The paper introduces the principles of object-oriented translation for target machine which provides executing the sequences of elementary operations on persistent data presented as a set of relations (programmable relational system). The language of this target machine bases on formal operations of relational data model. An approach is given to convert both the description of complex object-oriented data structures and operations on these data, into a description of relational structures and operations on them. The proposed approach makes it possible to extend the target relational language with commands allowing data to be described as a set of complex persistent objects of different classes. Object views are introduced which allow relational operations be applied to the data of complex objects. It is shown that any operation and method can be executed on any group of the objects without explicit and implicit iterators. Binding of both attributes and methods with their polymorphic implementations are discussed. Classes can be co-used with relations as scalar domains, in referential integrity constraints and in data query operations.

Categories and Subject Descriptors: F1.1 [COMPUTATION BY ABSTRACT DEVICES]: Models of Computation; D3.3 [PROGRAMMING LANGUAGES]: Language Constructs and Features; H2.3 [DATABASE MANAGEMENT]: Languages --- Database (persistent) programming languages; H2.4 [DATABASE MANAGEMENT]: Systems

General Terms: Languages, Design, Theory

Additional Key Words and Phrases: object-oriented language, translation, target machine, data base management system, data independence, data structures, relation, translation, persistent object, relational data model, group operation, ad-hoc query, progress operations

1. INTRODUCTION

Existant object-oriented languages are result of an evolution of programming languages for computers which use addressable memory [Pratt and Zelkowitz. 2001]. Such memory organization is not only in programmable systems. In particular, relational database management system can be considered as virtual computer providing executing the sequence of elementary operations on persistent data presented as a set of relations. We will name such computer "programmable relational system" (PRS). Implemented in PRS, the data structures and elementary operations on the data are described by relational data model [Codd.1970]. The most important property implemented by PRS is data persistence.

We offer approach to creation object-oriented translator for PRS which is target machine in this case (further: relational target machine). Source language of the translator can be described as declarative object-oriented language. Its commands allow the next. Classes of unique objects having complex (non 1NF) structures can be created. References are allowed. Constraints can be defined on the classes. The objects encapsulate persistent state and behavior. Procedural extensions are used to implement the class methods. The object properties can be redefined in multiple class inheritance. Persistent objects can be created and manipulated by both data access commands and methods. The objects data can be received used ad-hoc queries.

It may be interesting that all operations on complex objects including object methods execution can be performed on any set of the objects without both explicit (described by source language) and implicit (described by target machine language) iterators.

Let us note that command translation is used which differs from program translation as it is usually implied when a object-oriented translation is being discussed. The main difference is that program translation is a process precedes the program execution. As a result, all possible program data structures and procedures on the data are fixed by program code and cannot be changed during execution of the program. Command translation is continuous process concurring with system life. It makes possible to change data descriptions and system functionality. To perform command translation persistent symbol tables have to be used which contain all object metadata and can be considered as part of system catalog.
Our approach allows existing relational DBMSs be extended in the direction set by the "Third-Generation Database System Manifesto" [Stonebraker at all. 1990] but in a different way than offered by current SQL standards [ISO/IEC 9075:2008. 2008] and existing object-relational DBMSs. Offered interpretation of concept "class" and "object" is similar to the ones used in traditional object-oriented languages [Booch. 1991]. We consider a class as a set of unique persistent objects so tables are not necessary to group the objects and to set them as persistent. Reached object persistence is a consequence of total data persistence implemented by relational target machine. Relation target machine also allows the situation be avoided when source object-oriented language includes some awkward features dictated by target machine with addressable memory. For example extents and inverse references used in ODMG data model [Cattell at all. 2000] are explainable only by usage of such target machine where one-direction pointers are the only way to get access to the data stored in the memory.

All data described as complex objects of different classes are presented in form of normal relations (object views) which are calculated according to names and name sequences set by the class description. The object views keep the meaning of the complex objects structures inside the names and headers. The object views (i.e. normal form relations) are the only way to operate with the data of complex objects; therefore no other formal data model than the relational one is necessary to operate with the data. Actually the classes and their implementations are orthogonal to relational data model as it required by [Date and Darwen. 1998].

Offered approach has implemented in prototype described in [Grigoriev. 2011, 2012].

We proceed as follow. In section 2, the PRS definition of is given and its control commands are listed. These commands are used further to write the results of translation. In section 3, common properties of classes and objects, implemented by offered approach are formulated according to features of relational target machine. In section 4, source language commands and principles of their translation are described. In section 5, compatibility of classes and relations in the system is discussed. New operation of object progress through class hierarchy is introduced in conclusion part.

2. PROGRAMMABLE RELATIONAL SYSTEM

PRS maintains an existence of a relational database. Here the term "relational database" is used in a formal way. According to [Date and Darwen. 1998] a relational database is set of named relational variables \( R \) (relvars) which is interacted by set of transactions \( tr \). The relational variables can be real (stored) or virtual (calculated). A set of relvar \( R \) definitions together forms the schema of a database. Values of the relvars \( R \) together form total database value. Transactions \( tr \) are used to change the database value.

PRS is operated by declarative language which allows procedures be defined, stored and executed. The procedures may take parameters and use local variables. In the paper the next five PRS commands will be used

1) To create and describe relational variable the next command is used:

\[
\text{CREATE } R(a_1:D,... a_n:D) \text{ KEY}(...a_i...) \text{ FKEY}(...) \text{ ON}(...);
\]

where \( R \) is a name of new relational variable, \( a \) is an attribute of relation, \( D \) is a domain of the attribute. PRS is supposed to implement a finite set of domains (i.e. base scalar types e.g. INTEGER, FLOAT etc.) and operations on them. \( \text{KEY} \) is a part defining integrity constraints (keys), \( \text{FKEY} \) \( (...) \text{ ON}(...) \) is an optional part defining referential integrity constraints (foreign keys).

Virtual relvars are created with a command binding the described variable name \( R \) with expression \( RValue \) which returns a relational value.

\[
\text{CREATE } R \ldots \text{ AS } RValue,
\]

where \( RValue \) expression is one of the next:

- A name \( R \) of other relational variable.
- A composition \( f(...)RValue(...) \) of relational algebra operations \( op \) on results of other \( RValue \) expressions \( op_1(...) \text{ op}_2(RValue, \text{ op}_3(...)) \). Next relational operations on \( RValues \) are used further:
  - \( R_1 \times R_2 \) – Cartesian product,
  - \( R_1 \cup R_2 \) – union operation,
- $R_1 - R_2$ – difference operation,
- $R_1 \text{ JOIN}_{\text{criteria}} R_2$ – join operation (join criteria is in lower index). LEFT JOIN \text{criteria} operation is also used.
- $R[a_1, a_2, \ldots]$ – projection operation ($a_i$ is attribute),
- $R \text{ WHERE}$ criteria – selection.
- $R \text{ RENAME} a \text{ AS} b$ – attribute renaming operation.

This relational operation can contain scalar operations on values of attributes $a$, which are defined for domains $D_i$ of the attributed.

- A procedure which returns the value by return $RValue$ operator.

  begin
  ...
  return $RValue$;
  end;

- An explicit relational value given by user.

2) To set values of relational variables the assignment command is used

$\text{SET} R := RValue$

If the right part of assignment command contains explicit relational value, such command is considered as data input one. This command is equal to traditional relational commands INSERT, UPDATE, DELETE [Date and Darwen, 1998] which also are used further.

3) To get values from relational database the next command is used.

$\text{GET} RValue$

This command is considered as data output one and can be used as ad-hoc query. It doesn't change the total database value.

4) We consider transactions as sequences of elementary operations changing the database state (i.e. as procedures), either all occur, or nothing occurs. They may contain commands changing database schema or/and values of relational variables. A transaction $tr$ can be defined and stored with the next command

$\text{TRANS} \ tr ($parameters_definition$)$

AS begin
  ...
  create $R(\ldots a_i : D_i \ldots)$;
  $\text{SET} R := RValue$;
  ...
end;

5) To execute the transaction $tr$ the next command is used

$\text{EXEC} t$r$ (parameters);

Direct execution of transaction formed of sequence of command is also possible in PRS.

$\text{EXEC begin}$
  ...
$\text{end}$;

These commands CREATE, SET, GET, TRANS and EXEC are used further to write result of translation.

3. CLASSES AND OBJECTS

Our main aim is a language consisting of commands which allow persistent data be described as a set of unique complex objects of different classes. Let us formulate how these common object-oriented concepts may be implemented considering the features of the relational target machine.

In programming languages any complex data object is built as a set of simplest data objects of data types which are realized by target machine [Pratt and Zelkowitz, 2001]. We use this principle to build a complex data object using data types realized by the relational
target machine. In this case, an object variable is a set of named components which are relational variables.

(Remark. The term "object components" is equal to "object attributes". We use it in further text to distinguish "object components" from "attribute of a relation".)

Let us note that a relation have two characteristics: arity(A) and cardinality(M). Usually these characteristics are implied to have any values. But object components can be described by simpler language constructions. Such description fixes one or both relation characteristics to unity. For example, components described as a tuple can be considered as a relation variable with cardinality fixed to unity. As a result the next kinds of components are possible: relations (A:n, M:n), sets (A:1, M:n), tuples (A:n, M:1), scalars (A:1, M:1). It is correct to say that the target relation machine allows any object value be described as a set of values which are not more complex than relations.

(Remark. Further only simplest scalar components (simple ones) and most complex relation components (complex ones) are considered for sake of simplicity. The sets and the tuples can be reasoned by analogy.)

Objects also have a behavior which is defined by a set of a methods allowing state of the objects be changed.

The components and the methods together form an object interface. A class is a set of objects having the same interface; this interface is described in class specification. The objects can be accessed as elements of the class. Classes can be inherited. Multiple inheritance is possible. Child class specification is result of UNION set operation on specifications of parent classes and the set of own components and methods (virtual inheritance).

A class specification is distinguished from an implementation of the class. The implementations are set separately for each component and method. Class components can be stored or calculated in different ways. Thus persistence of data is the property encapsulated in object components and defined by their implementation. Class methods are implemented by procedures. All the implementations can be redefined during inheritance.

Each object is unique among other ones. This common object property is implemented in relational target machine with OID values of dOID domain. The OID value is not associated with the object state. It is generated by the RPS at object creation and stays unchanged during all object life. The OID values are not accessible for users.

All OIDs of objects of some class are united in corresponding reference type having the same name. Next operations are defined for reference variables: assignment, comparison and dereferencing operation. Reference types are scalar ones which the set of domains D is being expanded by.

Besides the OID, the objects uniqueness inside their classes can be defined by optional explicit keys. With them, explicit foreign keys are possible too.

4. SOURCE LANGUAGE COMMANDS AND TRANSLATION

This part describes main commands and expressions of the source language and gives principles of their translation into machine language commands. Further the source language commands and the expressions are discussed in next order:

- Class creation command,
- Data access commands,
- Implementation command (implementing expression),
- Implementation command (binding expression),
- Object creation command.

For each of the listed group of commands, their essence is described firstly. Then the principles of their translation are given. Examples are used when necessary. The syntax of source language used in the examples is close to the SQL and to the one realized in prototype described in [Grigoriev, 2011].

Let us start with a common example which uses commands listed above. a simple model of a trade company is created in the example. Classes are specified and implemented and objects are created. Data structure created by these command is shown. Data access operations (inc. queries) are given. Then the classes are inherited and re-implemented and objects of child classes are created too. Group method execution and data access operations to polymorphic classes are illustrated.
4.1 Common example

Classes are created and specified by command CLASS ... (this command is described in 4.2). The next classes is used as example ones in the paper. Simple class BANKS contains data on banks serving contractors.

CLASS BANKS
( Name STRING );

Contractors are unique with their IDs. Also each of class CONTRACTORS objects references on some BANKS object.

CLASS CONTRACTORS
( Name STRING,
   Bank BANKS,   //reference
   ID STRING
)KEY(ID);

Class GOODS describes assortment of sold goods which are unique by their nomenclature articles. Objects of class GOODS also contains information on the goods turnover and current quantity of pieces on stock.

CLASS GOODS
( Art STRING;
   Turnover SET OF   //complex component
   ( DocN STRING,
      Cntr CONTRACTORS,   //reference
      Pieces INTEGER
   )KEY(DocN),   //component key
   Pieces INTEGER   //...remain on stock
)KEY(Art);

Documents on deliveries and shipments are described by class DOCS. Documents are unique with their number DocN. This class contains method DoShip. The attribute Art of complex components Items defined as foreign key on key component Art of class GOODS.

CLASS DOCS                       //
(E01)
( DocN STRING,
   Date DATETIME,
   Comment STRING,
   Cntr CONTRACTORS,   //reference
   DoShip(inDate DATETIME),   //method
   Items SET OF   //complex component
   ( Art STRING,
      Pieces INTEGER
   )KEY(Art)   //component key
)KEY(DocN)   //class key
REFERENCE Items(.Art)           //foreign key
ON GOODS(.Art)                  //foreign key

Let us note that ad-hoc queries on classes can be executed right after the classes are defined (data access commands are described in 4.3) f.e. the next query.

SELECT #S.DocN,
   #S.Comment,
   #S.Items.Art,
   #S.Items.Oty
FROM DOCS<DocN LIKE "%1"> #S; // #S is alias

In this point classes is not implemented so result of the query is empty. Class must be implemented fully to allow objects be created. Some class components can be implemented as stored (implementation command is described in 4.4).
ALTER BANKS REALIZE Name AS STORED;
ALTER CONTRACTORS REALIZE Name, Bank, ID AS STORED;
ALTER GOODS REALIZE Art AS STORED;
ALTER DOCS REALIZE DocN, Date, Comment, Cntr, Items AS STORED;

Other class components can be implemented as calculated. E.g. complex component Turnover is implemented in class GOODS as calculated by a query on data presented in class DOCS

ALTER GOODS REALIZE Turnover AS
SELECT #g.DocN,
#g.Cntr,
SUM(#g.Items.Pieces) AS Pieces
FROM DOCS #g
WHERE #g.Items.Art = Art
GROUP BY
#g.DocN,
#g.Cntr;

Here, in comparition #g.Items.Art = Art last Art is the name of simple component of class GOODS. Simple component Pieces is implemented in class GOODS by the next procedure

ALTER GOODS REALIZE Pieces AS {
DECLARE tmpPieces INTEGER;
tmpPieces :=
SELECT SUM(#g.Items.Pieces) AS Pieces
FROM DOCS #g
WHERE #g.Items.Art = Art;
IF(tmpPieces IS NULL)
THEN tmpPieces := 0;
RETURN tmpPieces;
}

Methods are implemented by a procedure. Method DoShip is implemented by the next procedure in class DOCS

ALTER DOCS REALIZE DoShip(inDate DATETIME) AS {
IF(Date IS NULL) THEN
BEGIN
Date := inDate;
Comment := "Shipped!";
END
}

After a class was implemented fully its objects can be created. Command NEW is used to create objects (this command is described in 4.6). In this command a part after WITH SET contains constructing expressions.

NEW CONTRACTORS WITH SET
.Name:="TheShop",
.Bank := (NEW BANKS WITH SET
 .Name:="TheBank"),
.ID:="CoID001";

This command contains nested NEW command; so reference on new-created BANKS object is used to initialize the reference component Bank of new-created CONTRACTORS object. Objects are available as elements of class. In the next command the reference component Bank is initialized with reference on existing object using expression FIRST OF
BANKS<.Name="TheBank"> returning reference on the only object of class BANKS which satisfies to condition <.Name="The Bank">.

NEW CONTRACTORS WITH SET .Name:="TheRetail", .Bank:=(FIRST OF BANKS<.Name="TheBank">), .ID:="CoID002";
NEW GOODS WITH SET .Art:="Tie";
NEW GOODS WITH SET .Art:="Axe";
NEW DOCS WITH SET .DocN:="Ship1", .Cntr:=FIRST OF CONTRACTORS<.Name="TheShop">;
NEW DOCS WITH SET .DocN:="Ship2", .Cntr:=FIRST OF CONTRACTORS<.ID="CoID001">;
NEW DOCS WITH SET .DocN:="Ship3", .Cntr:=FIRST OF CONTRACTORS<.ID="CoID002">;

Existing objects can be modified
INSERT INTO DOCS<.DocN = "Ship1">.Items (Art, Pieces) VALUES ("Axe", 2);
INSERT INTO DOCS<.DocN = "Ship2">.Items (Art, Pieces) VALUES ("Axe", 5);
INSERT INTO DOCS<.DocN = "Ship2">.Items (Art, Pieces) VALUES ("Tie", 10);

Figure 1 shows data structure being result of the given commands execution. Let us note that (according to implementation) the values of components Turnover and Pieces (underlined) of class GOODS objects are calculated from values stored in class DOCS.
BANKS

| Name   |
|--------|
| TheBank|

CONTRACTORS

| Name  | Bank ID |
|-------|---------|
| TheShop | CoID001 |
| TheRetail | CoID002 |

GOODS

| Art  | Turnover | (DocN, Cntr, Pieces) |
|------|----------|----------------------|
| Axe  | Ship1    | 2                    |
|      | Ship2    | 5                    |
| Tie  | Ship2    | 10                   |

DOCS

| DocN | Date | Comment | Cntr | Items |
|------|------|---------|------|-------|
|      |      |         |      |       |
| Ship1|      |         |      | Axe   |
|      |      |         |      | 2     |
|      |      |         |      | Ship2 |
|      |      |         |      | Axe   |
|      |      |         |      | 5     |
|      |      |         |      | Ship3 |
| Ship2|      |         |      | Tie   |
|      |      |         |      | 10    |

Fig. 1. Example object data structure
(Each rectangle contains data on separate object, arrows mean references. Values of calculated components are underlined).

Now the earlier used ad-hoc query

```
SELECT #S.DocN,  
   #S.Comment,  
   #S.Items.Art,  
   #S.Items.Oty  
FROM DOCS<DocN LIKE "%1"> #S;  
```

returns the next result

```
DocN   Comment   Items  Items    
------- -------    ------ ---------  
Ship1   Axe       2     
```

Here expressions "Items.Art" and "Items.Pieces" are used as a name of relational attribute keeping the meaning of complex structure defined by class specification.

Suppose a class SALES inherits both the existing class DOCS and new class VALUERECORDS. There is complex component SoldItems containing data on sold goods in new class SALES.

```
CLASS VALUERECORDS  
( ...  
   Amount FLOAT,...  
) ...  
```

The new component SoldItems is implemented as stored too.

```
ALTER SALES REALIZE SoldItems AS STORED;  
```
Implementations of both inherited component Items and method DoShip are changed in this class. Inherited component Items is calculated now.

```
ALTER SALES REALIZE Items AS
  SELECT Art, SUM(Pieces) FROM SaleItems GROUP BY Art;
```

Now inherited method DoShip() initializes inherited components of both parent class DOCS and VALUERECORDS.

```
ALTER SALES REALIZE DoShip(inDate DATETIME)
AS {
  IF(Date IS NULL) THEN
  begin
    Date := inDate;
    Comment:= "Sold!";
    Amount:=SELECT SUM(#si.Pieces,#si.Price) FROM SalesItems #si;
  end
}
```

New object of class SALES is created and modified

```
NEW SALES WITH SET
  DocN := "Sale1",
  Cntr:=FIRST OF CONTRAGENTS<.ID="CoID002">;
```

```
INSERT INTO SALES<.DocN="Sale1">.SaleItems (Art, Price, Pieces)
VALUES ("Tie", 10, 30);       //...
```

```
INSERT INTO SALES<.DocN="Sale1">.SaleItems (Art, Price, Pieces)
VALUES ("Tie", 11, 20);       //the same Art but other Price
```

```
INSERT INTO SALES<.DocN="Sale1">.SaleItems (Art, Price, Pieces)
VALUES ("Axe", 20, 50);
```

Let us execute the method for some of parent class objects

```
EXEC DOCS<DocN LIKE "%1">.DoShip(...);
```

Figure 2 shows data structure being result of the given commands execution. Let us pay attention to data dependences set by implementation of the classes. Underlined values are calculated ones. In each object of class SALES, its component Items inherited form class DOCS and component Amount inherited form class VALUERECORDS are calculated from stored component SaleItems. The values of components Turnover and Pieces (underlined) of class GOODS objects are calculated from values of components Items which is stored in class DOCS and calculated in child class SALES.
| GOODS | (Art | Turnover | Pieces) |
|-------|-------|----------|----------|
|       | (DocN | Cntr :  | Pieces) |
| Axe   | Ship1 | ● : 2    | 57       |
|       | Ship2 | ● : 5    |          |
|       | Sale1 | ● : "50" |          |
| Tie   | Ship2 | ● : 10   | 60       |
|       | Sale1 | ● : 50   |          |

| DOCS | (DocN | Date | Comment | Cntr | Items | (Art : Pieces) |
|------|-------|------|---------|------|--------|----------------|
|      | Ship1 |      | Shipped!| ●    | Axe    | 2              |
|      | Ship2 |      |         | ●    | Tie    |                |
|      | Ship3 |      |         | ●    |        |                |

| VALUERECORDS | (Amount) |
|--------------|----------|

| SALES | (Art : Price) | SaleItems |
|-------|----------------|------------|
|       |                |            |
| Sale1 | Sold!          | ● Axe 50   | 1520       |
|       |                 | ● Tie 50   | 30         |
|       |                 | ● "50"    | 20         |

Fig.2. Example object data structure after all changes. Unchanged classes BANKS and CONTRACTORS are not present.
(Each rectangle contains data on separate object, dots mean references. Values of calculated components are underlined).

Let us execute the same query again

```
SELECT #S.DocN, #S.Comment, #S.Items.Art, #S.Items.Oty
FROM DOCS<DocN LIKE "%1"> #S;
```
Now the result of the query is

| DocN   | Comment     | Items   | Items   |
|--------|-------------|---------|---------|
| Ship1  | Shipped!    | Axe     | 2       |
| Sale1  | Sold!       | Tie     | 50      |
| Sale2  | Sold!       | Axe     | 50      |

Column Comment shows that proper implementation of method DoShip(...) was executed for each of objects. Columns Items.Pieces shows other type of polymorphism. Value 2 is stored in complex component Items (as it is implemented in class DOCS) and the values 50 is result of the component calculation, as it is implemented in class SALES. The correct implementations are bound during the query execution (the binding is described in 4.5).

The query is applied to class DOCS only. But ability to inherit the class and to redefine its implementations allows data presented in result of the query be stored and/or calculated in different ways. It's possible to expand the data schema with new classes and to get new data by unchanged data access operations over classes.

Let us note that after method DoShip() execution the query on abstract class VALUERECORDS

```
SELECT #vr.Amount FROM VALUERECORDS #vr;
```

returns (according to implementations of child class SALES) the amount of the posted sale

```
Amount
--------
1520
```

4.2 Class Creation Command

The command creates class (i.e. set of objects) and describes its specification.

4.2.1 Description of Class Creation Command

Class creation command has the follow syntax

```
CLASS D_n+1 EXTEND D_n ...
     (=C_i: D_a, ...) 'C_i(a_i:D_a, ...)KEY(a_i...) , ...
     M_i(...),...
) KEY(=C_i...)
REFERENCE (...) ON (...)
```

where D_n+1 – name of new class (and corresponding reference type), EXTEND D_n – optional part listing parent classes; further (in parentheses) class are described with, =C_i – simple components defined on some D_a domain, ‘C_i(a_i:D_a, ...)KEY(a_i...) – complex components (relation with schema given), M_i(...) – method; KEY(=C_i...) – optional part describing integrity constrains (class keys consisting of scalar components), REFERENCE (...) ON (...) – optional part describing referential integrity constraints (foreign keys).

4.2.2 Translation of Class Creation Command

During translation of class creation command the translator performs the next action.
- All information about new class (names, structures, links etc.) is analyzed and entered into symbol table
- In relational target machine a set of relations are created which represent objects data.

The objective of class specification command translation is to create a schema of representation of object value structure into data structures possible in target machine (i.e. in relations). Such schema is basis for translation of other commands manipulating objects of the class. The object representation into relation is based on already formulated condition: any object state can be described as set of values which are not more complex than relations. It
allows the any object value structure be represented into relations almost directly. The main principle is the next: different components are represented in different relations. But since different kinds of components exist, so the representations schemas are different for the different kinds of components.

- Values of all scalar components \( \text{sc}_C \) of all class \( D \) objects are represented in single relation \( R_D \) (further – class scalar relation)

\[
( \text{OID} \times 1 \times \text{sc}_C^1 \times 1 \times \text{sc}_C^2 \times \ldots \times 1 \times \text{sc}_C^n ) \cup ( \text{OID} \times 2 \times \text{sc}_C^1 \times \ldots \times 2 \times \text{sc}_C^n ) \cup ( \ldots ) \cup \ldots \rightarrow R_D,
\]

where \( \text{OID} \) is identifier of some object, \( \text{sc}_C \) is some scalar component of this object, \( n \) is number of scalar components in class \( D \)

One class scalar relation \( R_D \) conforms to each class \( D \). One and only one tuple of the class scalar relation conforms to each object of the class. \( \text{OID} \) attribute is mandatory in the scalar relation. Other attributes of the scalar relation conform to class scalar components exactly.

- \( \text{OID} \) is key attribute.
- If scalar class components form class key, the corresponding attributes form scalar relation key.

- Values of each complex component \( \text{r}_C \) of all class \( D \) object are represented in relation \( R_D.C_i \) (further – class relation of complex component).

\[
( \text{OID} \times 1 \times \text{r}_C^1 ) \cup ( \text{OID} \times 2 \times \text{r}_C^1 ) \cup ( \ldots ) \rightarrow R_D.C_1
\]

\[
( \text{OID} \times 1 \times \text{r}_C^2 ) \cup ( \ldots ) \rightarrow R_D.C_2
\]

\[
( \ldots ) \rightarrow R_D.C_m
\]

Here \( \text{OID} \) is identifier of some object, \( \text{r}_C \) is some complex component of this object, \( m \) is number of complex components in class \( D \).

Number of the relations \( R_D.C_i \) is equal to number of set components. \( \text{OID} \) attribute is mandatory in the relation \( R_D.C_i \). Other attributes of the relation \( R_D.C_i \) conform to attributes of corresponding complex component exactly.

- Key of the set relation aggregates \( \text{OID} \) attribute and the attributes defined as key ones in corresponding component .
- If attributes of set component form global unique constraint on complex component tuples, then corresponding attributes form key of relation \( R_D.C_i \).

(remark. Such constraint is not class key because complex components can be empty.)

- If scalar components or attributes of complex components form a foreign key, then the corresponding attributes of corresponding class relation form foreign key too.

- At that, all references (simple reference components and reference attributes of complex components) are represented in class relations as attributes defined in domain of object identifiers \( \text{dOID} \).

Thus all data of all class \( D \) objects are represented in one scalar relation \( R_T \) and several relations \( R_D \), named together the class relations. The number of class relation depends on class structure only. At that all constraints defined for class \( T \) are represented in constraints set on the class relation. If a class is inherited form other classes its data are represented in both own class relations and class relations of all parent classes.

The described process can be informally defined as relational memory configuration. It's very important that the class relations are not stored but virtual ones. They allows data be presented in relation target machine according to class specification. The ways of how the data are calculated to be presented in class relations depend on components implementations only. On this stage of discussion the class relations are accepted as existing and containing data of all objects of the class. Details on class relations calculation will be discussed further (see 4.4 and 4.5).

Example. Source command of class \( \text{DOCS} \) creation (E01) is translated into the next sequence of relational target machine commands which create two class relations.
EXEC begin
    CREATE RDOCS (OID: dOID, DocN:STRING, Date: DATETIME, Cntr:dOID)
        KEY(OID) KEY(DocN)
        FKEY(Cntr) ON RCONTRACTORS(OID) AS …;
    CREATE RDOCS_Items (OID: dOID, Art: STRING, Pieces:INTEGER)
        KEY(OID, Art)
        FKEY(Art) ON RGOODS(Art) AS …;
end

4.3 Data Access Commands

Data access commands are used both to change and to get data described as a set of complex objects. This part devoted mainly to object views which represent data operated by data access commands. The object views are described and then operations which calculate object view are given.

4.3.1 From Paths to Relations

All data access commands use reference path expressions (paths). The paths are name sequences determined by structures and references defined in class specification. Idea of paths - to present hierarchy, to specify a part of a general - seems obvious enough.

Dot notation is used to write a path. Simplest paths include just one name. Type of the path is defined by type of its last element.

Paths used in data access command begin with any expression which means set of objects. In a global context (where commands are executed) any paths begin with a name of a class which are only names defined in global context.

The paths of base scalar types are terminal ones and don’t allow continuations. The paths ended with name of reference or with name of complex component have continuations (further : post-paths). Such paths will be named non-terminal.

Example. Path DOCS.Cntr (of reference type CONTRACTORS) is non-terminal and has next post-paths among other.

.Name
.Bank.Name

Let us note that post-paths begin with a dot in used source language syntax.

Any name of a class or reference existing in a path can be added with object selection expression.

name_of_class_or_reference<conditions_list>

This expression restricts the set of objects defined by the path using conditions_list. Here each of conditions is applied to scalar post-paths available after name_of_class_or_reference. Conditions can be combined with traditional logical operation AND, OR and NOT. Also new low-priority logical operation is added. Its name is intertuples_AND. Simple comma ",” are used to write this operation in source code.

The necessity of the new logical operation is dictated by the fact that complex components are relations containing set of tuples and selection cases inexpressible by usual in-tuple logical operations are possible. The example of such case is given below.

Example. To access class DOCs objects which contain in attribute Art of complex component Items both Tie and Axe string values (in different tuples) next path is used

DOCS<.Items.Art = "Tie", .Items.Art = "Axe">

Here the objects have to be selected which contains the tuples in complex component .Items, which satisfy to all different conditions applying to the same attribute .Art. Traditional in-tuple operation OR cannot be used to combine the conditions because it allows object be selected which satisfy only to one of given conditions. Traditional in-tuple operation AND cannot be used because the scalar tuple attribute cannot be equal to different values simultaneously.
Object selection expression can be nested and combined arbitrarily.

The next is base principle of how the paths are used in data access commands:

Any non-terminal path can be considered as a name of a relation; any scalar post-paths of this path can be considered as names of attributes of this relation. Such relations will be named as O-views (object views) further.

Example. Expression `DOCS<.Date >= '2010.01.01'>.Cntr` is non-terminal path which has, among other, post-paths `.Name`, `.ID` and `.Bank.Name`. It means, that the next O-view (i.e. relation) can be used:

```
DOCS<.Date >= '2010.01.01'>.Cntr
(`.Name` ,
`.ID` ,
.`Bank.Name`
)
```

Multitude of other O-views can be used, e.g.

```
BANKS
(`.Name`
)
```

```
GOODS.Turnover
(`.Cntr.Name`, `.DocN`, `.Date`, `.Pieces`
)
```

```
GOODS<.Art LIKE "Sometext%">.Turnover
(`.Cntr.Name`, `.DocN`, `.Date`, `.Pieces`
)
```
etc. and etc.

Here, the path expression (e.g. `GOODS.Turnover`) is considered as just name of relation i.e. as string identifier which differs from other name (e.g. `GOODS<.Art LIKE "Sometext%">.Turnover`). Also expressions like `.Art` , `.Pieces` `.Turnover.DocN` , `.Turnover.Date` etc. are considered as unique names of the view attributes. All these names keep the meanings of structures which exist in complex objects. They allow the data be presented in form of normal relations, using the names and name sequences defined in class specification. As a result, transition from complex object description to relational representation of the object data is imperceptible for users.

There is no certain relationship between the set of classes and the set of O-views. One class can relate to many O-views. One O-view can present data of different classes (combined by references and/or in inheritance). The set of classes and the set of O-views are linked with common names only. It’s possible to say that each of the O-views is not defined separately but the total set of O-views (i.e. relations) is defined by set of classes. In this way all data of complex objects are represented for user simultaneously as a set of normal relations.

### 4.3.2 O-views in Data Access Commands

O-views are relations. Therefore usual relational command can be applied to O-views to access complex object data.

Example. Next command changes value of simple components of some objects

```
UPDATE CONTRACTORS<.ID = "..."> SET (.Name := "...");
```

Example. Next command adds tuple into complex components of some objects

```
INSERT INTO DOCS<.DocN = "...">.Items (.Art, .Pieces)
VALUES("Hat", 1); //...no such "Art" - error
```

Example. Next query returns the data from objects forming complex reference structure
SELECT
    #gt.DocN,
    #gt.Cntr.Name,
    #gt.Cntr.Bank.Name
FROM GOODS<.Art LIKE "A"> .Turnover #gt;

The result is

| DocN | Cntr       | Cntr           |
|------|------------|----------------|
|      | Name       | Bank           |
| Ship1| TheShop    | TheBank        |
| Ship2| TheShop    | TheBank        |
| Sale1| TheRetail  | TheBank        |

Let us note that O-views are defined only by class specifications. Therefore if class are
defined (this means that class specification are defined too) it is possible to execute commands
to access class objects data even if no object of the class exists or the class is not implemented
yet. In this way class is considered by users as a set of objects which are accessible as
elements of the class. Object references and collections of the references (e.g. class extents)
are not obvious to access data of the objects with this ability.

Let us note also, that offered approach allows data be accessed both by references and
against references.

Example. It's possible to find data of class CONTRACTORS objects referenced by defined
class DOCS objects.

SELECT
    #c.Name
    #c.ID
    #c.Bank.Name
FROM DOCS[.DocN LIKE "%1"] .Cntr #c;

The result is

| Name  | ID      | Bank   |
|-------|---------|--------|
|       | Name    |        |
| TheShop| CoID001 | TheBank|
| TheRetail | CoID002 | TheBank|

Also it's possible to find data of class DOCS objects referencing to defined class
CONTRACTORS objects.

SELECT
    #d.DocN
    #d.Items.Art
    #d.Items.Pieces
FROM DOCS[.Cntr.Name = "TheRetail"] #d;

The result is

| DocN | Items | Items |
|------|-------|-------|
|      | .Art  | .Pieces|
| Ship3| Axe   | 50    |
| Sale1| Tie   | 50    |

4.3.3 Translation of Data Access Commands

O-views are not defined manifestly like e.g. SQL views are defined. O-views can be
calculated. Any command which manipulates object data contains anyhow some path and its
post-paths which together form O-view signature. Generally translation of the data access command includes the next steps:

1. The command is parsed to find O-view signatures.
2. Expressions are built which calculate the O-views defined by found signature.
3. In the command, all expressions formed O-view signatures are replaced with expressions calculated the O-views.

Thus the main point of the data access commands translation is O-view calculation. Further set of operations used for O-view calculation is considered. Their base operands are class relations \( R \) described in previous part. Their results are relations \( O \). Mandatory OID attribute identifies objects whose data are contained in the relations. Let us note that this attribute is not defined in O-view signature explicitly. Considering that O-views are used in commands only and the results \( O \) of their calculation aren't accessed directly, the OID attribute is implicit for the users (this is logically for attribute containing system values). The OID attribute is important for system functioning. It allows the structure of the result relations be unified with class relations; so the result relations can be used in further calculation as well as the class ones. Also it's possible to get OID value (i.e. reference on the object) by values of the object components.

### 4.3.3.1 Simple Projection

If O-view signature is a subset of one of class relations \( R \), then simple projection

\[
GET \ R[OID, \ldots]
\]

is used to calculate it.

*Example.* O-view \texttt{DOCS.Items(.Art)} is calculated as projection

\[
GET \ R_{DOCS.items}[OID, Art].
\]

### 4.3.3.2 In-Class Join

Suppose O-view combines data from different class relations of some class \( D \); then result is calculated by \texttt{LEFT JOIN} of class scalar relation on common OID attribute. Also attributes of class relations of complex components \( R_{D.C} \) are renamed to fit post-path expressions given in the O-view signature.

\[
GET \ R_o \ \text{LEFT JOIN} \ \text{OID}(R_{D.C} \ \text{RENAME} \ a \ \text{as} \ \text{C.a \ldots}) \ \ldots[OID, \ldots, \text{C.a,} \ldots]
\]

Required in-class joins perform before in-reference join operations described further.

*Example.* For example O-view

\[
DOCS(.DocN, .Items.Art)
\]

is calculated as

\[
GET \ (R_{DOCS} \ \text{LEFT JOIN} \ DOCS_items \ \text{RENAME} \ Art \ AS \ Items.Art))[OID, DocN, Items.Art]
\]

### 4.3.3.3 On-Reference JOIN in O-View Attributes

Suppose O-view attribute contains reference \texttt{ref} on object of class \( \text{ref.D} \); then \texttt{LEFT JOIN} allows referencing class relation be joined with referenced class relations on equality of the reference attribute \texttt{ref} of the referencing class relation with OID attribute of the referenced class relations. Also attributes \texttt{a..} of referenced class relations are renamed to fit post-path expressions given in the O-view signature.

\[
GET \ (R_o \ \text{LEFT JOIN}_{\text{ref=OID}} \ (R_{\text{ref.D}} \ \text{RENAME} \ a \ \text{AS} \ \text{ref.a}) \ ) [OID, \text{ref.a..}]
\]

This operation can be nested; reference expressions of any length may be calculated in O-view attributes.

*Example.* Next O-view signature

\[
DOCS(.DocN, .Cntr.Name)
\]

is calculated as
4.3.3.4 On-Reference JOIN in O-View Names

Let us name any expression defining a set of objects as group reference expression. For example the path \( D_{.\text{ref}} \) defines set of objects of class \( _D \) referenced by all existing objects of class \( D \). So, this path is group reference expression. Such group references are unary relations with OID attribute. They are calculated as O-views \( D_{(.\text{ref})} \) with single reference attribute which then are renamed to OID.

\[
\text{GET } R_d[\text{ref}] \text{ RENAME ref AS OID}
\]

Any path ended with reference is group reference expression.

Suppose the name of O-views is group reference expression \( D_{.\text{ref}} \); then the next JOIN is used to restrict output referenced relations \( R_{\text{refD}} \ldots \) with OID given in group reference.

\[
\text{GET } (D_{.\text{ref}} \text{ JOIN } R_{\text{refD}} \ldots ) \text{[OID, ...]}, \text{where } D_{.\text{ref}} \text{ is calculated as described above.}
\]

This operation can be nested too; reference expressions of any length may be calculated in O-view names.

*Example.* Next O-view signature

\[
\text{DOCS}.\text{Cntr}(.\text{Name}, .\text{ID})
\]
is calculated as

\[
\text{GET } (\text{DOCS}.\text{Cntr JOIN R}\text{CONTRACTORS}) \text{[OID, Name, ID]}, \text{where DOCS}.\text{Cntr is calculated as } R_{\text{DOCS}[\text{Cntr}] \text{ RENAME Cntr AS OID.}}
\]

4.3.3.5 Object Selection Expression

Object selection expression is other way to define a set of objects and can be considered as group reference expression too.

\[
\text{name}_0 < \text{condition}_\text{list} >
\]

Its result is unary relation contained a set of OIDs of objects which belong to objects subset defined by \( \text{name}_0 \) of class or reference and satisfy to \( \text{condition}_\text{list} \).

\( \text{Condition}_\text{list} \) can include a number of conditions listed by meant of commas "," which means low-priority operation \( \text{interrows AND} \). Each of the conditions is logical expression available in \( \text{WHERE} \) operation. Its operands can be any post-paths \( \text{cont} \) of name \( \text{name}_0 \).

\[
\text{expr}(\text{cont}_1, \text{cont}_2, \ldots)
\]

A set of object satisfied to this condition is calculated as

\[
\text{GET } (D(\text{cont}_1, \text{cont}_2, \ldots) \text{ WHERE expr(\text{cont}_1, \text{cont}_2, \ldots) }) \text{[OID]}
\]

Here the part \( D(\text{cont}_1, \text{cont}_2, \ldots) \) is implicit O-view which is calculated using operations given above. The result is OIDs of objects satisfying to condition. It's interesting that recording of the condition stays the same (underlined in calculation expression).

\( \text{Interrows AND} \) operation is calculated by relational target machine as \( \text{INTERSEPT} \) of the sets of OIDs which are results of calculating of each of conditions bound by commas. So the next expression

\[
\text{expr}(\text{cont}_1, \ldots), \text{expr}(\text{cont}_2, \ldots)
\]
is calculated as

\[
\text{GET } (D(\text{cont}, \ldots)) \text{ WHERE expr(\text{cont}, \ldots) }) \text{[OID] INTERSEPT (D(\text{cont}, \ldots)) \text{ WHERE expr(\text{cont}, \ldots) }) \text{[OID]}
\]

*Example.* Next O-view signature

\[
\text{DOCS<.Items.Art = "Tie", .Items.Art = "Axe">(.DocN)}
\]
is calculated as
GET ( 
((R_DOCS_item RENAME Art AS Items.Art) WHERE Items.Art = "Tie") [OID]
INTERSECT
((R_DOCS_item RENAME Art AS Items.Art) WHERE Items.Art = "Axe") [OID]
) JOIN
R_DOCS[OID, DocN]

4.4 Implementation Command

General syntax of the implementation command is

ALTER D REALIZE name AS expression

, where D is name of some class, name is a name of its component or method, expression describes how the component or the method is implemented. The command describes an implementation and binds the implementing expression with name of component or method.

In this part translation of the implementing expressions are discussed.

Class components can be stored or calculated. Stored component is implemented with command

ALTER D REALIZE C AS STORED

, where C is the component.

Calculated components are implemented with command

ALTER D REALIZE C AS RValue

, where RValue is any expressions returning value which is not complex than a value of relation.

The methods of class are implemented by procedures using the command

ALTER D REALIZE M(...) AS begin ... end,

, where M(...) is the method signature, begin ... end is implementing procedure.

4.4.1 Group Execution of Operations and Methods

Any operations on objects and object methods do not need iterators to be executed for group of objects. Let's illustrate this ability with simple example. Values of two scalar components a and b are summed in some implementing expression as a + b.

Translation of this operation has to include access operation to values contained in correspondent class relations R... As a result the translation roughly looks like the next expression

GET (R WHERE OID = this)[a] + (R WHERE OID = this)[b]

, where this contains OID of object which the implementation are executed for.

Let us note that the access operation can be "factorized"

GET (R WHERE OID = this)[a+b]

It's obvious that the last expression can be executed for group of objects which is defined by group reference these as

GET (R JOIN these)[OID, a+b]

The result is relation containing all sum values for each of objects referenced by these. Also it contains attribute OID; so it is structurally unified to class relations and can be used in next operations. It's interesting that the recording of the scalar operation stays the same (underlined).

This ability seems to be universal and can be formulated as the next Translation Thesis:

Any procedure p(...) on single object of class D can be translated in such procedure p'(these, ...) on the class relations R... that result of single execution of p' is equal to execution of p for each of objects referenced by any group reference these.

The same statement for single operations on objects can be considered as special case of the Translation Thesis.
(Remark. Further the objects of class D referenced by group reference these are named as these objects. The group reference these is hidden from user. Its value is the result of calculation of some reference expression THESE given in source code.)

The Thesis means e.g. that the expression EXEC THESE.M(...), where M(...) is the method implemented in class D by some procedure p(...), is executed in two next steps:
1. Group reference these is calculated according to reference expression THESE,
2. Translation p' of source p(...) is executed taking these as input parameter.

EXEC p'(these, ...)

At that all data existing in class relation R are changed as if the source procedure p(...) was executed for each of objects defined by source expression THESE.

The Translation Thesis is discussed in next paragraphs. The discussion can be considered as description of principles of translation of operations and procedures.

4.4.1.1 Translation of Expressions Defining Parameters and Local Variables

Let us note that logic of representing component C values in form of class relation R₀... (see 4.2.2) can be also applied to parameters and to local variables of procedures. Scalar parameters sc(par₁, sc(par₂, ... of procedure p can be represented as parameter relation Rpar(OID, sc(par₁, sc(par₂, ...), which is parameter of p' and must be created in code which executes p'. This relation contains scalar parameter values for each of these objects.

GET Rpar [OID] -> sc(these

If procedure p take no parameter, the parameter relation Rpar contains only attribute OID.

The same logic is true for local variables which can be represented in relation Rlocal. This relations must be created inside of p'.

(Remark. For sake of simplicity we consider scalars only but generally the parameters and locals can be values which are not more complex than relation.)

Translation of expressions describing parameters and local variables is similar to translation of class components specification. The result of the translation is temporary real relational variable.

In fact, the parameters and local variables are certainly defined inside procedure as well as other class components are. They can be treated as non-persistent class components with different lifetime. So, when procedures are discussed, they will be considered as elements of set C of class components. Accordingly parameters relation Rpar and local variables relation Rlocal will be considered as elements of set R₀... of class relations further.

4.4.1.2 Translation of RValue expressions.

Let us consider some operation RValue on values C, which describe the state of some class D object. Let us note that the values C are not only values of object component Cᵢ. The name of class components Cᵢ available in scope of some class D can be considered as a short record of expression “this.Cᵢ”, where this is a name of implicit self-reference variable (i.e. special case of group reference expression). With such consideration it is possible to say that in scope of class D a set of local O-views C is defined, whose full signatures includes the name of self-reference variable “this” and (then) the name of class component “Cᵢ” as first two items. In a simplest case for scalar object components Cᵢ the full signature of such local O-view is this(Cᵢ). More complex cases are possible if path “this.Cᵢ” allows post-paths. In these cases expression “this.Cᵢ” appears in the beginning of local O-view signature as a name of the O-view or as a start part of this name.

For each of local O-view C a corresponding O-view Rc exist such as

GET (Rc WHERE OID = this) -> C

Signature of O-view Rc differs from the signature of corresponding local O-view C in first element only; it begins with name of class D instead of name of self-reference variable this.

Let us consider some operation RValue on object views C, which calculates a relational value

f(C₁, C₂, ...) -> Cₙ  \quad (1)
Here $C_n$ is a result of operation, $f$ is composition of primitive relational operation $\text{prim}_i\text{op}_j$ on
object views $C_1$, $C_2$, … available in scope of class $D$
$\text{prim}_i\text{op}_j(C_1, \text{prim}_i\text{op}_j(C_2, \ldots (\ldots)))$

At that, the primitive relational operations $\text{prim}_i\text{op}_j$ can contain any scalar operations on
attributes of operands.

For each of primitive relational operations $\text{prim}_i\text{op}_j$ the next is true.

- Union $C_1 \cup C_2$ is equal to
  $\text{GET} ((R_{C_1} \cup R_{C_2}) \text{ WHERE OID = this})$

- Difference $C_1 - C_2$ is equal to
  $\text{GET} ((R_{C_1} - R_{C_2}) \text{ WHERE OID = this})$

- Cartesian product $C_1 \times C_2$ is equal to
  $\text{GET} ((R_{C_1} \text{ JOIN} R_{C_2} \text{ WHERE OID = this})$

- Selection $C \text{ WHERE condition}$ is equal to
  $\text{GET} ((R_{C \text{ WHERE condition}}) \text{ WHERE OID = this})$

- Projection $C[a_1, a_2, \ldots]$ is equal to
  $\text{GET} ((R_{C[OID, a_1, a_2, \ldots]} \text{ WHERE OID = this})$

At that, all possible in $\text{prim}_i\text{op}_j$ scalar operations on attributes of object views $C$ are applied to
corresponding relations $R$ attributes without changes.

So, all primitive operations $\text{prim}_i\text{op}_j(C_1, \ldots) \rightarrow C_{\text{res}}$ ($C_{\text{res}}$ is result of the operation) is deduced
to expression
$\text{GET} (\text{op}'(R_{C_1}, \ldots)) \text{ WHERE OID = this}) \rightarrow C_{\text{res}}$

(2)

Let's note that logic of representing component $C$ values in form of class relation $R_{C_0}$ (see
4.2.2) can be applied to values $C_{\text{res}}$. Result relation $R_{C_{\text{res}}}$ unites results of $\text{prim}_i\text{op}_j(C_1, \ldots)$
executed for all class $D$ objects.

$\text{GET} (R_{C_{\text{res}}} \text{ WHERE OID = this}) \rightarrow C_{\text{res}}$

(3)

Comparison of (2) and (3) gives that any primitive operation $\text{prim}_i\text{op}_j$ on $C$ can be deduces to
operation $\text{op}'$ on $R_C$ which produces result relation $R_{C_{\text{res}}}$.

$\text{GET} \text{op}'(R_{C_1}, \ldots) \rightarrow R_{C_{\text{res}}}$

Relational algebra closure allows any result relation $R_{\text{res}}$ be used as operand of other
operation
$\text{GET} \text{op}'(... R_{C_{\text{res}}} ...) \rightarrow R_{C_{\text{res}+1}}$

Thus, for any operation (1) such operation $f$ on $R_i$ exists

$\text{GET} f(R_{C_1}, R_{C_2}, \ldots) \rightarrow R_{C_n}$

that its result $R_n$ unites all results $C_i$ of source operation $f$ executed in each of class $D$ objects

$\text{GET} (R_{C_n} \text{ WHERE OID = this}) \rightarrow C_{n}$

Let's name $f$ as translation of source operation $f$. Result of operation

$\text{GET} f(R_{C_1}, R_{C_2}, \ldots) \text{ JOIN} \text{ these} \rightarrow \text{these} R_{C_n}$

is relation $\text{these} R_{C_n}$ which unites result of source operation $f$ executed in each of $\text{these}$ objects.

4.4.1.3 Translation of Procedures

Let us consider a procedure $p$ as algorithmic sequence of assignment operations
\( C_n := f(C_1, C_2, \ldots) \)

and calls of methods

EXEC \( M(\text{par}_1, \text{par}_2, \ldots) \), where \( \text{par}_i \) – parameters of the method.

Assignment operation

\[ C_n := f(C_1, C_2, \ldots) \]

is translated in

\[
\text{SET } R_n := f'(R_1, R_2, \ldots) \text{ JOIN these}
\]

\[
\bigcup \text{these}
\]

WHERE JOIN \( \text{ these } \)

\[
\text{MINUS these}
\]

, where part \( f'(R_1, R_2, \ldots) \text{ JOIN these } \) is result of \( f' \) execution on data of \( \text{these} \) objects, part \( (R_1, \text{WHERE JOIN } \text{ these } \text{ MINUS these}) \) is unchanged part of \( R_n \), containing data of not \( \text{these} \) objects.

This translation updates relation \( R_n \) by means of replacing tuples containing data of \( \text{these} \) objects with new ones. Further it is written as

**REPLACE** \( R_n \) with \( (f'(R_1, R_2, \ldots) \text{ JOIN these}) \)

Call of method

EXEC \( M(\text{par}_1, \text{par}_2, \ldots) \)

is translated in call of its translation

EXEC \( p'(R_{\text{par}}) \)

, where \( R_{\text{par}}(\text{OID}, \text{par}_1, \text{par}_2, \ldots) \) is parameter relation such as \( R_{\text{par}}[\text{OID}] \rightarrow \text{these} \).

Linear sequence of described command \( Sqns \) is translated in the same sequence of their translations \( Sqns'(\text{these}) \). For example the next linear sequence of commands

\[
C_n := f(C_1, C_2, \ldots)
\]

\[
\text{exec } p(\text{par}_1, \text{par}_2, \ldots)
\]

\[
C_{n+1} := f_2(C_1, C_2, \ldots)
\]

is translated in the same sequence of their translations

**REPLACE** \( R_n \) with \( (f'(R_1, R_2, \ldots) \text{ JOIN these}) \)

EXEC \( p'(R_{\text{par}}) \)

**REPLACE** \( R_{n+1} \) with \( (f'(R_1, R_2, \ldots) \text{ JOIN these}) \)

To translate common algorithm structures if... , while... [Bohm and Jacopini, 1966] a number of group reference variables \( \varphi \text{these} \) are used which are local in \( p'(\cdot) \). Each \( \varphi \text{these} \) variable corresponds to one of algorithm branch. Variables \( \varphi \text{these} \) are empty when \( p' \) is starting. Then their values are changed according to the algorithm and to the used condition. For all \( \varphi \text{these} \), used in \( p'(\cdot) \) the next two rules are always true

\[
\bigcup \varphi \text{these} = \varphi \text{these}
\]

\[
\varphi \text{these} \bigcap \varphi \text{these}_i = \emptyset , \ i \neq j
\]

Algorithm structure

If condition then \( Sqns_1 \);

is translated into next sequence of command of relational target machine

\[
\text{SET } \varphi \text{these}_{\text{true}} := \varphi \text{these}<\text{condition'>}
\]

\[
\text{SET } \varphi \text{these} := \varphi \text{these} - \varphi \text{these}_{\text{true}}
\]

if COUNT(\( \varphi \text{these}_{\text{true}} \) then \( Sqns_1'(\varphi \text{these}_{\text{true}}) \)

\[
\text{SET } \varphi \text{these} := \varphi \text{these} \bigcup \varphi \text{these}_{\text{true}}
\]

\[
\text{SET } \varphi \text{these}_{\text{true}} := \emptyset
\]
Here the group reference \texttt{these} corresponds to algorithm branch which precedes the \texttt{if} operator. The group reference \texttt{these\_true} contains identifiers OID of \texttt{these} objects which satisfy to condition given in source code (here an object selection expression is used). \texttt{COUNT (these\_true)} is operation returning number of elements in group reference \texttt{these\_true} (i.e. cardinality of \texttt{these\_true}). As a result a part of algorithm after \texttt{then} operator is executed only for objects which satisfy the given conditions.

Algorithm structure
\begin{verbatim}
while condition  
  Sqns;
\end{verbatim}
is translated as
\begin{verbatim}
SET \texttt{these\_true} := \texttt{these}\<condition'>
SET \texttt{these} := \texttt{these} - \texttt{these\_true}
while COUNT(\texttt{these\_true})>0
begin
  \texttt{Sqns}('\texttt{these\_true}')
  SET \texttt{these} := \texttt{these} UNION (\texttt{these\_true} - \texttt{these\_true}\<condition'>)  
  SET \texttt{these\_true} := \texttt{these\_true}\<condition'>
end
\end{verbatim}

Here the loop will be continuing while objects satisfying to condition exist in \texttt{these\_true} subset.

Let us also note that any algorithm branches can contain statements \texttt{return} which cause execution to leave the current procedure and resume at the point in the code immediately after where the procedure was called. This statement is translated as
\begin{verbatim}
SET \texttt{out\_these} := \texttt{out\_these} UNION \texttt{these}
SET \texttt{these} := \emptyset
\end{verbatim}

This translation does not leave the current procedure but empties current subset \texttt{these} and adds its value to special variables \texttt{out\_these} which collects all the subsets which goes through branches containing \texttt{return} statements. All \texttt{out\_these} objects stay intact while other algorithm branches are executed for other objects.

Thus, any expressions and procedures which implement class components and methods can be translated in operations and procedures of relational target machine, which manipulates the data of group of objects at once.

4.4.2 Translation of STORED Implementation Expression

Stored components are implemented in relational target machine with real relational variables.

\begin{verbatim}
CREATE \texttt{real} \texttt{RD... (OID, ...)}...;
\end{verbatim}

Schemas of the real variables are equal to schemas of corresponding class relations \texttt{RD...}. Real variables which implements the class scalar relations \texttt{RD} (see 4.5.1) exist always.

4.5 Implementation Binding

In this part the \texttt{ALTER \ldots REALIZE\ldots} command are discussed as a command which binds component or method with implementing expression.

Implementations of components and methods can be changed during inheritance. So when component is accessed or method is called it has to be bound with proper implementation. In group operation a number of implementation can be used.

Considering the binding expression translation we will use the next case. Suppose class \texttt{D} was created. Its component \texttt{C} is implemented as stored. Method \texttt{M(\ldots)} is implemented with procedure \texttt{p;}

\begin{verbatim}
CLASS \texttt{D(...C, M(...), ...)};
ALTER \texttt{D REALIZE C AS STORED;}
ALTER \texttt{D REALIZE M(...)} AS \texttt{p;}
\end{verbatim}
Child class \( \text{sub} \text{D} \) is created. In it the inherited component \( C \) is re-implemented as calculated by operation \( f(…) \) and the inherited method \( M \) is re-implemented with procedure \( p_2 \).

\[
\begin{align*}
\text{CLASS } & \text{sub} \text{D EXTEND D \ldots; } \\
\text{ALTER } & \text{sub} \text{D REALIZE C AS } f(\ldots); \\
\text{ALTER } & \text{sub} \text{D REALIZE } M(\ldots) \text{ AS } p_2; 
\end{align*}
\]

### 4.5.1 Component Binding and Class Relation Calculation

Components are bound during class relations calculation. The expression which is used to calculate virtual class relation \( R_{D\ldots} \) is the binding one.

Binding expression for complex components is \( \text{UNION} \) of results given by all implementations. If component \( C \) of class \( D \) is complex ones then corresponded class relation \( R_{D\ldots} \) unites translations of implementation expressions which are set in class \( D \) and in child class \( \text{sub} \text{D} \). The next binding expression is used for the calculation.

\[
\begin{align*}
\text{GREATER } & R_{D\ldots}(\ldots) \text{ AS } \\
& \text{real } R_{D\ldots} \text{ UNION } f_1'(\ldots),
\end{align*}
\]

where \( \text{real } R_{D\ldots} \) is implementation as stored component (as it is set in class \( D \)), \( f_1'(\ldots) \) is translation of realization \( f(\ldots) \) set in child class \( \text{sub} \text{D} \).

Binding expression for simple components is replaced the attribute of stored scalar class relation with calculated value. If component \( C \) of class \( D \) is complex ones then corresponded class relation \( R_{D\ldots} \) is calculated with the next expression

\[
\begin{align*}
\text{GREATER } & R_{D\ldots}(\ldots) \text{ AS } \\
& \text{real } R_{D\ldots} \text{ LEFT JOIN } \text{calcD}(f_1'(\ldots) [OID, calcC]) [OID, \ldots, \text{SUBST}(C, \text{calcC}), \ldots]
\end{align*}
\]

where \( \text{real } R_{D\ldots} \) is stored scalar class relation, \( f_1'(\ldots) \) is translation of realization \( f(\ldots) \) set in child class \( \text{sub} \text{D} \), \( \text{SUBST}(C, \text{calcC}) \) is operator substituting attribute of simple component for calculated value.

It's clear that any changes in component implementation have to result in the changes of the binding expression. Thus translation of source command

\[
\text{ALTER } D \text{ REALIZE } C \text{ AS } \ldots
\]

has to re-create the binding expression executed by target machine to calculate class relations.

### 4.5.2 Method Binding

A number of implementing procedures can be executed if method is called for group of \textit{these} objects. In this case binding expression is a procedure which calls all the method implementations. At that for each of the implementations the corresponding subset of \textit{these} objects has to be determined.

Method call for group of objects

\[
\text{exec } \text{THES}E.M(\ldots)
\]

means the execution of the next stored transaction

\[
\begin{align*}
\text{TRANS } & D.M'(\text{these}, \ldots) \text{ AS } \\
\text{begin } & \\
\text{EXEC } & p_1'(\text{these INTERSEPT scope}(p_1), \ldots); \\
\text{EXEC } & p_2'(\text{these INTERSEPT scope}(p_2), \ldots); \\
\text{end },
\end{align*}
\]

where \( p_1' \) and \( p_2' \) are translation of implementing procedures \( p_1 \) and \( p_2 \), \textit{scope(p)} is a group reference on objects which the implementation \( p \) is bound with.

It's clear that any changes in method implementation have to result in the changes of the binding expression. Thus translation of source command

\[
\text{ALTER } D \text{ REALIZE } M(\ldots) \text{ AS } \ldots
\]

has to re-create the binding procedure \( D.M'(\text{these}, \ldots) \) which is ran in target machine to execute proper implementations of the method.
4.6 Object Creation Command

Object is created using the instruction NEW.

NEW D constructing_expression

Let us note that it can be used both as in-procedure instruction and as a declarative language command. In the last case it doesn't return reference on new objects. Any objects are element of classes and can be accessed using data access commands by values of its components which could be set by constructing_expression (any command sequence or procedure defined in context of class D).

4.6.1 Translation of Object Creation Command

During object creation the relational target machine performs the next steps
1. New unique OID values is generated
2. Tuples with new OIDs is added into the stored class scalar relation \( R_D \). For object of child classes all such relations of parent classes have to be added with the tuples too.
3. Translated \( \text{construction}_{\text{expression}}' \) is performed using new OID values as these group reference

5. COMPATIBILITY OF CLASSES AND RELATIONS

Because of using the relational data model as format basis and the relational programmable system as target machine the offered approach doesn't impede collateral usage of both classes and relations which are native data structure for the target machine. There are a three ways of how classes and relations can be co-used. Let us consider class D and relation T.

1) Relations can be defined on a set of domains which is extended with new reference types. Suppose relation T contains attribute \( \text{ref} \) of reference domain D.

CLASS D(C;...);

CREATE T(i:INTEGER, ref:D);

Then any of the class D post-paths can be used when the relation T is accessed.

\[ T(i, \text{ref}.C) \]

The last relation is calculated in relational target machine by meant of JOIN on relation attributes \( \text{ref} \). Also attributes are renamed to fit path expressions given as attribute name.

GET (T LEFT JOIN \( \text{ref} = \text{OID}(R_D \text{ RENAME C AS ref}.C) \)) [i, \text{ref}.C]

2) Mutual constraints (foreign key) can be set for any class D and any relation T. Such constraints implemented in relational target machine as the ones set on the relation T and corresponded class relation \( R_D \).

3) Because data of classes presented in form of relations (object views) they can be combined with the explicit relations in data access operations (e.g. in ad-hoc queries).

6. CONCLUSION

We introduced base principles of object-oriented translation for programmable relational system. Offered approach allows formal-based evolution of existing relational systems towards the systems which can be described as independent object-oriented environments for active persistent manageable model of problem domain. At that possibilities and features of existing relational systems (e.g. transaction, access protocols etc.) stay the same so current population of applications used the systems may stay intact.

In conclusion let us recall the idea of data independence defined by E.F.Codd as "the independence of application program and terminal activities from growth in data types and changes in data representation". The term "data" can be treated very widely. It means both values describing the state of objects of problem domain and functionality which reflects the objects mutability and their dependences. All the data form a model of problem domain and in this case the data independence means independence of the model.
ACKNOWLEDGEMENT

The author thanks Sergey D. Kuznetsov for numerous works which was fertile ground for offered suggestions.

REFERENCES

CODD, E. F. 1970. A relational model for large shared data banks. *Communications of the ACM* 13(6): 377–387.

BOOCH, G., 1991. *Object-oriented Analysis and Design with Applications*. Benjamin-Cummings Publishing.

DATE, C. J. AND DARWEN, H. 1998. Foundation for Object/Relational Databases – The Third Manifesto. Addison Wesley.

PRATT, T. W. AND ZELKOWITZ, M. V. 2001. *Programming Languages: Design and Implementation*, London: Prentice Hall, Inc.

STONEBRAKER, M., ROWE, L. A., LINDSAY, B. G., GRAY J., CAREY, M. J., BRODIE, M. L., BERNSTEIN, PH. A. AND BEECH, D. 1990. Third Generation Database System Manifesto. *ACM SIGMOD Record* 19, No. 3, 31–45.

CATTELL, R. G. G., BARRY D. K., BERLER M., EASTMAN J., JORDAN D., RUSSELL C., SCHADOW O., STANIENDA T., AND VELEZ F. 2000. The Object Database Standard: ODMG 3.0, Morgan Kaufman Publishers.

ISO/IEC 9075:2008. 2008. Information Technology, Database Languages, SQL. Part 2: Foundations.

BOHM, C. AND JACOPINILG. 1966. Flow diagrams, turing machines and languages with only two formation rules, *Communications of the ACM*, v.9 n.5, p.366-371

ABITEBOUL, S. AND BIDOIT, N. 1986. Non first normal form relations: An algebra allowing data restructuring, *Journal of Computer and System Sciences*, v.33 n.3, p.361–393.

BADIA, A. 2003. Automatic generation of XML from relations: The nested relation approach. In *Proceedings of the ER Workshop*, 330–341.

GRIGORIEV, E. 2011. RxO system. Simple semantic approach for representation complex objects data in table form, [http://odbms.org/download/RxO1.pdf](http://odbms.org/download/RxO1.pdf)

GRIGORIEV, E. 2012. Impedance mismatch is not "Objects vs. Relations" problem, [http://odbms.org/download/EvgeniyGrigoriev.pdf](http://odbms.org/download/EvgeniyGrigoriev.pdf)