ON THE MIGRATION OF RELATIONAL SCHEMAS AND DATA TO OBJECT-ORIENTED DATABASE SYSTEMS[1]

Andreas Behm, Andreas Geppert, Klaus R. Dittrich[2]

Abstract

Many organizations have collected large volumes of data which are often stored in relational databases. Now that object-oriented database management systems (OODBMSs) are becoming mature, many organizations also want to benefit from object-oriented technology in general and OODBMSs in particular. One general approach to migrate to object technology is to divide the process into two phases, where the first one transforms the relational into an object-oriented schema and the second one migrates the data into the object-oriented database system. Our approach differs from comparable approaches because it provides assistance in both phases, while other work considers only the first phase. In the first phase, we use transformation rules to construct an object-oriented schema which is semantically equivalent to the relational schema. In the second phase, the schema transformation information is used to generate programs that migrate the relational data into an object-oriented database. The concepts of both, schema transformation and data migration are implemented using O2 as the OODBMS.

1 Introduction

Many organizations have collected large volumes of data which are often stored in relational databases. Relational database systems (RDBSs) [EN94] consist of a relational DBMS (RDBMS), a relational schema and a database. A database is structured into relations whose elements (tuples) are in turn sets of atomic attributes. Tuples can reference each other by foreign keys. Hierarchies and relationships are implemented by joins. Applications, usually written in imperative programming languages like C or Cobol, can access a database using the (embedded) declarative language SQL.

The two major drawbacks of RDBSs are that complex structures cannot be modelled directly and the *impedance mismatch* between the type systems of (object-oriented and imperative) programming languages and database schemas. As a response to these shortcomings, object-oriented database systems (OODBSs) have been proposed [ABD+89]. In these systems, data is stored in the form of objects which are identified by system-maintained and usually invisible object identifiers (oids). Objects reference each other by references (i.e., pointers). The specialization/generalization relationship can be modelled directly by inheritance, and aggregation can be expressed through complex relationships. Moreover, in contrast to RDBMSs, class-specific behaviour can be modelled in OODBSs. This short summary already shows that the relational and

In Proc. of the 5th International Conference on
Re-Technologies in Information Systems,
Klagenfurt, Austria, December 1997

1. This research was supported by BBW, the Federal Office of Education and Science in Switzerland through ESPRIT III project SYNDAMMA, no. 94.0118.

2. Database Technology Research Group, Department of Computer Science, University of Zurich, Winterthurerstr. 190, CH-8057 Zurich, Switzerland, {abehm|geppert|dittrich}@ifi.unizh.ch

the object-oriented paradigm are too different to migrate from one to the other automatically. The motivation to introduce OODBMS is the impedance mismatch between the object-oriented data model and the relational schema.

Now that OODBMSs are becoming mature, many organizations using object-oriented technology also want to benefit from OODBMSs. In other words, these organizations want to *migrate* from relational to object-oriented databases and database applications. Migrating to object technology [Gra95] consists of reverse engineering of the application programs and migration of the database. The goal of database migration is to have an equivalent schema and database available under the new OODBMS. This can of course be achieved through manually coded transformation programs, which however is a complex and error prone task. Remain three kinds of approaches how to make use of object technology for relational databases:

- building an object-oriented interface on top of a relational database system,
- migration to object-relational database systems [Sto95],
- converting the relational schema and database into an object-oriented one.

The first approach retains the relational database and creates an object-oriented interface on top of it. This approach is the easiest one (and existing class libraries exist for this purpose); there is no system interruption for data migration, and no semantic information is lost. On the other hand, performance decreases due to the "impedance mismatch" between the two paradigms at runtime. In the second approach data has to be migrated in responsibility of the DBMS (e.g. Oracle version 7 to 8), and object-oriented features can be applied only when extending or modifying the schema. The third approach is database migration where a new schema under a new OODBMS is created and the data is migrated from the relational database to the object-oriented one.

Each of the three possibilities has advantages and disadvantages. Especially in the third one, an underlying method and tool support are critical for successful migration. In this paper, we present such a method how to create an object-oriented schema incrementally from a relational one. In contrast to the other approaches, migration to an OODBMS also requires migration of the database. Particularly, the process of schema transformation should determine how to migrate the database, i.e. the instance migration program is generated based on the schema transformation. In this paper we present our approach towards a systematic migration which is intended to meet the following requirements:

- to provide for methodical support of the migration process,
- to assist users in transforming relational schemas and databases,
- to allow users to extend the method for schema transformation,
- to exploit the full set of OODBMS-features,
- to cover properties of the migration itself such as correctness and completeness.

Note that we consider the database schema and instances, but not application programs. Migrating these programs not only means to convert DBMS-calls in programs, but in addition requires the re-engineering of imperative programs into object-oriented ones and the re-construction of user interfaces. Thus, to a large extent re-engineering of database application programs is a special case of software re-engineering and will be possible once this task will be better understood and supported by tools.

The paper is organized as follows: In Chapter 2 related approaches are discussed. An overview of our migration process is presented in Chapter 3. Chapter 4 contains the concept for schema transformation and data migration. It explains the formal notation with which the transformation rules are defined. Tool support for the migration process is sketched in Chapter 5. The implementation of the prototype is described in Chapter 6.

2 Related Work

Several approaches for mapping relational to object-oriented schemas have been proposed so far. Most of them present a formalism with which the relational schema is transformed more or less automatically to an object-oriented one. A brute-force solution (pursued, e.g., in Pegasus [AAK+93]) is to simply map each relation definition to a class definition and to consider the relation instance as the class extension. This approach can be automated and gateways from some OODBMS (e.g. Object Store, O2) to relational DBMSs already exist. The drawback of this solution is that the object-oriented database system is used in a "relational" manner and that many of the object-oriented features are not exploited. Other approaches [CS91, YL93, And94, PB94, BMP94, FV95, Joh94, VA95b] consider object-oriented concepts like references or inheritance, but due to a non-interactive schema transformation, the resulting object-oriented schema still "looks rather relational". A design environment for interactive schema transformation is sketched in [JSZ96]. In general, existing approaches do not (yet) regard data migration.

Schema transformation needs information about the semantics of the relational schema, such as

- a classification of relations and attributes,
- inclusion and foreign key dependencies, or
- possible candidate keys [EN94].

Often, this information is not directly available from the relational database system (i.e. catalogue). In this case *semantic enrichment* has to be performed. Semantic enrichment can be seen as a special area of *reverse engineering* [DA87, CBS94]. The reverse engineering process cannot be completely automated. User interaction is necessary when ambiguities occur and domain semantics cannot be inferred. Reverse engineering in general may start with analysing DDL code, queries or applications.

In [CBS94] a methodology for extracting an extended entity-relationship (EER) model from a relational database is presented. They analyse not only the data schema, but also data instances which contain detailed information about the application domain. The relational schema is extensively classified into different kinds of relations, attributes, and inclusion dependencies. Based on this information, the entity and relationship types can be identified. Hainaut et al. [HCTJ93, HTJC93, HEH+95] present an extensive methodology for database reverse engineering. Few approaches regard also queries in the relational schema to extract additional semantic of the relational schema [VA95a, JSZ97]. Reverse engineering in [PTBK96] starts with analyzing the equi-join queries embedded in application programs and by querying the database.

Up to now there is no approach which considers a complete migration process, i.e. semantic enrichment, schema transformation and data migration. Some approaches do not even take advantage of all object-oriented features like the Pegasus project [AAK+93].

3 The Migration Process

In this chapter we introduce our notion of the migration process in general. The starting point of database migration is a relational schema and database in a relational DBMS. The result is an object-oriented schema and database in an object-oriented DBMS.

Our approach assumes precise information about relation and attribute definitions, inclusion dependencies and relation "types" (i.e. weak or strong). Sometimes, this information can be obtained from the catalogue or the DDL-code. In many cases, however, it is not available from either source and thus must be specified by the user. The phase of gathering additional information about the relational schema is called semantic enrichment, and forms the first phase of the migration process. Semantic enrichment cannot be done automatically and the semantic information cannot be checked for correctness. Data analysis can be performed in order to check whether the current database would satisfy the semantic enrichment. Input for the second step is thus a relational schema enriched with correct semantic information.

The second and third step are *schema transformation* and *data migration*. To implement schema transformation we introduce the concept of *transformation rules*. A transformation rule transforms a part of the relational schema, in the following called *pattern*, to a part of the object-oriented schema. For example, a foreign key dependency can be transformed to a *part-of* relationship, or a relation can be transformed to a class.

```
repeat
        for each transformation rule determine applicable pattern
        select transformation rule/pattern
        execute transformation rule/pattern
until no more rule applicable
```

Figure 1: Process of schema transformation

Schema transformation is an iterative process as shown in Fig. 1 where a user incrementally creates the object-oriented schema by applying a predefined set of transformation rules. In each iteration step, the user can choose among possibly multiple applicable transformation rules. Schema transformation rules are subdivided into *class creation rules* and *attribute creation rules* The result of applying class creation rules is a class structure consisting of classes and the inheritance hierarchy. Attribute creation rules determine the structure of these classes. The correlation between schema transformation and data migration is shown in Fig. 2.

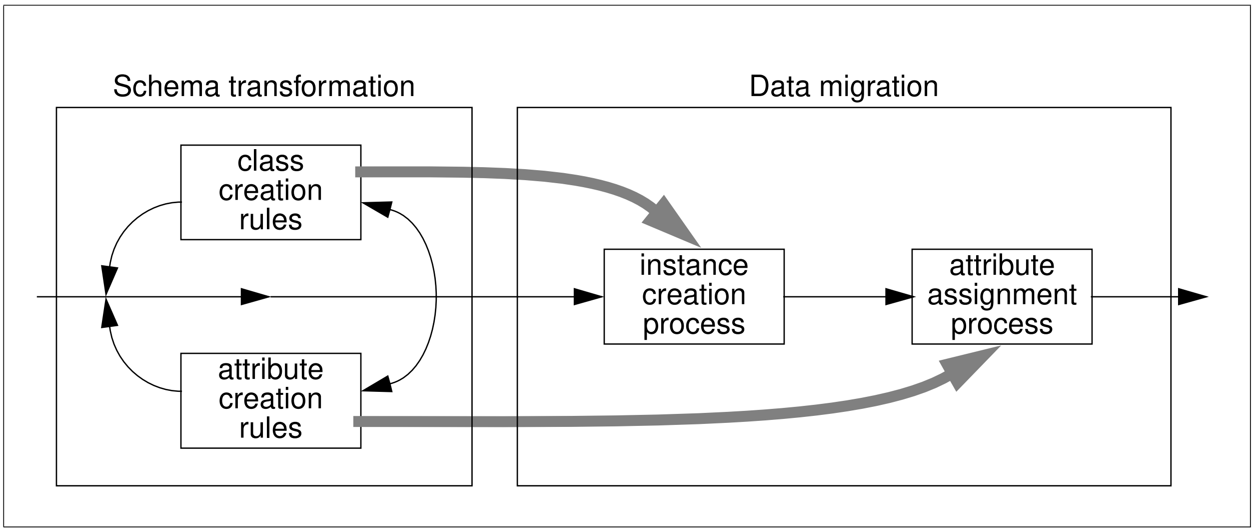

Figure 2: Schema transformation and data migration

Data migration is divided into two separate processes. In the first step, called *instance creation process*, the instances are created, and in the second step, called *attribute assignment process*, values are assigned to the attributes of the instances. The values of some attributes may contain object references and this separation ensures that these object references exist when the attributes values are computed.

In the following we give an example which we will later use to demonstrate all the class and attribute creation rules. The ER-schema in Fig. 3 and the relational schema in Fig. 4 model a library of

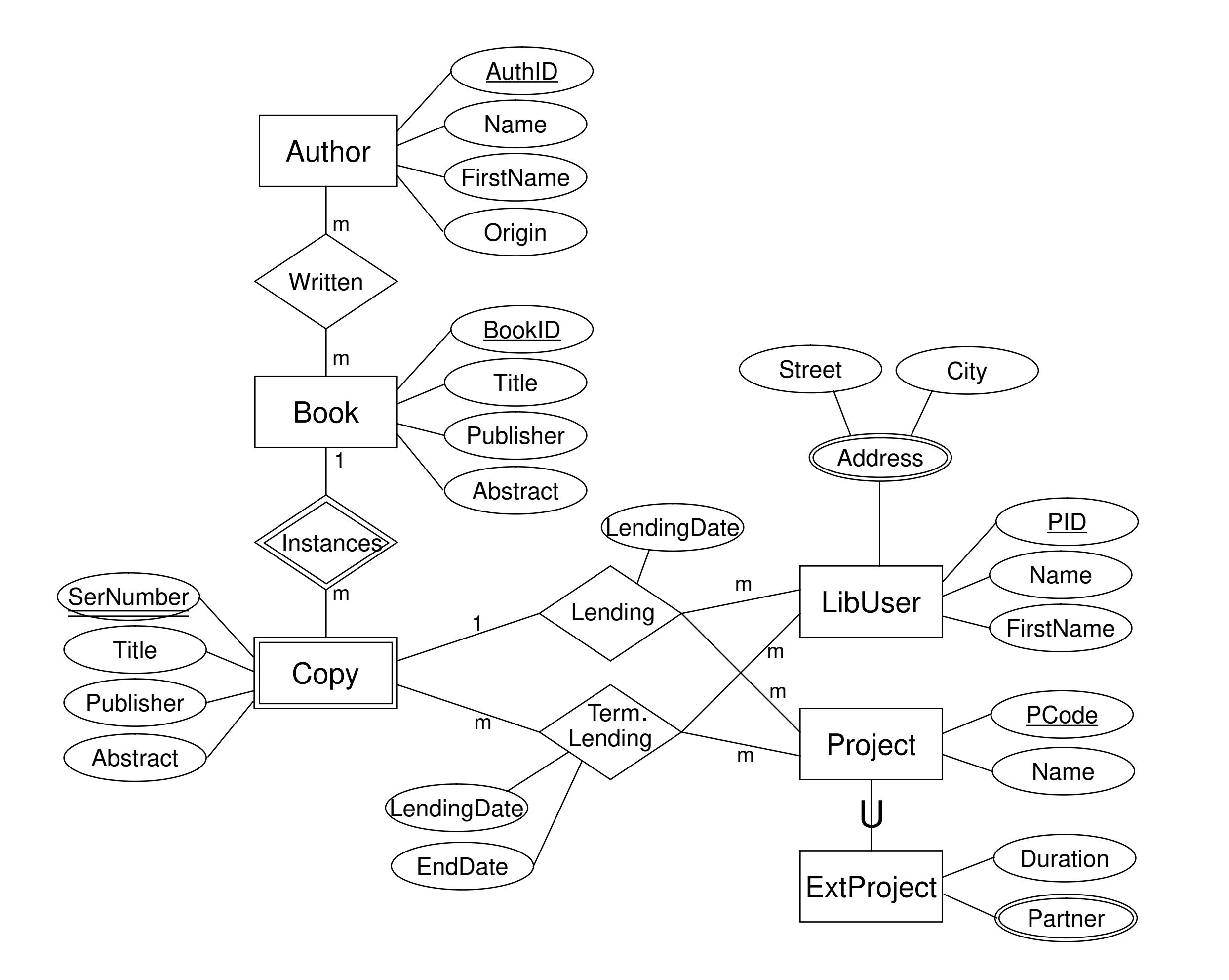

Figure 3: EER schema of a library

books. A book can be written by several authors who in turn can have written several books. Multiple copies of a specific book can be available in the library. Furthermore, the relation `LibUser` stores the users of the library and projects for which copies can be lent. External projects differ from common projects in having several external partners. The copies which are actually lent are registered in `Lending` whereby a copy must be lent by a certain library user for a certain project. After a book is returned, the corresponding tuple in the relation `Lending` is moved to `TerminatedLending`.

4 Schema Transformation and Data Migration

In this chapter we present the four subparts of schema transformation and data migration, namely class creation rules, attribute creation rules, instance creation process and attribute assignment process. As these parts rely on the features of relational and object-oriented schemas, we first introduce a formal notation with which the schemas and the relationship between a relational and an object-oriented schema can be described. Then we investigate the operations with which the object-oriented schema can be created step by step. With this formalism, it is possible to express a first set of trans-

Author (AuthID, Name, FirstName, Origin)
contains the authors of the books in the library
Book (BookID, Title, Publisher, Abstract)
contains the books of which copies are stored in the library
Written (BookID, AuthID)
defines a many-to-many relationship between books and authors
Copy (BookID, SerNumber, DateAcq, NrVolumes, State)
contains instances of books which are physically stored in the library
LibUser(PID, Name, FirstName)
contains people who can lend books from the library
Address(PID, Street, City)
contains addresses of the library users
Project (PCode, Name)
contains projects for which books can be lent
ExtProject (PCode, Duration)
specialization of projects with external partner
Partner (PCode, Partner)
contains partner of external projects
Lending(BookID, SerNumber, LendingDate, PCode, PID)
contains copies which are actually lent by a certain library user for a certain project
TerminatedLending (BookID, SerNumber, LendingDate, EndDate, PCode, PID)
contains the history when copies have been lent out

Figure 4: The relational schema of the library example

formation rules and we will demonstrate them in an example. Finally, we consider data migration and propose two algorithms for instance creation and attribute assignment.

4.1 Formal Notation

The foundation of schema transformation and data migration is a formal model which includes the description of the relational schema, the semantic enrichment, the object-oriented schema and the relationship between the relational and object-oriented schema.

4.1.1 Relational Schemas

The definition of a relational schema contains elements like relations, attributes, keys of the relations and the types of the attributes. To express the result of the semantic enrichment we add a set of inclusion dependencies [EN94] to the schema. So far, we only consider functional dependencies limited to those induced by keys, in other words, we expect the relational schema to be in third normal form [EN94]. The formal model of relational schemas comprises:

- A finite set of relations $R = \{r_1, r_2, r_3, \ldots\}$
- A finite set of attributes $A_R = \{a_1, a_2, a_3, \ldots\}$
- A function *att*: $R \rightarrow 2^{A_R}$, $att(r_i) = \{a_{i_1}, a_{i_2}, a_{i_3}, \ldots\}$
- A function *key* which determines the key of each relation: $key(r_i) = \{a_{i_{k_1}}, a_{i_{k_2}}, a_{i_{k_3}}, \ldots\}$ with the restriction: $key(r_i) \subseteq att(r_i)$
- A set T_R of the basic, built-in types in relational databases.
- A function *type* from A_R to T_R which gives the type of each attribute.

- A set of inclusion dependencies I where each element has the form $((r_1,A_1),(r_2,A_2))$ where r_1 and r_2 are relations; A_1 and A_2 are subsets of $att(r_1)$ and $att(r_2)$ such that $A_1 = \{a_{1_1}, a_{1_2}, a_{1_3}, \ldots\}$ and $A_2 = \{a_{2_1}, a_{2_2}, a_{2_3}, \ldots\}$ with the following restrictions: $|A_1| = |A_2|$ and $type(a_{1_i}) = type(a_{2_i})$
- A set I_C, which is the transitive closure of I.
  The set of inclusion dependencies of the library example is represented in Tab. 1.

I	r1	A1	r2	A2
1	Written	BookID	Book	BookID
2	Written	AuthID	Author	AuthID
3	Copy	BookID	Book	BookID
4	ExtProject	PCode	Project	PCode
5	Lending	BookID, SerNumber	Copy	BookID, SerNumber
6	Lending	PCode	Project	PCode
7	Lending	PID	LibUser	PID
8	TerminatedLend.	BookID, SerNumber	Copy	BookID, SerNumber
9	TerminatedLend.	PCode	Project	PCode
10	TerminatedLend.	PID	LibUser	PID
11	Address	PID	LibUser	PID
12	Partner	PCode	ExtProject	PCode

Table 1: Inclusion dependencies in the library schema

4.1.2 Object-Oriented Schemas

An object-oriented schema consists of a set of classes, an inheritance hierarchy, a set of attributes of each class and the types of the attributes. The formal model of an object-oriented schema comprises:

- A set of classes $C = \{c_1, c_2, c_3, \ldots\}$
- A set of attributes $A_O = \{a_1, a_2, a_3, \ldots\}$
- A function *att*: $C \to 2^{A_O}$, $att(c_i) = \{a_{i_1}, a_{i_2}, a_{i_3}, \ldots\}$
- A set T_O of types which includes basic types as well as classes and set, list or tuple constructs which can be described as follows:
  type = **simple_type** | **composed_type**
  simple_type = **basic_type** | classname
  basic_type = int | char | ...
  composed_type = set(**type**) | list(**type**) | tuple(a_1: **type**, ... a_n: **type**)
- A function *type* from A_O to T_O which gives the type of each attribute
- A set H describing the class hierarchy, where each element has the form (c_1, c_2), denoting that c_1 inherits from c_2.

4.1.3 Relationship between the relational and object-oriented schema

Finally we formalize how the relational schema is transformed into the object-oriented schema:

- A total function *rel* from C to R which denotes the relation to which a class is associated. At the moment, we assume that each class is derived from exactly one relation.
- A function *tt* from T_R to T_O which transforms relational into object-oriented types. For example, the relational type `VARCHAR` would be mapped into the type `String` in O_2.

4.2 Schema Transformation

In this chapter we introduce the class and attribute creation transformation rules. Before we present these rules, we introduce three basic operations with which the object-oriented schema is incrementally created.

- add_class(r, c): adds the class c to *C* so that *rel(c)=r*
- add_attribute(c, a, t): adds an attribute a of type t to c so that $a \in att(c)$ and *type(a)=t*
- add_inheritance(c_1, c_2): denotes an inheritance relationship so that $(c_1, c_2) \in \mathbf{H}$

4.2.1 Transformation Rules

Each class or attribute creation rule specifies the structure of a transformation pattern, a set of preconditions **P** and a sequence of basic operations **Q** which are executed when the rule is applied. The pattern consists of one or more parameterized elements of the relational schema, i.e. relations, attributes or inclusion dependencies. The preconditions are predicates over elements of the pattern. All preconditions have to hold when a rule is applied. The parameters with which the basic operations are invoked are either elements of the pattern or new values. All the basic operations of **Q** are executed when applying a rule. A certain pattern can only be transformed once. To ensure that a pattern cannot be transformed again, it is marked after the rule is applied. In the following, class and attribute creation rules are introduced with an informal description, the structure of the pattern and formal lists of preconditions **P** and operations **Q**. Each rule is followed by an example which consists of a pattern of the library schema and the code which represents the elements of the object-oriented schema created by the operations **Q**.

4.2.1.1 Class Creation Rules

Class creation rules are introduced to create the class structure of the object-oriented schema. This comprises the declaration of the classes (without attributes) and of the inheritance hierarchy. Since a class is derived from exactly one relation, there is only one rule for class creation. Extracting inheritance relationships from a relational schema is a non-trivial task because it requires behavioural information as well. So far, our model does not support a formalization of behavioural information and we present only one rule for the creation of inheritance relationships.

Rule 1: **Class creation from relation**

Creation of a class c_{new} derived from a relation r. This is the rule to create the classes themselves as mentioned above. There is only one precondition necessary where the relation is defined from which the class is derived.

Pattern	Preconditions	Operations
$r \in R$	$\neg\exists c \in C$: r = rel(c)	add_class(r, c_{new})

Example:

Pattern: r = Author

Result: `class Author;`

Rule 2: **Class inheritance by key inclusion dependency:**

Inheritance relationships can be created when an inclusion dependency between the keys of two relations exists. If for each tuple in a relation r_1 a tuple with the same key in another relation r_2 exists, this can be seen as a specialization because to a key in a tuple of r_1 belong attributes of both r_1 and r_2.

Pattern	Preconditions	Operations
id $((r_1,A_1),(r_2,A_2)) \in I$	$\exists c_1 \in C$: $r_1 = rel(c_1)$ $\exists c_2 \in C$: $r_2 = rel(c_2)$ $A_1 = key(r_1)$ $A_2 = key(r_2)$ $((r_2, A_2), (r_1, A_1)) \notin I_C$	add_inheritance (c_1, c_2)

Example:

Pattern: id = ((ExtProject, {PCode}), (Project, {PCode}))

Result:
```
class ExtProject inherit Project;
```

4.2.1.2 Attribute Creation Rules

Attribute creation rules are defined the same way as class creation rules. It is not necessary to complete the class structure before creating attributes.

Rule 3: **Attribute creation from relational attribute:**

Add an attribute to a class c from the associated relation rel(c). This is the general rule which can always be used for any attribute of any relation from which a class has been derived with rule 1. Consequently this rule will be applied for attributes which can not be transformed with one of the other attribute crreation rules, i.e. for attributes which do not appear in any set A_1 in the inclusion dependencies I.

Pattern	Preconditions	Operations
$r \in R$ $a_r \in A_R$	$\exists c \in C$: $r = rel(c)$ $a_r \in att(r)$	add_attribute(c, a_o, tt(type(a_r)))

Example:

Pattern: r = Author
a_r = AuthID

Result:
```
class Author
   AuthID: integer;
```

Rule 4: **Foreign key relationship:**

Transform a foreign key dependency into object references. Foreign key constraint may represent a one-to-many relationship between two relations, so the result in the object-oriented schema is a one-to-many relationship implemented by object references. A relation r_1 has a foreign key A_1 referencing the key A_2 of the relation r_2. The class c_1 derived from r_1 gets an attribute a_{c2} of type c_2, the class c_2 derived from r_2 gets an attribute a_{c1} of type set(c_1).

Pattern	Preconditions	Operations
id $((r_1,A_1),(r_2,A_2)) \in I$	$\exists c_1 \in C: r_1 = rel(c_1)$ $\exists c_2 \in C: r_2 = rel(c_2)$ $A_1 \subseteq att(r_1)$ $A_1 \cap key(r_1) = \{\}$ $A_2 = key(r_2)$	add_attribute(c_1, a_{c2}, c_2) add_attribute(c_2, a_{c1}, set(c_1))

Example:

Pattern: id = ((Lending, {PCode}), (Project, {PCode}))

Result:

```
class Lending
   project: Project;

class Project
   lending_set: set(Lending);
```

Rule 5: **Many-to-many relationship:**

Transform a relation whose attributes are a concatenation of primary keys of two other relations to many-to-many relationships. Such a relation may represent a binary many-to-many relationship. Consider a relation r_1 with only key attributes where one subset A_1 references a relation r_2 and the other subset A_3 references r_4. Then the classes c_2 and c_4 derived from r_2 and r_4 get set-valued attributes whose elements reference instances of the other class.

Relations representing relationships of higher degrees or binary relationships with attributes must be transformed into further classes.

Pattern	Preconditions	Operations
id1 $((r_1,A_1),(r_2,A_2)) \in I$ id2 $((r_3,A_3),(r_4,A_4)) \in I$	$\exists c_2 \in C: r_2 = rel(c_2)$ $\exists c_4 \in C: r_4 = rel(c_4)$ $\neg\exists c_1 \in C: r_1 = rel(c_1)$ $r_1 = r_3$ $key(r_1) = att(r_1)$ $A_1 \cap A_3 = \{\}$ $A_1 \cup A_3 = att(r_1)$ $A_2 = key(r_2)$ $A_4 = key(r_4)$	add_attribute(c_2, a_{c4}, set(c_4)) add_attribute(c_4, a_{c2}, set(c_2))

Example:

Pattern: id1 = ((Written, {BookID}), (Book, {BookID}))
id2 = ((Written, {AuthID}), (Author, {AuthID}))

Result:

```
class Book
   author_set: set(Author);

class Author
   book_set: set(Book);
```

Rule 6: **One-to-one relationship:**

An inclusion dependency between the keys of two relations in both directions leads to a one-to-one relationship which is implemented by adding single object references to both derived classes. The preconditions of this rule are a subset from those of rule 2. Therefore, this case requires user interaction for rule application.

Pattern	Preconditions	Operations
id $((r_1,A_1),(r_2,A_2)) \in I$	$\exists c_1 \in C: r_1 = rel(c_1)$ $\exists c_2 \in C: r_2 = rel(c_2)$ $A_1 = key(r_1)$ $A_2 = key(r_2)$	add_attribute (c_1, a_{c2}, c_2) add_attribute (c_2, a_{c1}, c_1)

Example:

Pattern: id = ((Lending, {BookID, SerNumber}), (Copy, {BookID, SerNumber}))

Result:

```
class Lending
   copy: Copy;

class Copy
   lending: Lending;
```

Rule 7: **Weak entity relationship:**

Create a one-to-many relationship from a relation where an inclusion dependency exists between a proper subset of the key attributes and the key of another relation.

Pattern	Preconditions	Operations
id $((r_1,A_1),(r_2,A_2)) \in I$	$\exists c_1 \in C: r_1 = rel(c_1)$ $\exists c_2 \in C: r_2 = rel(c_2)$ $A_1 \subset key(r_1)$ $A_2 = key(r_2)$	add_attribute (c_1, a_{c2}, c_2) add_attribute $(c_2, a_{c1}, set(c_1))$

Example:

Pattern: id = ((Copy, {BookID}), (Book, {BookID}))

Result:

```
class Copy
   book: Book;

class Book
   copy_set: set(Copy);
```

Rule 8: **Multivalued attribute:**

Transform a multivalued attribute with an atomic type into a set attribute of the corresponding class.

Pattern	Preconditions	Actions
id $((r_1,A_1),(r_2,A_2)) \in I$	$\neg\exists c_1 \in C: r_1 = rel(c_1)$ $\exists c_2 \in C: r_2 = rel(c_2)$ $att(r_1) = key(r1)$ $A_1 \subset att(r_1)$ $\lvert att(r_1)\rvert = \lvert A_1\rvert + 1$ $A_2 = key(r_2)$	add_attribute $(c_2, a_{r1},$ $set(tt(att(r_1) \setminus A_1)))$

Example:

Pattern: id = ((Partner, {PCode}), (ExtProject, {PCode}))

Result:
```
class ExtProject
    partner: set(string);
```

Rule 9: **Composite multivalued attribute:**

Transform a composite multivalued attribute representing a tuple of atomic values into a set attribute of tuples containing the composite attributes.

Pattern	Preconditions	Operations
id $((r_1,A_1),(r_2,A_2)) \in I$	$\neg\exists c_1 \in C$: $r_1 = rel(c_1)$ $\exists c_2 \in C$: $r_2 = rel(c_2)$ $att(r_1) = key(r1)$ $A_1 \subset att(r_1)$ $A_2 = key(r_2)$	add_attribute (c_2, a_{r1}, set(tuple(att(r_1) - A_1)))

Example:

Pattern: id = ((Address, {PID}), (LibUser, {PID}))

Result:
```
class LibUser
    address_set: set(tuple(street: string,
                           city:   string));
```

These nine transformation rules allow a transformation to an object-oriented schema which exploits object-oriented features. However it is possible that a user wants to define additional rules. Since all rules are defined in the same structure, new rules can be added by defining the pattern structure, a set of preconditions and a sequence of operations.

4.3 Data Migration

Once the object-oriented schema is created, the process of *data migration* can start. The objective of this task is the population of the object-oriented database based on the tuples of the relational database. In contrast to schema transformation, this task will be executed automatically so that no interaction with the user is necessary. As mentioned above, data migration will be executed in two separate steps. In the first step, all the required instances of the classes are created without considering values of attributes. These are assigned in the second step.

4.3.1 Instance Creation

In this section we present an algorithm for creating the instances of the object-oriented database. Before creating objects we have to know how to access instances. To simplify this problem we assume the presence of an extension for each class where all instances of this class and all subclasses are stored persistently.

```
foreach class c ∈ C traversing inheritance hierarchy bottom up do
    foreach t ∈ rel(c) do
         if ∀(et ∈ E_c): et.key ≠ t.key
              E_c += (new(c), t.key)
         end
    end
    foreach (c,ci) ∈ H do
         foreach et ∈ E_c do
              if ∀(eit ∈ E_ci): eit.key ≠ et.key
                   E_c += et
              end
         end
    end
end
```

Figure 5: Algorithm for instance creation

During the migration process objects are created and foreign key dependencies are transformed into object references. Therefore objects are stored together with the key value of the related tuple in the relational database, i.e. the extensions contain pairs of the form (object, key value).

4.3.2 Attribute Assignment

After the creation of all instances in the object-oriented database, we can start computing the values of the instance attributes. The algorithms presented in this section are independent of the attribute creation rules, i.e. changes or extensions of the attribute creation rules do not influence the attribute assignment process. The way how values are assigned to the attributes is determined by the executed patterns in the schema transformation process. Each pattern generates the code which computes the values of the attributes created by its transformation rule. Since the value of an attribute has to be computed from both, the relational and the object-oriented database, a pattern generates two queries per attribute, one accessing the relational and one accessing the object-oriented database, i.e. one SQL and one OQL query. The process of generating data migration is represented in Fig. 7. On the

```
foreach extension E_c do
    foreach (object o, key_value key) in E_c do
         foreach attribute a defined in class c in o do
              a := fill_query_a(key)
         end
    end
end
```

Figure 6: Algorithm for attribute assignment

left side the code generation process within schema transformation is displayed. The data migration process consists of three nested loops like in Fig. 6 where the outermost one iterates over all extensions, the second one over all objects in the current extension, and the innermost one over the attributes of the current object. The SQL and OQL queries of the attribute assignment rules are shown in Tab. 2. These queries contain variables which are replaced when the query is instanciated, i.e. when a rule is applied for a certain pattern. Instantiated queries are given in the migration program in

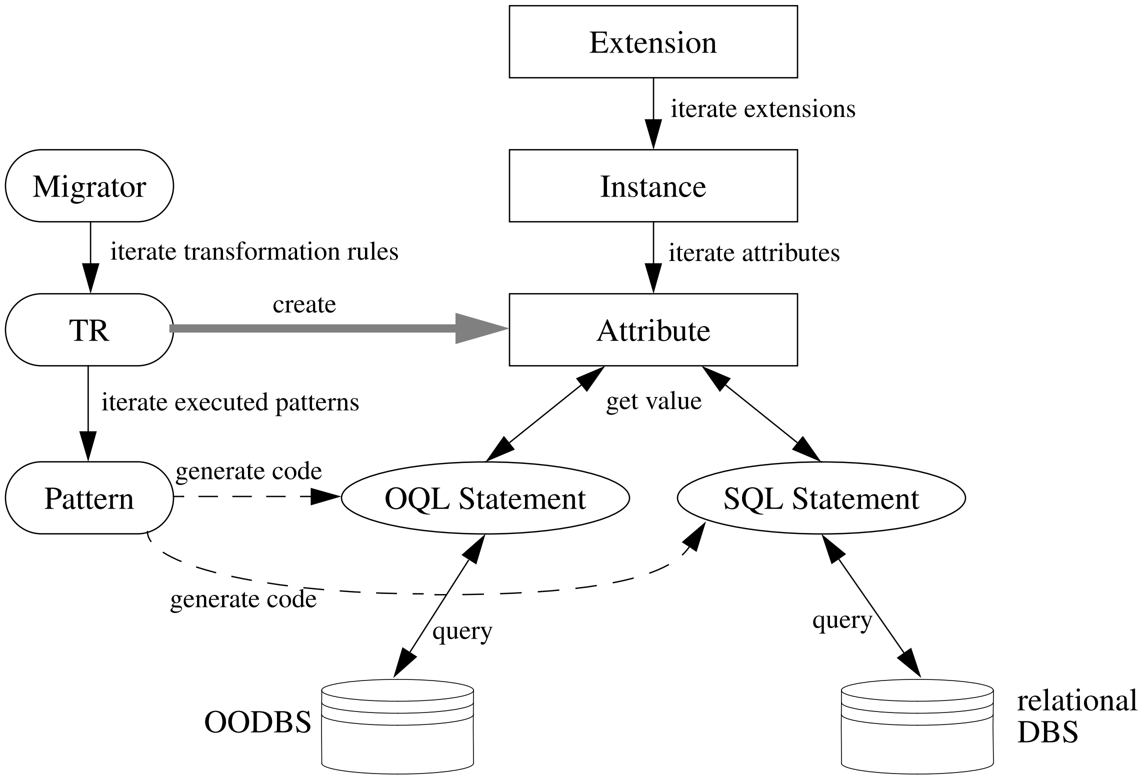

Figure 7: Data Migration

the appendix. The operator `element` is used as a shorthand for the extraction of the element of a singleton set. Not every attribute assignment requires both, an SQL and an OQL query. If both are necessary, the SQL query is executed first and the result is given to the OQL query. Each query is invoked with the instance key value (self.key in SQL-queries) as a parameter and returns the value of the attribute. Both queries together determine the value of an attribute (see the assignment in the innermost loop in Fig. 6).

Rule	Attributes			SQL Query	OQL Query
3	r1(a0)	c1(ac0)	ac0	element(select a0 from r1 where r1.key=self.key)	
4	r1(key, A1,...)	c1(ac2: c2)	ac2	element(select r1.A1 from r1 where r1.key=self.key)	element(select e.object from e in Ec2 where e.key = sql_res)
	r2(key,...)	c2(ac1:set(c1))	ac1	select r1.key from r1 where r1.A1 = self.key	select e.object from e in Ec1 where e.key in sql_res

Table 2: Attribute assignment queries

5	r2(key,...) c2(ac3:set(c3)) ↑ r1(A1,A2) ↙ r3(key,...) c3(ac2:set(c2))	ac3	select r1.A2 from r1 where r1.A1 = self.key	select e.object from e in Ec3 where e.key in sql_res
		ac2	select r1.A1 from r1 where r1.A2 = self.key	select e.object from e in Ec2 where e.key in sql_res
6	r1(key, ...) c1(ac2: c2) ↓ r2(key,...) c2(ac1:c1)	ac2		element(select e.object from e in Ec2 where e.key = self.key)
		ac1		element(select e.object from e in Ec1 where e.key = self.key)
7	r1(A1,A, ...) c1(ac2: c2) ↓ r2(key,...) c2(ac1:set(c1))	ac2		element(select e.object from e in Ec2 where e.key = self.A1)
		ac1		select e.object from e in Ec1 where e.key.A1 = self.key
8	r1(A1,a1) c1(ar1: set(tt(ar1))) ↓ r2(key,...)	ar1	select r1.a1 from r1 where r1.A1 = self.key	
9	r1(A1,A2) c1(ar1: set(tuple(A2) ↓ r2(key,...)	ar1	select r1.A2 from r1 where r1.A1 = self.key	

Table 2: Attribute assignment queries

5 The Architecture of the Migration Tool

The transformation rules and algorithms presented so far are not meant to be used manually by users. Instead, users should be provided with tool support. In this section, we sketch the architecture of an extensive tool.

When a large database has to be migrated, it is obvious that the transformation rules have to be applied for a large number of patterns, and that at most points in time during the migration process several transformation rules can be applied. Therefore the migration tool has to assist the user in several respects to simplify the migration process. A full-fledged migration tool implements the following tasks:

- visualizing the migration process and the artifacts it uses and manipulates.
- controlling the migration process,
- performing the transformation rule application, assuming all the parameters are already determined (the relation, possibly a class, attributes, inclusion dependencies, etc.), and
- proposing which rule to apply in a concrete situation,

These different tasks are reflected by the architecture sketched in Fig. 8.

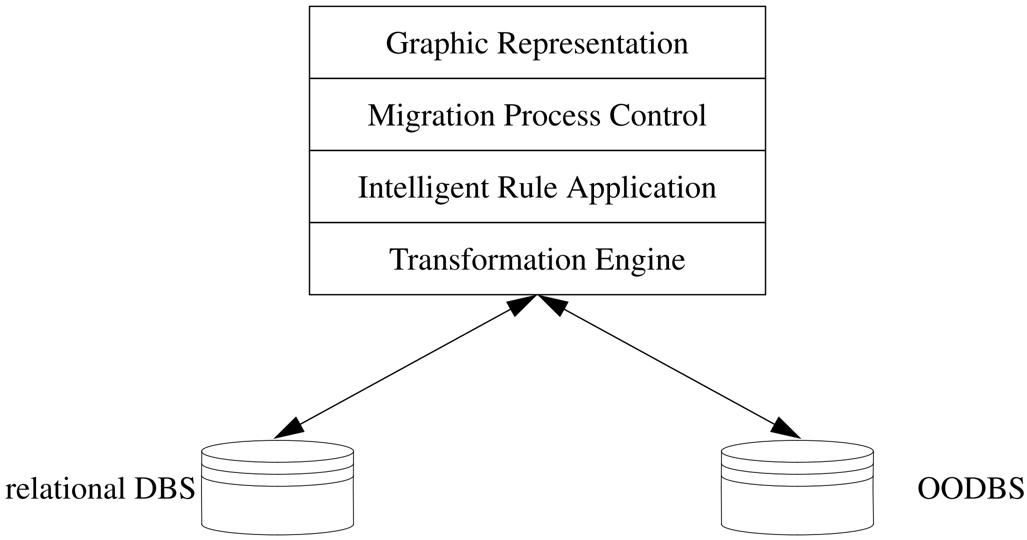

Figure 8: Architecture of the Migration Tool

The lowest level is called *Transformation Engine* and implements the application of transformation rules. All concepts described in this paper belong to this level.

The function of the next level, *Intelligent Rule Application*, is to find out which transformation rules should be chosen at a certain time. There are two cases where this level can support the user. The first case occurs when several conflicting transformation rules can be applied at a certain time, i.e. when the application of one rule disables the application of another rule. In this case, intelligent rule application has to support priorities of transformation rules. The other case occurs when it is possible to apply several non-conflicting transformation rules. An example of this case would be “apply rule 4 for all foreign key dependencies”.

Migration Process Control guides the user and records the ongoing migration process. The user thus has an overview of the migration process, i.e. about which parts of the schema have already been or still have to be transformed. Furthermore, it is possible to undo parts of the migration process.

Finally, the level *Graphical Representation* visualizes all items of the migration process like the relational and object-oriented schema, the transformation rules and the complete progress of the migration process.

6 Implementation

This chapter describes the prototype in which the concepts of schema transformation and data migration have been implemented. The prototype uses O2 as object-oriented DBMS for the object-oriented target schema. To avoid programming an interface to another DBMS, O2 is also used to simulate the relational DBMS in which the source schema is defined, without loss of generality. Programming a gateway for a concrete relational DBMS is a trivial, yet time consuming task and currently beyond the scope of the prototype. Input for the prototype is a relational schema according to the formal model in Chapter 4.1. The prototype contains an internal meta model to describe the relational and the object-oriented schema, their relationships and the transformation rules. The output of the prototype is the definition of the relational and object-oriented schema and the migration program. The generated code for the example can be found in the appendix.

6.1 Schema Transformation

The prototype uses the data definition language O2C to express the object-oriented schema. To implement data migration within O2, the relational schema has to be encoded in O2C as well. For each class of the object-oriented schema the following four parts are created: the declaration of the class itself, a type for the key of the relation from which the class is derived, a type for the extension which is a tuple of the object and the key type mentioned in Chapter 4.3.1, and the extension itself which is defined with the keyword `name`.

6.2 Data Migration

After schema transformation the prototype creates code for an instance creation program and an attribute assignment program according to Chapter 4.3. The attribute assignment queries are embedded in global functions which are invoked from the attribute assignment program. The code in the appendix contains functions of the attributes `AuthID` and `Book_set` of class `Author`. Since the relational database is simulated in O2, the SQL query is implemented with an OQL query as well.

7 Conclusion

In this paper we argued for an interactive environment for transforming relational into object-oriented database schemas. In contrast to other approaches, we propose an interactive and expandable schema transformation process in order to obtain a more suitable object-oriented schema, and present a method for automatic data migration. To that end, we introduced the concept of transformation rules which have a common structure: a pattern, a set of preconditions and a sequence of operations. This structure is flexible enough for defining user-defined transformation rules. The process of schema transformation generates a program for automatic data migration.

Thus, the contribution of this paper and our approach is that it

- provides a model for the flexible specification of transformation rules,
- proposes concrete transformation rules which exploit most of the object-oriented concepts,

• is able to generate data migration programs from schema transformation information.

Object-relational DBMS do not automatically perform such a migration process. Thus, investigating in a migration process is also necessary when using an ORDBMS in order to profit from object-oriented features.

Current approaches for schema transformation, including ours, only consider the structure of the relational schema. In object oriented design the structure and behaviour influence each other and cannot be devoloped independend. Thus, in our future work we will investigate how behavioural information can be added to the object-oriented schema.

References

[AAK+93] J. Albert, R. Ahmed, M. Ketabchi, W. Kent, and M. Shan. Automatic importation of relational schemas in pegasus. In *3rd International workshop on research issues on data engineering: interoperability in multidatabase systems*, 1993.

[ABD+89] M. Atkinson, F. Bancilhon, D. J. DeWitt, K.R. Dittrich, D. Maier, and S.B. Zdonik. The object-oriented database system manifesto. In *1st Intl. Conf. on Deductive and Object-Oriented Databases*, Kyoto, Japan, 1989.

[And94] Martin Andersson. Extracting an entity relationhip schema from a relational database through reverse engineering. In *13th Int. Conf. on ER Approach*, 1994.

[BMP94] J. Biskup, R. Menzel, and T. Polle. Transforming an entity-relationhip schema into object-oriented database schemas. Technical Report 17, Hildesheimer Informatik-Berichte, June 1994.

[CBS94] R. Chiang, T. Barron, and V. Storey. Reverse engineering of relational databases: Extraction of an EER model from a relational database. *Data & Knowledge Engineering*, 12:107–142, 1994.

[CS91] M. Castellanos and F. Saltor. Semantic enrichment of database schemas: An object oriented approach. In Y. Kambayashi et al, editor, *First International Workshop on Interoperability in Multidatabase Systems*, pages 71–78, 1991.

[DA87] K Davis and A Arora. Converting a relational database model into an entity-relationship model. In *6th Int. Conference of the Entity Relationship Approach*, 1987.

[EN94] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 2 edition, 1994.

[FV95] C. Fahrner and G. Vossen. Transformation relationaler datenbank-schemas in objekt-orientierte schemas gemaess ODMG-93. In *Datenbanksysteme in Buero, Technik und Wissenschaft, GI-Fachtagung*, pages 111–129, Dresden, March 1995. Informatik Aktuell, Springer.

[Gra95] Ian Graham. *Migrating to Object Technology*. Addison-Wesley, 1995.

[HCTJ93] J. Hainaut, M. Chandelon, C. Tonneau, and M. Joris. Contribution to a theory of database reverse engineering. In *IEEE Working Conf. on Reverse Engineering*, pages 161–170, Baltimore, May 1993. IEEE CSP.
http://www.info.fundp.ac.be/~dbm/references.html

[HEH+95] J. Hainaut, V. Englebert, J. Henrard, J. Hick, and D. Roland. Requirements for information system reverse engineering support. In *Proc. of the IEEE Working Conf. on Reverse Engineering*, Toronto, July 1995, IEEE Computer Society Press.
http://www.info.fundp.ac.be/~dbm/references.html

[HTJC93] J. Hainaut, C. Tonneau, M. Joris, and M. Chandelon. Transformation-based database reverse engineering. Technical report, Institut d'Informatique, University of Namur, 1993.

[Joh94] P. Johannesson. A method for transforming relational schemas into conceptual schemas. In *10th International Conference on Data Engineering*, pages 190–201, Houston, February 1994. IEEE Computer Society.

[JSZ96] Jens Jahnke, Wilhelm Schäfer, and Albert Zündorf. A design environment for migrating relational to object oriented database systems. In *International Conference on Software Maintenance*. IEEE Computer

Society, 1996.

[JSZ97] Jens Jahnke, Wilhelm Schäfer, and Albert Zündorf. Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications. In *6th European Software Engineering Conference*, 1997.

[PB94] W. J. Premerlani and M. R. Blaha. An approach for reverse engineering of relational databases. *Communications of the ACM*, 37(5):42–49, May 1994.

[PTBK96] J-M. Petit, F. Toumani, J-F. Boulicaut, and J. Kouloumdjian. Towards the reverse engineering of denormalized databases. In *International Conference on Data Engineering*. IEEE, 1996.

[Sto95] Michael Stonebraker. Object-relational DBMS - the next wave. White Paper, Illustra Information Technologies Inc., 1995.

[VA95a] M. Vermeer and P. Apers. Enhancing the semantics of federated schemata by translating SQL-queries into object methods. Technical report, University of Twente, Computer Science Department, 1995.

[VA95b] M. Vermeer and P. Apers. Reverse engineering of relational database applications. In *14th International Conference on ER/OO Modelling*, 1995.

[YL93] L. Yan and T. Ling. Translating relational schema with constraints into OODB schema. *IFIP Transactions of Interoperable Database Systems (DS-5)*, 1993.

Appendix

The following presents an extract of the generated O2C code for the library example.

Declarations of the relational schema.

```
type Author_Type :tuple(
    AuthID: integer,
    Name: string,
    FirstName: string,
    Origin: string);
name Author :set(Author_Type);
```

Declarations of the object-oriented schema.

```
class Author inherit Object public type tuple(
    Book_set: set(Book),
    AuthID: integer,
    Name: string,
    FirstName: string,
    Origin: string)
end;
type Author_KeyType :tuple(
    AuthID: integer);
type Author_ExtType :tuple(
    object: Author,
    key: Author_KeyType);
name Author_Ext :set(Author_ExtType);
```

Generated implementation of instance creation of class Author.

```
program body instance_creation in application Migration
{
o2 Author_ExtType Author_ext_tuple;
o2 Author_Type Author_tuple;
o2 Author_KeyType Author_key_tuple;
o2 Author Author_object;
...
for(Author_tuple in Author)
{
    Author_key_tuple.AuthID = Author_tuple.AuthID;
    for(Author_ext_tuple in Author_Ext)
```

```
    if(!(Author_ext_tuple.key == Author_key_tuple))
    {
         Author_ext_tuple.object = new Author;
         Author_ext_tuple.key = Author_key_tuple;
         Author_Ext += set(Author_ext_tuple);
    }
}
...
}
```

Generated functions for attribute assignment.

```
function Author_AuthID(Author_key_tuple: Author_KeyType): integer;
function body Author_AuthID(Author_key_tuple: Author_KeyType): integer
{
    o2 integer result;
    /* SQL query */
    o2query(result,"element(select t.AuthID \
                             from t in Author \
                             where t.AuthID = $1)",
                             Author_key_tuple.AuthID);
    return(result);
};
function Author_Book_set(Author_key_tuple: Author_KeyType): set(Book);
function body Author_Book_set(Author_key_tuple: Author_KeyType): set(Book)
{
    o2 set(Book) result;
    o2 set(Book_KeyType) sql_res;
    /* SQL query */
    o2query(sql_res,"select tuple(BookID: t.BookID) \
                     from t in Written \
                     where t.AuthID = $1",
                     Author_key_tuple.AuthID);
    /* OQL query */
    o2query(result,"select t.object \
                    from t in Book_Ext \
                    where t.key in $1",
                    sql_res);
    return(result);
};
```

Generated process of attribute assignment.

```
program body attribute_assignment in application Migration
{
o2 Author_ExtType Author_ext_tuple;
o2 Author_Type Author_tuple;
o2 Author_KeyType Author_key_tuple;
o2 Author Author_object;
...
for(Author_ext_tuple in Author_Ext)
{
Author_ext_tuple.object->Book_set = Author_Book_set(Author_ext_tuple.key);
Author_ext_tuple.object->AuthID = Author_AuthID(Author_ext_tuple.key);
Author_ext_tuple.object->Name = Author_Name(Author_ext_tuple.key);
Author_ext_tuple.object->FirstName = Author_FirstName(Author_ext_tuple.key);
Author_ext_tuple.object->Origin = Author_Origin(Author_ext_tuple.key);
}
...
};
```