still is up to now. Our paper addresses this open problem.

The ORM-situation sketched above is schematically summarized below:

	SQL	ORM-approach
DDL:	CREATE, DROP, ALTER	+(ORM's origin)
DRL:	SELECT	+(ConQuer)
DML:	INSERT, DELETE, UPDATE	–

In [13] we proposed a syntactic extension of ORM with a DML for specifying transactions at the conceptual level in a purely declarative way, to be easily validated by domain experts. By a transaction we informally mean an *attempt* to update the contents of the database; the attempt fails when any specified constraint will be violated (also known as a *rollback*). We will call our DML *ConTrans* (for Conceptual Transaction), analogous to *ConQuer* (which stands for Conceptual Query).

In [13] we introduced a collection of four basic classes of specifiable transactions: add a fact, add a query result, remove a subset, and change a subset. We also introduced compound transactions in ORM. Together they constitute an expressive collection of transactions. Other conceptual constructs are not needed.

In principle, a database treats only one transaction at a time (except for compound transactions). We note however that the order of application of two transactions can be semantically relevant: the transaction sequence T1; T2 (where the semicolon means 'followed by') can have another end result than the transaction sequence T2; T1. For instance, let T1 be the transaction to increase the salary of all employees living in London with 10% and T2 be the transaction to increase the salary of all employees with a salary less than 5000 with 100. Then the transaction sequence T1; T2 implies that all Londoners with a salary originally less than 4545 will earn 10% more plus an additional 100 (i.e., $1.10 \cdot \text{Old_Salary} + 100$), while the transaction sequence T2; T1 implies that all Londoners with a salary originally less than 5000 will earn 10% more plus an additional 110, namely $1.10 \cdot (\text{Old_Salary} + 100)$.

Although one table (resp. one tuple) in a relational database is directly associated to a *set* of fact types (resp. facts) in ORM, it is a crucial decision in our theory that the smallest *unit* of transaction is a fact, whereas in SQL it is a tuple. Nevertheless a nice feature of the syntax (and semantics) of transactions we propose for ORM at the conceptual level is that it is similar to that of SQL, and that other conceptual constructs were not necessary. The basic transactions attempt to populate (add), de-populate (remove) or re-populate (change) a fact type (or an independent object type). An independent object type can be considered as a fact type having only one role associated to it. Recall that the attempt fails when any specified constraint will be violated. Each basic transaction will apply to only one fact type (or one independent object type) at a time. The proposal is inspired by (the expressiveness of) the DML of SQL.

Because the ORM-tradition distinguishes several kinds of specifications, in [13] we proposed for each transaction a syntax (in a formal language), a verbalization (in natural language, "fully communication oriented" [14]), and a diagram (in a graphical language). A verbalization of a transaction (i.e., in natural language) is intended for communication and validation with domain experts and future users.

This paper extends [13] by adding a clear formal semantics to the proposed ORM-transactions. This is a set-theoretic semantics along the lines of [15,16] and [17], formally validating the proposed DML. It also allows to study the expressiveness of the DML. And although [17] formally treats declarative specifications of (complex) transactions, it does not provide an accompanying specification language. And instead of ORM (as in this paper), Chapter 9 of [16] contains a similar approach towards SQL.

The rest of the paper is organized as follows. Section 2 introduces our semantics for ORM. Sections 3–6 subsequently treat each of the four aforementioned classes of basic transactions, i.e. add an instance, add a query result, remove a subset, and change a subset. Section 7 treats compound transactions. For the transaction classes in Sections 3–7 we introduce (1) its syntax, (2) its verbalization, and (3) its semantics, and (4) one or more illustrative examples. For each class of transactions we define its syntax, verbalization, and semantics in terms of the syntax, verbalizations, and semantics of its constituents respectively (*compositionality principle*). The general structure we give is more important than the actual syntax and verbalizations we chose.

In summary, Sections 3–7 discuss and fill the table below (row by row):

Kind of specification →	(1) Syntax (formal language)	(2) Verbalization (natural language)	(3) Semantics
↓ Transaction class			
§3: add a fact			
§4: add a query result			
§5: remove a subset			
§6: change a subset			
§7: compound transaction			

Section 8 shows the applicability of our proposal by solving a well-known transaction specification problem. We end the paper with some concluding remarks and possible future research.

2. A population semantics for ORM

Since we want to model transactions, which are related to *state changes*, we will be interested in the possible states for a given ORM-schema. In order to come up with a semantics suitable for modeling conceptual transactions, we first have to explain the structure of the populations (or states) that are possible for a given ORM-schema.

The population of a given fact type can be represented by a fact table with one column for each role in the fact type and one row for each fact. A fact table can be modeled as a *set* of facts. And a fact can be modeled as a *function* that assigns to a role the corresponding value. So if R is any set of roles then we can define a *fact table under R* as a set of functions over R , i.e., functions with R as their (common) domain.

10000	J. Jones
10010	P. Peters
10020	K. Brown
10030	B. Black
10040	W. White

Fig. 1. A fact type with a corresponding fact table (containing 5 facts).

As an example we introduce a fact type for a fact type called *Employee has Name* with two roles, *Employee* and *Name*. Our sample fact table under $\{Employee, Name\}$ in Fig. 1 represents 5 facts. The diagram expresses that each *Employee* (an entity with reference mode *nr*) has exactly one *Name* (which is a value).

A population of a given ORM-schema is known as a *conceptual database* and can be represented by associating a fact table to each of its fact types and independent object types. (Not only fact types but also independent object types can have populations; an independent object type typically has only one role associated to it.) We will call the types of an ORM-schema that can be populated (i.e., the fact types and independent object types) the *inhabitable types* of the schema.

By the *conceptual skeleton* of an ORM-schema we mean the function that assigns to each inhabitable type of the schema the set of roles in that type. Note that an ORM-schema uniquely determines its conceptual skeleton.

For example, the conceptual skeleton G_0 we will use in this paper consists of 6 fact types, each with 2 roles, see Fig. 2. The first fact type was already introduced above, see Fig. 1. Our conceptual skeleton G_0 is determined by the following 6 equations, which assign to each of the 6 fact types the corresponding set of roles:

$G_0(\text{Employee has Name})$	$= \{\text{Employee, Name}\}$
$G_0(\text{Employee lives in City})$	$= \{\text{Employee, City}\}$
$G_0(\text{Employee earns Salary})$	$= \{\text{Employee, Salary}\}$
$G_0(\text{Applicant has Name})$	$= \{\text{Applicant, Name}\}$
$G_0(\text{Applicant obtains Hire Advice})$	$= \{\text{Applicant, Hire Advice}\}$
$G_0(\text{Applicant receives Employee Number})$	$= \{\text{Applicant, Employee Number}\}$

Fig. 2. A conceptual skeleton with 6 fact types (each with 2 roles).

A *conceptual database* assigns to each inhabitable type a fact table under the corresponding set of roles. Formally: if G is a conceptual skeleton (hence a set-valued function) and I its domain (representing the set of inhabitable types) then we say that d is a *conceptual database over G* iff d is a function over I and for each $T \in I$: $d(T)$ is a fact table under $G(T)$.

Finally, we say that U is a *conceptual universe over G* iff U is a set of conceptual databases over G .

An ORM-schema uniquely determines a corresponding conceptual universe as we will now explain. First, as we already noted, an ORM-schema uniquely determines its conceptual skeleton. In an ORM-schema each role is

connected to an object type. Each object type determines a set of permitted instances. The conceptual skeleton combined with a set of permitted instances for each role in the skeleton gives an upper bound for the set of conceptual databases that are possible for that ORM-schema. This set is further restricted by the constraints that are specified in the ORM-schema. These ORM-constraints can be translated to constraints on conceptual databases. They determine a subset of the just mentioned upper bound for the set of possible conceptual databases. Now this subset constitutes the conceptual universe corresponding to the ORM-schema. Ref. [4] shows various examples of specifying such constraints.

With this population semantics we are able to give a clear semantics for our conceptual transactions. Specifying conceptual transactions in the context of a given conceptual universe can be considered as specifying functions that assign to each 'old' conceptual database the 'new' conceptual database (a state change).

More formally: if U is a conceptual universe then we say that t is a conceptual transaction on U iff t is a function from U into U . So in other words, t assigns to each element of U an (other) element of U .

Often a transaction t on a universe U is specified in the form of an unconditional update (i.e., a function f over U but not necessarily into U), but modified on the basis of U , in the sense that for any $d \in U$:

$$\begin{aligned} t(d) &= f(d) \text{ if } f(d) \in U \text{ and} \\ t(d) &= d \text{ if } f(d) \notin U \end{aligned}$$

We call the function t the adaption of f based on U and denote it by $\text{Adap}_U(f)$: So, $t = \text{Adap}_U(f)$. Note that it is guaranteed that $t(d) \in U$ for each $d \in U$, i.e., that t is a function from U into U , in other words, that t is a conceptual transaction on U . It expresses that the transaction t is indeed an *attempt* to update the contents of the information system and that the attempt fails when any constraint specified in the definition of U will be violated, in which case the database state remains the same (thus formalizing a *rollback*).

We note that this simple adaption is generally applicable, i.e., for any function f over any database universe U in any database state d . Other general strategies to evade violations are often problematic, see for instance under (3) in Sections 4–6. All transactions we define in Sections 3–7 are actually of this general form of adaption.

3. Addition of an instance

We start with a simple class of transactions, the addition of a given instance to an inhabitable type F (i.e., a fact type or an independent object type). Suppose F is n -ary and $v_1, \dots, v_n$ are the respective values for the instance to be added.

We will introduce the syntax under (1), the verbalization under (2), the semantics under (3), and an example under (4).

(1) We propose the following ORM-syntax: **ADD TO F VALUES ($v_1; \dots; v_n$)**

(2) We define the verbalization of the transaction in terms of the verbalization of its constituents:

$$\text{Verbalization}(\text{ADD TO } F \text{ VALUES } (v_1; \dots; v_n)) = \text{add to Verbalization}(F) \text{ Verbalization}(v_1; \dots; v_n)$$

where $\text{Verbalization}(F) = \text{fact type } F$ and

$\text{Verbalization}(v_1) = \text{the instance with value } v_1$ (for $n=1$) and, for $n > 1$

$\text{Verbalization}(v_1; \dots; v_n) = \text{the instance with values } v_1, \dots \text{ and } v_n \text{ respectively}$

For example, in the ternary case (i.e., when $n=3$) we get:

$$\text{Verbalization}(\text{ADD TO } F \text{ VALUES } (v_1; v_2; v_3)) =$$

add to fact type F the instance with values v_1, v_2 and v_3 respectively

(3) The row to be inserted in the fact table can formally be described by the function $f = \{(r_1; v_1), \dots, (r_n; v_n)\}$, assigning to each role r_i the value v_i .

If d represents the 'old' conceptual database then the 'new' conceptual database d' is defined as follows:

$$d'(F) = d(F) \cup \{f\} \text{ and } d'(T) = d(T) \text{ for all other types } T,$$

provided that d' satisfies all specified constraints; otherwise $d' = d$.

In other words, the 'new' fact table for F consists of the 'old' fact table for F plus the row f , provided that that situation still satisfies all specified constraints; otherwise the database state remains the same.

Here we clearly see that the transaction is an *attempt* to add an instance, that the attempt fails when any specified constraint will be violated (i.e., a *rollback*), that the transaction always results in a *correct* conceptual database (i.e., satisfying all specified constraints), and that the transaction applies to only one type at a time.

(4) **Example.** We give here as an example the addition of employee 123 called 'J. Smith' to the (binary) fact type *Employee has Name* introduced in Section 2. We recall that this fact type has the following ORM-diagram:

The ORM-syntax given under (1) for this addition will then result in:

**ADD TO Employee has Name
VALUES (123; 'J. Smith')**

Applying the verbalization rules given under (2) we get:

$\text{Verbalization}(\text{ADD TO Employee has Name VALUES (123; 'J. Smith')}) =$

**add to fact type Employee has Name
the instance with values 123 and 'J. Smith' respectively**

4. Addition of a query result

We continue with the addition of the result of a query q (multiple instances) to an inhabitable type F (i.e., a fact type or an independent object type). For the query-part we can make use of the retrieval language ConQuer [5].

Again we introduce the syntax under (1), the verbalization under (2), the semantics under (3), and an example under (4).

(1) We propose the following ORM-syntax: **ADD TO F RESULT** q

The query q should produce the same number of roles as F has. If the roles order in q , say $(r_1; \dots; r_n)$, is (perhaps) not the same as the roles order in F then the expression $(r_1; \dots; r_n)$ has to be added:

ADD TO $F(r_1; \dots; r_n)$ **RESULT** q instead of **ADD TO F RESULT** q

(2) The verbalization of the transaction is expressed in terms of the verbalization of its constituents:

$\text{Verbalization}(\text{ADD TO F RESULT } q) =$

add to $\text{Verbalization}(F)$ **the** $\text{Verbalization}(q)$

where $\text{Verbalization}(F) = \text{fact type } F$ (see Section 3)

and $\text{Verbalization}(q)$ heavily depends on q itself. (For examples of queries see [5] and the example under (4) in this section.)

(3) Let row set S be the result of the query q in the 'old' conceptual database d . (So, S depends on d .) Then the 'new' conceptual database d' is defined as:

$d'(F) = d(F) \cup S$ and $d'(T) = d(T)$ for all other types T ,

provided that d' satisfies all specified constraints; otherwise $d' = d$.

In other words, the 'new' fact table for F consists of the 'old' fact table for F plus the row set S , provided that that situation still satisfies all specified constraints; otherwise the database state remains the same.

Again we clearly see that the transaction is an *attempt* to add a row set, that the attempt fails when any specified constraint will be violated (i.e., a *rollback*), that the transaction always results in a *correct* conceptual database (i.e., satisfying all specified constraints), and that the transaction applies to only one type at a time.

Another strategy you might think of is to change $d'(F) = d(F) \cup S$ into $d'(F) = d(F) \cup S'$ where S' is the largest subset of S such that the result does not violate the specified constraints. However, such a largest subset is not always uniquely determined. Also, such a possibly non-deterministic (and anyway unasked) addition might not always be desirable.

(4) **Example.** We introduce three fact types, already announced in Section 2, called *Applicant has Name*, *Applicant obtains Hire Advice*, and *Applicant receives Employee Number*, with the following ORM-diagrams:

All 3 diagrams express that Applicant is an entity with reference mode *nr*. Also, the successive diagrams express that each Applicant has exactly one Name (a value), an

Applicant obtains one Hire Advice (a value), and an Applicant receives one Employee Number (a value).

Now we want to add all applicants with the hire advice 'Yes' to the fact type *Employee has Name* introduced earlier. Then the underlying query q_0 expressed in ConQuer is

$q_0 =$ Applicant
 └─ receives $\sqrt{\text{Employee Number}}$
 └─ has $\sqrt{\text{Name}}$
 └─ obtains Hire Advice = 'Yes'

Substituting this in the ORM-syntax given under (1) will result in:

ADD TO Employee has Name

RESULT Applicant

└─ receives $\sqrt{\text{Employee Number}}$
 └─ has $\sqrt{\text{Name}}$
 └─ obtains Hire Advice = 'Yes'

The verbalization of the query q_0 will be: Employee Number **and** Name **of each** Applicant **who** obtains Hire Advice **equals** 'Yes'.

Applying the verbalization rules given under (2) we then get:

$\text{Verbalization}(\text{ADD TO Employee has Name RESULT } q_0) =$

add to fact type Employee has Name **the** Employee Number **and** Name **of each** Applicant **who** obtains Hire Advice **equals** 'Yes'.

5. Removal of a subset

We now treat the removal of a subset (multiple instances) from an inhabitable type F . The set to be removed has the form $\{t \mid t \in d(F) \text{ [and } t \text{ satisfies } C]\}$ for some optional condition C on the instances in the fact table of F ; here the brackets '[' and ']' denote optionality.

Again we introduce the syntax under (1), the verbalization under (2), the semantics under (3), and some examples under (4).

(1) We propose the following ORM-syntax: **REMOVE FROM F [WHEN C]**

For the condition-part we can again make use of the language ConQuer [5].

(2) The verbalization of the transaction is again expressed in terms of the verbalization of its constituents:

$\text{Verbalization}(\text{REMOVE FROM F [WHEN C]}) =$

remove from $\text{Verbalization}(F)$ **all instances** [**for which** $\text{Verbalization}(C)$]

where $\text{Verbalization}(C)$ heavily depends on C itself. (See under (4) in this section and in Section 6 for examples.)

(3) Let d be the 'old' conceptual database and row set S the set to be removed. (S is dependent on d .) Then the 'new' conceptual database d' is defined as:

$d'(F) = d(F) - S$ and $d'(T) = d(T)$ for all other types T ,

provided that d' satisfies all specified constraints; otherwise $d' = d$.

In other words, the 'new' fact table for F consists of the 'old' fact table for F minus the row set S , provided that that

situation still satisfies all specified constraints; otherwise the database state remains the same.

Again we clearly see that the transaction is an *attempt* to remove a row set, that the attempt fails when any specified constraint will be violated (i.e., a *rollback*), that the transaction always results in a *correct* conceptual database (i.e., satisfying all specified constraints), and that the transaction applies to only one type at a time.

Another strategy you might think of is to change $d'(F) = d(F) - S$ into $d'(F) = d(F) - S'$ where S' is the largest subset of S such that the result does not violate the specified constraints. However, such a largest subset is not always uniquely determined. Moreover, such a possibly non-deterministic (and anyway unasked) removal might not always be desirable.

(4) **Example (a).** We continue our running example with the removal of all applicants with the hire advice 'Yes' from the fact type *Applicant obtains Hire Advice*. Hence, the condition C_a is: Hire Advice='Yes'. The ORM-syntax will then be:

REMOVE FROM Applicant obtains Hire Advice

WHEN Hire Advice='Yes'

Applying the verbalization rules given under (2) we get:

Verbalization(**REMOVE FROM** Applicant obtains Hire Advice **WHEN** Hire Advice='Yes')

= **remove from** Verbalization(Applicant obtains Hire Advice)

all instances for which Verbalization(C_a)

= **remove from fact type** Applicant obtains Hire Advice

all instances for which Hire Advice **equals** 'Yes'

Example (b). Also in the fact type *Applicant has Name* we want to remove all applicants having the hire advice 'Yes'. Therefore we have to look into another fact type, *Applicant obtains Hire Advice*, for those applicants for which Hire Advice='Yes'. This leads to the following ORM-syntax:

REMOVE FROM Applicant has Name

WHEN Applicant

└ obtains Hire Advice = 'Yes'

Applying the verbalization rules given under (2) results in the verbalization

remove from fact type Applicant has Name

all instances for which Applicant obtains Hire Advice **equals** 'Yes'

Example (c). To finish our running example, we want to remove from the fact type *Applicant receives Employee Number* all applicants having the hire advice 'Yes'. Again we will look in the fact type *Applicant obtains Hire Advice* for those applicants for which Hire Advice='Yes'. This

leads to the following ORM-syntax:

Similarly, the verbalization rules given under (2) will result in the verbalization

remove from fact type Applicant receives Employee Number

all instances for which Applicant obtains Hire Advice **equals** 'Yes'

6. Change of a part of a fact type

A change simply seems to be a compound transaction (Section 7), consisting of a removal followed by an addition. However, after the removal the data to be added (in a changed form) do not exist anymore... Therefore we will introduce and treat changes separately, as usual in database language design. (Otherwise, we would need to introduce and use something like a "snapshot" mechanism.)

We will now introduce the change of a part of a fact type of a fact type F , i.e., replacing the values of some roles $r_1, \dots, r_k$ in F by the (old) values of expressions $z_1, \dots, z_k$ *simultaneously* for *all* instances satisfying a certain (optional) condition C . We depict the basic idea of the simultaneity as follows (where the gray part indicates the part of the table that might change):

		r_1	...	r_k		
		ch	ch	ch		C
		ch	ch	ch		C
		ch	ch	ch		C
		ch	ch	ch		C
		ch	ch	ch		C

If the fact type F is elementary then k will be 1, i.e. the values of only one role – the non-key one – will be changed.

Again we introduce the syntax under (1), the verbalization under (2), the semantics under (3), and an example under (4).

(1) We propose the following ORM-syntax

CHANGE IN F SET $r_1 := z_1, \dots, r_k := z_k$ [**WHEN** C]

(2) The verbalization of the transaction is expressed in terms of the verbalization of its constituents:

Verbalization(**CHANGE IN F SET** $r_1 := z_1, \dots, r_k := z_k$ [**WHEN** C]) =

change in Verbalization(F)

all instances [for which Verbalization(C)

simultaneously such that

REMOVE FROM Applicant receives Employee Number

WHEN Applicant

└ obtains Hire Advice = 'Yes'

Verbalization(r_i) **becomes the old value of** Verbalization(z_i),

..... **and**

Verbalization(r_k) **becomes the old value of** Verbalization(z_k)

Sometimes it is clear from condition C that at most one instance can satisfy it (typically when C is a condition on a key value, e.g., Employee Number=123), in which case the expression **all instances** can be replaced by the expression **the instance** and the word **simultaneously** can be omitted.

(3) Let d be the ‘old’ conceptual database, h the function over $R=\{r_1, \dots, r_k\}$ such that $h(r_i)$ is the value of the expression z_i in d , and S be the set of rows in d to be changed, i.e., $S=\{t \mid t \in d(F) \text{ [and } t \text{ satisfies } C]\}$. And let g be the function over S that assigns to each $t \in S$ the new fact row. To be precise: $g(t)(r)=h(r)$ for each $r \in R$ and $g(t)(r)=t(r)$ for each F -role $r \notin R$. (Hence h , S , and g depend on d .) Now the ‘new’ conceptual database d' is defined as follows:

$$d'(F) = (d(F) - S)$$

$$\cup \{g(t) \mid t \in S\} \text{ and } d'(T) = d(T) \text{ for all other types } T,$$

provided that d' satisfies all specified constraints; otherwise $d'=d$.

In other words, the ‘new’ fact table for F consists of the ‘old’ fact table for F minus the set of rows to be changed, augmented with the set of new fact rows, provided that that situation still satisfies all specified constraints; otherwise the database state remains the same.

Again we see that the transaction is an *attempt* to change, that the attempt fails when any specified constraint will be violated (i.e., a *rollback*), that the transaction always results in a *correct* conceptual database (i.e., satisfying all specified constraints), and that the transaction applies to only one type at a time.

Another strategy you might think of is to change $d'(F) = (d(F) - S) \cup \{g(t) \mid t \in S\}$ into $d'(F) = (d(F) - S') \cup \{g(t) \mid t \in S'\}$ where S' is the largest subset of S such that the result does not violate the specified constraints. However, such a largest subset is not always uniquely determined. Moreover, such a possibly nondeterministic (and anyway unasked) change might not always be desirable.

(4) **Example.** We introduce two fact types, already announced in Section 2, called *Employee lives in City* and *Employee earns Salary*, with the following ORM-diagrams:

The diagrams express that Employee is an entity with reference mode *nr*. Also, the successive diagrams express that each Employee lives in exactly one City (a value) and an Employee earns exactly one Salary (a value).

We choose as our change-example an increase of 10% of the salary of all employees living in London with a salary less than 5000 (our new so-called ‘London allowance’). So

the content of the fact type *Employee earns Salary* has to be changed. The condition C is:

Employee
 └─ lives in City = ‘London’
 └─ earns Salary < 5000

The ORM-syntax will then be:

CHANGE IN Employee earns Salary
SET Salary := Salary * 1.10
WHEN Employee
 └─ lives in City = ‘London’
 └─ earns Salary < 5000

Applying the verbalization rules given under (2) we get:

change in fact type Employee earns Salary
all instances for which Employee lives in City **equals** ‘London’ **and**
 Employee earns Salary **less than** 5000
simultaneously such that
 Salary **becomes the old value of** Salary **times** 1.10

7. Compound transactions

Sometimes we want several elementary transactions to be considered as one composite transaction that will be accepted or rejected as a whole. For instance, if the three Applicant roles introduced under (4) in Section 4 would all be mandatory then the three removal transactions under (4) in Section 5 should be treated as one composite update. (Under (4) in this section this case will be used as the example.)

Therefore we also need the notion of compound transaction in ORM, in order to obtain an expressive collection of transactions. This is not only because one needs compound transactions anyway (since the basic transactions are only one-table-at-a-time or one-facttype-at-a-time), but also because ORM-transactions usually have a finer granularity than, say, SQL-transactions: One *basic* SQL-transaction can typically correspond to a *set of* ORM-transactions. This set of ORM-transactions should then be accepted or rejected as a whole. So compound transactions will probably occur more frequently in ORM than for instance in SQL.

We introduce the syntax for compound transactions under (1), the verbalization under (2), the semantics under (3), and an example under (4).

- (1) In line with [4] we indicate the beginning and end of a compound transaction by means of the keywords **BEGIN** and **END** as follows:

BEGIN
 Transaction 1;
 :
 :
 Transaction $n-1$;
 Transaction n
END

- (2) The verbalization of the compound transaction is expressed in terms of the verbalization of its constituents:

```

Verbalization( BEGIN T1; ...; Tn-1; Tn END )=
try to Verbalization(T1), then
:
:
Verbalization(Tn-1), and then
Verbalization(Tn)
as a whole

```

- (3) All transaction descriptions in Sections 3–6 ended with the clause “provided that d' satisfies all specified constraints; otherwise $d' = d$ ”. Therefore all transactions can be described in the form of $\text{Adap}_U(f)$, the adaption of an unconditional update f . Now let T_1 be described by $\text{Adap}_U(f_1)$, ..., T_{n-1} by $\text{Adap}_U(f_{n-1})$, and T_n by $\text{Adap}_U(f_n)$, then the semantics of the non-compound sequence of transactions $T_1; \dots; T_{n-1}; T_n$ is described by the function composition $\text{Adap}_U(f_n) \circ \text{Adap}_U(f_{n-1}) \circ \dots \circ \text{Adap}_U(f_1)$. In other words, the end result of every individual unconditional update could be adapted separately.

On the other hand, the semantics of the compound transaction

BEGIN $T_1; \dots; T_{n-1}; T_n$ **END** is described by $\text{Adap}_U(f_n \circ f_{n-1} \circ \dots \circ f_1)$

In other words, only the end result after evaluation of all unconditional updates will be adapted (if necessary).

We see that the compound transaction is an *attempt* to update, that the attempt fails – and the compound transaction is rejected as a whole – when at the end any specified constraint will be violated (i.e., a *rollback* of the complete compound transaction), that the transaction always results in a *correct* conceptual database (i.e., satisfying all specified constraints), but that the compound transaction can apply to several types at a time.

- (4) **Example.** As mentioned in the first paragraph of this section, if the three Applicant roles introduced under (4) in Section 4 are all mandatory then we want the three removal transactions given under (4) in Section 5 to be treated as one composite update, to be accepted or rejected as a whole. Substituting them in the ORM-syntax given under (1) will result in:

BEGIN

REMOVE FROM Applicant obtains Hire Advice

WHEN Hire Advice = 'Yes';

REMOVE FROM Applicant has Name

WHEN Applicant

└ obtains Hire Advice = 'Yes';

REMOVE FROM Applicant receives Employee Number

WHEN Applicant

└ obtains Hire Advice = 'Yes'

END

Applying the rule given under (2) to the verbalizations under (4) in Section 5 will result in the verbalization

```

try to
remove from fact type Applicant obtains Hire Advice
all instances for which Hire Advice equals 'Yes',
then remove from fact type Applicant has Name
all instances for which Applicant obtains Hire Advice equals
'Yes',
and then remove from fact type Applicant receives Employee
Number
all instances for which Applicant obtains Hire Advice equals
'Yes'
as a whole

```

8. A composite example

As an illustration of the applicability of our proposal we will treat and solve the (in)famous transaction specification problem of the transfer of an amount from one account to another one.

Suppose we have a fact type called *Account has Balance* with two roles, *Account* and *Balance*. The diagram expresses that each Account (an entity with reference mode *nr*) has exactly one Balance (a value).

We will consider the transaction (that applies to only one fact type)

Transfer an amount B from account A1 to account A2 (where A1 ≠ A2), which can be illustrated as follows:

A1	S1
A2	S2
A3	S3
A4	S4
...	...

State A

A1	S1 – B
A2	S2 + B
A3	S3
A4	S4
...	...

State B

The intention is that when we have state A before the transaction, we will have state B after the transaction, provided that state B satisfies all specified constraints; otherwise the state remains the same (i.e., state A).

We will abbreviate *Account has Balance* to AhB, *Account* to Acc and *Balance* to Bal. The intended transaction can now be expressed as follows:

```

BEGIN
CHANGE IN AhB SET Bal:= Bal-B WHEN AhB=A1;
CHANGE IN AhB SET Bal:= Bal+B WHEN AhB=A2
END

```

For easy reference we will abbreviate the transaction as: **BEGIN** T₁; T₂ **END**

When we apply the definitions in this paper we get the following:

According to Section 7, the compound transaction **BEGIN** T₁; T₂ **END** is described by $\text{Adap}_U(f_2 \circ f_1)$, when T₁ is described by $\text{Adap}_U(f_1)$ and T₂ by $\text{Adap}_U(f_2)$. We are now able to compute $\text{Adap}_U(f_2 \circ f_1)$, i.e., the adaptation of the composition of the two transaction effects:

According to (3) of Section 6, after application of transaction T₁ in state d we have for the new conceptual database d':

$d'(\text{AhB}) = (d(\text{AhB}) - \{X_1\}) \cup \{Y_1\}$ where X₁ is the fact in state d for which AhB=A1 holds and Y₁ is the new fact (derived from X₁) where Balance has been lowered with B; and d'(T)=d(T) for all other types T.

Now we have to apply transaction T₂ starting from state d'. According to (3) of Section 6, after application of transaction T₂ in state d' we have for the new conceptual database d'' (using substitution and reordering):

$$\begin{aligned}
 d''(\text{AhB}) &= (d'(\text{AhB}) - \{X_2\}) \cup \{Y_2\} \\
 &= (((d(\text{AhB}) - \{X_1\}) \cup \{Y_1\}) - \{X_2\}) \cup \{Y_2\} \\
 &= (d(\text{AhB}) - \{X_1, X_2\}) \cup \{Y_1, Y_2\}
 \end{aligned}$$

where X₂ is the fact in state d' for which AhB=A2 holds and Y₂ is the new fact (derived from X₂) where Balance has been increased with B; furthermore d''(T)=d'(T) for all other types T.

Finally, when applying Adap_U to this composition, this all holds provided that d'' satisfies all specified constraints; otherwise d''=d.

Note that this is just what we wanted: $(d(\text{AhB}) - \{X_1, X_2\}) \cup \{Y_1, Y_2\}$, i.e., the old fact table d(AhB) without the old facts about A1 and A2 but with the new facts about A1 and A2 added, provided that all specified constraints are satisfied; otherwise the state remains the same.

State B clearly shows that the intended transaction applies to one fact type only, where only changes take place. When account numbers happen to be numeric, the intended transaction can in principle (but somewhat improperly) also be expressed by one basic transaction as follows:

```

CHANGE IN AhB
SET Bal:= Bal+B*(2*Acc - A2-A1)/(A2-A1)
WHEN AhB=A1 OR AhB=A2

```

To prove this we apply the definitions in this paper to this transaction:

According to (3) of Section 6, application of this transaction in state d will lead to the new conceptual database d' where $d'(\text{AhB}) = (d(\text{AhB}) - S) \cup \{g(t) | t \in S\}$ and $d'(T) = d(T)$ for all other types T, provided that d' satisfies all specified constraints; otherwise d'=d. Here S and g(t) are defined as:

$S = \{t | t \in d(\text{AhB}) \text{ and } (t(\text{Acc}) = A1 \text{ or } t(\text{Acc}) = A2)\}$, i.e., the set of rows to be changed, and $g(t)(\text{Bal}) = t(\text{Bal}) + B * (2 * t(\text{Acc}) - A2 - A1) / (A2 - A1)$ while $g(t)(r) = t(r)$ for all other roles r.

For $t(\text{Acc}) = A1$ the expression reduces as follows:

$$\begin{aligned}
 g(t)(\text{Bal}) &= t(\text{Bal}) + B * (2 * A1 - A2 - A1) / (A2 - A1) \\
 &= t(\text{Bal}) + B * (A1 - A2) / (A2 - A1) \\
 &= t(\text{Bal}) - B
 \end{aligned}$$

For $t(\text{Acc}) = A2$ the expression reduces as follows:

$$\begin{aligned}
 g(t)(\text{Bal}) &= t(\text{Bal}) + B * (2 * A2 - A2 - A1) / (A2 - A1) \\
 &= t(\text{Bal}) + B * (A2 - A1) / (A2 - A1) \\
 &= t(\text{Bal}) + B.
 \end{aligned}$$

Note that this is indeed the same result as before, i.e. the old fact table d(AhB) without the old facts about A1 and A2 but with the new facts about A1 and A2 added, provided that all specified constraints are satisfied; otherwise the state remains the same.

9. Summary and conclusions

In [13] we proposed an extension of the ORM-method with a DML for specifying transactions at the conceptual level and in a purely declarative way: We introduced a collection of four basic classes of specifiable transactions, as well as compound transactions. For all transactions we proposed syntaxes and verbalizations. However, we did not give a formal semantics then.

The current paper adds a clear population semantics to the proposed ORM-transactions and illustrates the applicability of our proposal with various examples. The paper also formally defines rollbacks. The paper ends with an illustration of the applicability of our proposal with the solution of a well-known transaction specification problem.

One of our contributions is the proposal of an expressive set of primitive (purely declarative) transactions for ORM, including syntax, verbalizations, and a clear semantics. In retrospect, we also used or introduced the syntax and verbalizations for a fact type, a condition, a query, a role, a value, and an instance.

The extension with completely declaratively specified transactions makes ORM complete as a specification method at the conceptual level. It also makes the ORM language to a certain extent comparable to and even competitive with SQL (because it is at a higher level): after this extension ORM provides a complete database specification language (with a DDL, a DRL, and a DML as well), but now completely at the conceptual level. Verbalization (i.e., in natural language) as used in ORM supports

validation with domain experts as well as communication with the future users of the system to be developed, which we now extended to transactions as well.

10. Possible future research

Along the lines of the relation mapping procedure Rmap described in e.g. [4], rules for translation to SQL can be worked out. Although the translation is not exactly one-to-one (since *one* table in the relational database corresponds to a *set* of fact types in ORM), the DML counterparts of our ORM-constructs in SQL are:

ADD VALUES	⇒	INSERT VALUES
ADD RESULT	⇒	INSERT <i>query</i>
REMOVE	⇒	DELETE
CHANGE	⇒	UPDATE

Our proposal for transactions was inspired by (the expressiveness of) the DML of SQL. The expressiveness of our DML for ORM also depends on the expressiveness of the ConQuer language and could now be studied formally.

Acknowledgments

This paper clearly benefited from discussions with Herman Balsters, who also helped in drawing the ORM-diagrams.

References

- [1] H. Garcia-Molina, J.D. Ullman, J.D. Widom, *Database Systems: The Complete Book*, second ed. Prentice Hall, Upper Saddle River, 2009.
- [2] Object Role Modeling, (<http://www.orm.net>) (accessed on 01.03.16).
- [3] The ORM foundation, (<http://www.ormfoundation.org>) (accessed on 01.03.16).
- [4] T. Halpin, T. Morgan, *Information Modeling and Relational Databases*, second ed. Morgan Kaufmann, San Francisco, 2008.
- [5] A. Bloesch, T. Halpin, ConQuer: a conceptual query language, in: *Proceedings of the 15th International Conference on Conceptual Modeling*, ER'96, Springer, Heidelberg, vol. 1157, 1996, pp. 121–133.
- [6] A. Bloesch, T. Halpin, Conceptual queries using ConQuer-II, in: *Proceedings of the 16th International Conference on Conceptual Modeling*, ER'97, Springer, Heidelberg, vol. 1331, 1997, pp. 113–126.
- [7] P. Chen, *The entity-relationship model – toward a unified view of data*, *ACM Trans. Database Syst.* 1 (1) (1976) 9–36.
- [8] The Unified Modeling Language, (<http://www.uml.org>) (accessed on 01.03.16).
- [9] A. Bloesch, T. Halpin, Data modeling in UML and ORM: a comparison, *J. Database Manag.* 10 (4) (1999) 4–13.
- [10] T. Halpin, Data modeling in UML and ORM revisited, in: *Proceedings of the 4th IFIP WG8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design*, Heidelberg, EMMSAD, 1999.
- [11] H. Balsters, A. Carver, T. Halpin, T. Morgan, Modeling dynamic rules in ORM, in: R. Meersman, et al., (Eds.), *OTM Workshops 2006*, vol. 4278, Springer, Heidelberg, 2006, pp. 1201–1210.
- [12] H. Balsters, T. Halpin, Formal semantics of dynamic rules in ORM, in: R. Meersman, et al., (Eds.), *OTM Workshops 2008*, vol. 5333, Springer, Heidelberg, 2008, pp. 699–708.
- [13] E.O. de Brock, Transactions in ORM, in: T. Halpin, et al., (Eds.), *BPMDS 2009 and EMMSAD 2009*, vol. 29, Springer, Heidelberg, 2009, pp. 291–301.
- [14] G.P. Bakema, J.P.C. Zwart, H. van der Lek, Fully communication oriented NIAM, in: G.M. Nijssen, J. Sharp (Eds.) *Proceedings of the NIAM-ISDM 1994 Conference Working Papers*, Albuquerque, USA, pp. L1–35.
- [15] E.O. de Brock, *De Grondslagen van Semantische Databases*, Academic Service, The Hague, 1989.
- [16] E.O. de Brock, *Foundations of Semantic Databases*, Prentice Hall International Series in Computer Science, Hemel Hempstead, 1995.
- [17] E.O. de Brock, Declarative specifications of complex transactions, with an application to cascading deletes, in: *Post-Proceedings of the TDD'99-Conference*, LNCS, Springer, Berlin, vol. 1773, 2000, pp. 150–166.
- [18] M. Albert, J. Cabot, C. Gómez, V. Pelechano, Generating operation specifications from UML class diagrams: a model transformation approach, *Data Knowl. Eng.* 70 (4) (2011) 365–389.