ROVER: A Framework for the Evolution of Relationships *

Kajal T. Claypool, Elke A. Rundensteiner, and George T. Heineman

Worcester Polytechnic Institute, 100 Intitute Road,
Worcester, MA, USA
{kajal, rundenst, heineman}@cs.wpi.edu
http://davis.wpi.edu/dsrg

Abstract. Relationships have been repeatedly identified as an important object-oriented modeling construct. Today most emerging modeling standards such as the ODMG object model and UML have some support for relationships. However while dealing with schema evolution, OODB systems have largely ignored the existence of relationships. We are the first to propose comprehensive support for relationship evolution. A complete schema evolution facility for any OODB system must provide (1) primitives to manipulate all object model constructs; (2) and also maintenance strategies for the structural and referential integrity of the database under such evolution. We hence propose a set of basic evolution primitives for relationships as well as a compound set of changes that can be applied to the same. However, given the myriad of possible change semantics a user may desire in the future, any pre-defined set is not sufficient. Rather we present a flexible schema evolution framework which allows the user to define new relationship transformations as well as to extend the existing ones. Addressing the second problem, namely of updating the schema evolution primitives to conform to the new set of invariants, can be a very expensive re-engineering effort. In this paper we present an approach that de-couples the constraints from the schema evolution code, thereby enabling their update without any re-coding effort.

Keywords: Schema Evolution, Relationships, Object-Oriented Databases, Consistency Management.

1 Introduction

Object-oriented database (OODB) systems today are popular with communities that require the modeling of complex data types, such as CAD, multimedia applications and e-commerce applications [16]. The very nature of these applications has also brought forth the need for the OODB to provide mechanisms

* This work was supported in part by the NSF NYI grant #IRI 94-57609. We would also like to thank our industrial sponsors, in particular, IBM for the IBM partnership award and Excelon Inc. for software contribution.

for (1) modeling associations between types and (2) for dynamically changing not only the data but also the structure of the types that contain them [18]. Research and industry alike have rushed forth to fulfill these requirements but have approached them as two independent problems with independent solutions. Relationship modeling has been a much studied topic such as in composite objects [10] or in aggregation relationships [2, 5] and has been adopted to some degree in commercial OODB systems [14, 15]. On the other hand, while most OODB systems to date provide some basic evolution support [19, 14, 1], the OODB community at large does not treat the relationship construct as a first-class citizen. Thus, while there is some relationship support at the data level, evolution support for relationships has not been investigated. Our work is the first to bridge this gap and to provide a complete framework for the support of schema evolution for relationships.

A complete schema evolution facility for any OODB system must provide the ability to (1) manipulate and change all of the object model constructs, i.e, a new set of schema evolution primitives to evolve the new construct(s); (2) and to maintain the structural and referential integrity of the database, i.e., it must modify the existing schema evolution primitives to assure that they obey the new constraints of the new object model augmented with the relationship construct. For the relationship construct [5] we now propose, a set of schema evolution primitives to handle their addition and deletion. One example of such a primitive is `form-relationship` that creates a bi-directional relationship between two classes that already have two independent uni-directional relationships.

Evolution primitives maintain the consistency of an OODB by preserving the invariants of the object model at all times. However, an update to the object model changes the set of invariants. Hence, the existing evolution primitives must be updated to conform to the invariants of the new object model. Consider for example, the existing evolution primitive `delete-class`. Today, while most OODB systems provide support for relationships by modeling them as reference attributes, the evolution primitives such as `delete-class` treat them as ordinary literal attributes. And while at the object level most OODB systems do referential integrity checks to ensure referential consistency, at the schema level, referential relationships are largely mis-treated during schema evolution, resulting in structurally inconsistent schemas. And while most systems may be able to trap and deal with the referential problems, the schema inconsistencies are largely ignored.

However, updating the schema evolution primitives to now conform to the new set of invariants is a huge re-engineering effort. To diminish the cost of updating an existing schema evolution facility while still providing a consistency preserving schema evolution facility, we propose Evolution Wrappers based on the concept of Software Contracts [12] to effectively maintain existing schema evolution primitives in the light of a changing object model. These contracts, declarative in nature, are easy to update and can be changed without the overhead of re-coding.

For relationships, a complex construct with numerous special properties such as cardinality, uni- vs bi-directional, etc., a pre-defined primitive set of operations is not sufficient. Many different primitives may be combined to compose more complex transformations. Moreover, this must all be executed *atomically* like any pre-defined system-provided primitive. We provide a framework that offers such composite evolution operations while maintaining the integrity and the atomicity of the operations themselves.

A big drawback of the compound primitives is their lack of flexibility. A user such as a DBA administrator, may wish to create a bi-directional relationship between two completely disjoint classes. Two options for the user are to (1) have the system provide a schema evolution primitive to handle this scenario or (2) write an ad-hoc program to accomplish the same. From a system's perspective to apriori assess the needs of the users and to plan for it in a hard-coded fashion is a hard if not an impossible problem. On the other hand, ad-hoc programs by the users do not have any guarantee of consistency or atomicity. Thus it is necessary to offer extensibility in our framework that would allow not just the system but also the users to express desired transformations while still providing the guarantee of database integrity as well as the atomicity of the transformations. The solution we present in this paper based on SERF [6] provides an extensible and flexible schema evolution framework for supporting complex relationship evolution.

Summary. In this paper we present a complete solution framework, *ROVER* for the evolution of relationships which provides for consistent evolution of relationships while supporting extensibility for user-defined evolution operations. Here, (1) We identify the problem that current OODBs while offering relationship support in their object models do not handle evolution of such relationships. We characterize the *consistency problems* that arise from the use of the existing evolution primitives in these systems; (2) We propose a minimal set of *new evolution primitives* needed for supporting the evolution of both uni-directional and bi-directional relationships; (3) We propose a set of *compound evolution primitives* for relationships; (4) We generalize our approach such that users as opposed to the system can now extend the set of schema evolution operations provided by the OODB with their own complex schema evolution operations for relationships; and (5) We present ROVER Wrappers to reduce the re-engineering costs that would otherwise be incurred in updating the old schema evolution primitives to conform to the new set of invariants.

2 Background

In this section, we briefly introduce relationship constructs as defined by ODMG [5], while full details can be found in [5]. Paralleling the concept of foreign keys in relational databases, object models almost always have support for the association between two classes. Most models support the notion of a reference attribute which defines a one-way association between two classes. The ODMG object model also defines the notion of a bi-directional association wherein if

class A refers to class B then class B must refer to class A. The user can define the cardinality of these references as one-to-one, one-to-many or many-to-many. To capture this notion of association, we use the *referential relationship* ($\longrightarrow$) that specifies when one type refers to another type; and a *bi-directional relationship* ($\longleftrightarrow$) that specifies a referential relationship and its inverse.

Table 1 presents some of the notation and functions for the meta information as used in the paper. More details can be found in [7].

Term	Description
$\mathbf{types}(\mathcal{C})$	The set of all types in the system
$s,\ t,\ T,\ \bot$	Elements of $\mathbf{types}(\mathcal{C})$
super(t)	The set of all direct supertypes of type t
sub(t)	The set of all direct subtypes of type t
in-paths(t)	The set of all paths <c,r> referring to type t
in-degree(t)	The count of all paths referring to type t
out-paths(t)	The set of all paths <t,r> going out of type t
obj-in-degree(o_i)	The number of objects referring to the object o_i

Table 1. Notation for Axiomatization of Schema Changes

In Table 2 we list the basic schema evolution operations that are supported by most OODBs [14, 19].

Evolution Primitive	Description
add-class(c, $\mathcal{C}$)	Adds new class c to $\mathcal{C}$ in the schema **S**
delete-class(c)	Deletes class c from $\mathcal{C}$ in the schema **S**
add-ISA-edge(c_x, c_y)	Adds an inheritance edge from c_x to c_y
delete-ISA-edge(c_x, c_y)	Deletes the inheritance edge from c_x to c_y
add-attribute(c_x, a_x, t, d)	Add attribute a_x of type t and default value d to class c_x
delete-attribute(c_x, a_x)	Deletes the attribute a_x from the class c_x

Table 2. Taxonomy of Basic Schema Evolution Primitives for Classes, Attributes and Inheritance Hierarchy.

3 Minimal Primitives for Relationship Evolution

In this section we present a set of evolution primitives needed for the evolution of uni-directional (unary) as well as for bi-directional (binary) relationships. The

primitives presented here are *minimal* in that they cannot be decomposed into any other evolution primitives and *essential* in that they are all required for the evolution of relationships. They can be composed together with other evolution primitives to form more complex transformations, as we will discuss in Section 5.

3.1 Evolution of Unary Relationships

As per the ODMG object model and other object models [19], unary relationships are generally modeled via the use of a reference attribute. For example, Figure 2 shows an unary relationship between classes `Teacher` and `Course` via the reference attribute `teaches`. We propose two evolution primitives *add-reference-attribute* and *delete-reference-attribute* that allow us to add and delete an unary relationship.

Fig. 1. Graphical Schema Description of the Uni-Directional Relationship between Two Classes `Teacher` and `Course`.

```
class Teacher
{
    attribute Course teaches;
}
```

Fig. 2. ODMG Syntax for a Uni-Directional Relationship, `teaches`.

The add-reference-attribute primitive adds a uni-directional relationship between two types. For example, the primitive *add-reference-attribute (Teacher, teaches, Course, null)* adds a complex attribute `teaches` of the type `Course` to the class `Teacher`. Its default value is set to `null`. The domain type of the reference attribute signifies the cardinality of the relationship. Thus, a `Set` signifies a *many* relationship and a `type` signifies a *one* relationship. In this example, we thus have an *one-to-one* relationship as `Course` is an application `type` signifying a cardinality of one.

The delete-reference-attribute primitive allows the deletion of an existing uni-directional relationship between two types. For example, the primitive *delete-reference-attribute(Teacher, teaches)* deletes the complex attribute `teaches` from the class `Teacher`. This primitive is an inverse operation of the primitive *add-reference-attribute.*

3.2 Evolution of Bi-directional Relationships

The ODMG relationship syntax for Figure 3 is given in Figure 4. In the class `Teacher`, the attribute `teaches` is a *reference* attribute of type `Course` and the inverse of this relationship is given by the attribute `is-taught-by` in class

`Course`. A binary relationship thus is modeled by the following characteristics: a *source-class* (`Teacher`), *inverse-class* (`Course`), the *source-relationship-name* (`teaches`), the *inverse-relationship-name* (`is-taught-by`), cardinality of the relationship in the *source* class referred to as *source-card* (`many`), cardinality of the relationship in the *inverse* class referred to as *inverse-card* (`one`), type of storage (class or collection) for the source relationship *source-type* (`set`), and type of storage for the inverse relationship *inverse-type* (`Teacher`).

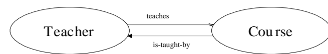

Fig. 3. Graphical Schema Description of the teaches - is-taught-by Relationship.

```
class Teacher {
  relationship set<Course> teaches
     inverse Course::is-taught-by;
}
class Course {
  relationship Teacher is-taught-by
     inverse Teacher::teaches;
}
```

Fig. 4. ODMG Syntax for Specifying a Bi-Directional Relationship.

A bi-directional relationship can be broken down into a pair of uni-directional relationships between the two classes. Hence we propose that the only two additional primitives needed for the manipulation of bi-directional relationships are: *form-relationship* and *drop-relationship* (see Table 3).

The form-relationship primitive elevates the status of two already existing uni-directional relationships between two types. For example, in Figure 3 *form-relationship(Teacher, teaches, Course, is-taught-by)* indicates that from now onwards the two uni-directional relationships `Teacher.teaches` and `Course .is-taught-by` are to be modeled as a bi-directional relationship. Hence, any update of objects in one class `Teacher` will be automatically reflected in the other class `Course`.

The drop-relationship primitive transforms a bi-directional relationship to a pair of uni-directional relationships. For example, *drop-relationship(Teacher, teaches, Course, is-taught-by)* drops the binary relationship down to two separate uni-directional relationships `Teacher.teaches` and `Course.is-taught-by`. Henceforth, the two relationships are maintained independently of the other. Any update to one relationship will not effect the other relationship.

Other evolution operators needed for the support of relationships such as changing the cardinality or changing the domain type can be accomplished via a combination of the already available basic primitives and hence do not require any additional primitives (see Section 5 for more details). Table 3 summarizes the new evolution primitives for relationships.

Evolution Primitive	Description
add-reference-attribute(c_x, r_x, c_y, d)	Add unary relationship from class c_x to class c_y named r_x with default value d
delete-reference-attribute(c_x, r_x)	Delete unary relationship in class c_x named r_x
form-relationship(c_x, r_x, c_y, r_y)	Promotes the specified two unary relationships to a binary relationship
drop-relationship(c_x, r_x, c_y, r_y)	Demotes the specified binary relationship to two unary relationships

Table 3. Taxonomy of Basic Evolution Primitives for Relationships.

4 Compound Evolution Operations for Relationships

While providing some basic capability to evolve relationships these evolution alone are not adequate to manipulate all the properties of a relationship. For instance, using only the primitives in Section 3 the users are unable to change the cardinality of the relationship, i.e., change from a `collection` to an application `type` or vice versa. This change could however be accomplished by a combination of other existing schema evolution primitives. Similar to Breche [4] and Lerner [11] we now support atomic system-defined evolution operations that are composed of several (basic) evolution primitives.

In our work we identify the set of complex operations to evolve the different properties of a relationship construct. As an example, Figure 5 depicts a system-defined operation to accomplish a cardinality change, i.e, it changes the cardinality of a relationship from `many` (set) to `one` (an application type). In this example, we first use the `drop-relationship` primitive to break the bi-directional relationship into two individual uni-directional relationships. The primitive `add_atomic_attribute` adds a temporary attribute `tmp-attr` whose domain is representative of the cardinality change. Thus, the domain of `tmp-attr` is $\mathtt{t}_d$. In the next step, for all objects in the extent of the class $\mathtt{c}_x$, we need to convert the set-value ($\mathtt{r}_x$) to a single value. We use a system-defined method `most-common`($\mathtt{r}_x$) that finds the most commonly occurring object $\mathtt{a}_i$ in a given collection $\mathtt{r}_x$. Thus, for all objects in the extent of $\mathtt{c}_x$ the single-value attribute `tmp-attr` is assigned `a`, the most common occurring value in the object's collection $\mathtt{r}_x$. We then delete the attribute $\mathtt{r}_x$ and rename the temporary attribute `a` to $\mathtt{r}_x$. As a last step we re-formulate the bi-directional relationship between the two classes. This evolved relationship is now a one-to-one relationship as opposed to a one-to-many relationship.

Table 4 summarizes the set of compound changes necessary to manipulate a bi-directional relationship.

```
change-cardinality-m1(c_x, r_x, t_d, inverse-c_x, inverse-r_x)
{
    // Drop the relationship to two individual uni-directional relationships
    drop-relationship(c_x , r_x , inverse-c_x , inverse-r_x)

    // Create a new attribute in class c_x
        add_atomic_attribute(c_x , tmp-attr, t_d , null);

    // For each object in the extent, find the
    // most-common-value of the set r_x and use
    // that as a representative value for the tmp-attr.
    while ((o = extent(c_x).nextElement()) != null)
        o.tmp-attr = most-common-value(o.r_x);

    // Remove the attribute r_x from class c_x
    delete_attribute(c_x , r_x);

    // Rename the attribute attr to r_x
    rename_attribute(c_x , attr, r_x);

    // Form the bi-directional relationship from two uni-directional relationships
    form-relationship(c_x , r_x , inverse-c_x , inverse-r_x)
}
```

Fig. 5. A Compound Primitive to Change the Cardinality of a Relationship.

5 Flexible Evolution of Relationships

One of the biggest drawback of the compound evolution primitives presented in Section 4 is their *pre-determined* semantics for the compound changes. Consider for example the rather simple compound change, `cardinality-change` shown in Figure 5. We arbitrarily use the most commonly occurring object value as the single-value representative. However, other conversion at the object level such as taking the first, last or median value of the set values are possible. By hard-coding these compound changes, the user has no flexibility to alter their semantics.

This set of evolution primitives, simple or compound, is *pre-determined* and *fixed* not allowing the user[1] any flexibility. Consider that the user now wants to build a bi-directional relationship between two types where only one uni-directional relationship exists [2]. Their only alternative would be an ad-hoc program to combine existing primitives but without the atomicity

[1] By user we imply a database administrator who has knowledge and permission to alter the structure of the database.

[2] This is as opposed to two uni-directional relationships between the two types, as required by schema evolution primitive `form-relationship`.

Compound Evolution Operation	Description
change-cardinality-1m(c_x, r_x, m_d)	Changes cardinality of relationship r_x in class c_x from its current type to collection m_d
change-cardinality-m1(c_x, r_x, t_d)	Changes cardinality of relationship r_x in class c_x from collection m_d to type t_d
change-rel-name(c_x, r_x, y_d)	Changes name of relationship r_x in class c_x to y_d
change-type(c_x, r_x, t_d)	Changes type of relationship r_x in class c_x to t_d

Table 4. Taxonomy of Compound Evolution Operations for Relationships.

and consistency guarantees that a system-defined primitive would have. To address these issues we present our framework which gives users the *flexibility* to alter semantics for compound changes as well as *extensibility* to define their own schema evolution operations within a framework guaranteeing the atomicity of the operation and the consistency of the database.

5.1 SERF: A Framework for Extensible Schema Evolution

Our solution [6] is based on the hypothesis that complex schema evolution transformations can be broken down into a sequence of basic evolution primitives, (an invariant-preserving atomic operation) glued together by a query language such as OQL [5].

The user could now on the fly introduce any new transformation when desired. For example the transformation in Figure 6 converts the schema of Figure 1 to the schema depicted in Figure 3. To accomplish this we first create the uni-directional relationship between the class `Course` and the class `Teacher` with the name `is-taught-by`. Once we have two uni-directional relationships we can use the *form-relationship* evolution primitive to convert it to a bi-directional relationship. This operation is defined by the user within the confines of our proposed framework that can guarantee the atomicity and consistency of the system.

SERF Template. An OQL transformation as in Figure 6 *flexibly* allows a user to define different schema transformation. However, these transformations are *not generic*, i.e., they cannot be applied to other existing classes or different schemas. For example, the transformation shown in Figure 6 is valid only for the classes `Teacher` and `Course`. To address this, we introduce the notion of templates [6]. A template uses the query language's ability to query over the meta data (as stated in the ODMG Standard) and is enhanced by a name and a set of parameters to make transformations *generic* and *re-usable*. Figure 7 shows a templated form of the transformation presented in Figure 6. Here we query over

```
// This transformation adds a relationship between two partially disjoint classes
// i.e., one class has a one-sided relationship to the other. This transformation
// also performs the object transformations.

add_reference_attribute (Course, is-taught-by, Teacher, null);


// Get the extent of the class

define extents (cName) as
    select c
     from cName c;

// Do the object transformations
// set: Course.is-taught-by = Teacher
for all obj in extents (Teacher):
     obj.teaches.set(obj.teaches.is-taught-by, obj);

// form a binary relationship
form-relationship(Teacher, teaches, Course, is-taught-by);
```

Fig. 6. Transformation From Uni-directional to Bi-directional Relationship

the meta-data to discover the **type** of the attribute `source-attrib-name` and then add the `inverse-attrib-name` to it (shown in Courier font). The object transformation here remains the same as in the transformation of Figure 6. This template shows how we can take advantage of the single referential relationship and perform the object transformations all within our template structure.

6 Contract-Based Solution for Consistent Relationship Evolution

Conventionally, schema evolution primitives contain hard-code constraints that parallel the invariants of the object model. These constraints must be satisfied in order to guarantee the consistency of the system. Changes in the object model result in changes to the invariants which may in turn be reflected as an update to schema evolution operations [7]. This is an expensive process requiring the re-engineering of the affected software. Our approach *ROVER* instead focuses on de-coupling the constraints from the actual implementation code of the schema evolution operations. To accomplish this we introduce the notion of *contracts*, a declarative mechanism for expressing the constraints for a template. A template with contracts is termed a *ROVER Wrapper*. Changes to the invariants of the object model now merely result in the update of the declarative contracts associated with the evolution operations rather than the update of the actual system code. Below we briefly introduce *contracts* and show how the de-coupling of constraints can be achieved while more details can be found in [7].

```
begin template add-inverse-relationship ( source-class, source-rel-name, inverse-name)
{
    // find the inverse class
        refClass = select c.attrType
                   from Attribute c
                   where c.name = $source-relName and
                   c.parent.name = $source-class;


    // add the reference attribute to the inverse class
    add-reference-attribute (refClass, $inverse-name, $source-class, null);


    // get the extent of the class
    define extents (cName) as
        select c
        from cName c;

    // object transformations
    for all obj in extents ($source-class):
        obj.$source-rel-name.set( obj.$source-rel-name.$inverse-name, obj);

    // create the bi-directional relationship
    form-relationship ($source-class, $source-rel-name, refClass, $inverse-name);
}
end template
```

Fig. 7. Template for Converting a Uni-directional Relationship to a Bi-directional Relationship

Contracts provide a declarative description of the behavior of a template (or primitive) as well as a mechanism for expressing the constraints that must be satisfied prior to the execution of the actual evolution primitive. Contracts are divided into two categories **pre-conditions** and **post-conditions**.

The constraints, termed *pre-conditions*, are placed prior to any body of template code (OQL statement including system-defined schema evolution primitive). The *pre-conditions* are separated from the actual OQL statements by means of the keyword **requires**. *Post-conditions*, a set of contracts that appear after the body of the actual schema evolution operation at the end of the SERF template, specify the behavior of the primitives. These post-conditions are preceded by the keyword **ensures:** and describe the exact changes that are made to the schema by the evolution operator and hence its behavior.

Example: As an example consider the addition of relationship constructs to the object model of an OODB system. An upgrade to the schema evolution facility in this case requires new schema evolution primitives to handle the creation, modification and deletion of uni-directional and/or bi-directional relationships. This upgrade cannot be circumvented and hence new schema evolution primitives must be added to the system. However, an update of all existing schema evolution operations to conform to the new set of invariants is also required. For example, to delete a class prior to the existence of relationships, the constraint that a class needed to be a *leaf* class was necessary to ensure that the result-

ing schema and database was consistent, i.e., the `delete-class` preserved the database consistency. With the addition of relationships, this constraint alone is not sufficient. We now also need to ensure that the `to-be-deleted` C_i class is not referred to by another class. Moreover, no objects in the database must refer to the objects of the C_i class. So while the conditions that need to be enforced prior to the execution the schema evolution operation have to be upgraded, the actual actions of the operations do not change. Hence the evolution primitive `delete-class` itself does not change.

Figure 8 shows the constraints after the addition of a relationship for the *delete-class* primitive as pre-conditions[3]. Thus, in this model it is easy to extend or modify the constraints without re-writing the code for the evolution primitive.

delete-class (C_i)
{
  requires:
    $C_i \in \mathcal{C} \wedge$
    $\sigma(C_i) \in \mathbf{types}(\mathcal{C}) \wedge$
    $sub(C_i) = 0 \wedge$
    $in\text{-}degree(C_i) = 0 \wedge$
    $\forall\ o_i \in \mathbf{extent}(t)$
      $obj\text{-}in\text{-}degree(o_i) = 0$

  `template body here`
}

Fig. 8. Pre-Conditions for Delete-Class Primitive in Contractual Form

delete-class (C_i)
{ `template body here`

  ensures:
    $C_i \notin \mathcal{C} \wedge$
    $\sigma(C_i) \notin \mathbf{types}(\mathcal{C}) \wedge$
    $\forall <C_x, r_x> \in out\text{-}paths(C_i)$
      $(<C_i> \notin in\text{-}paths(C_x))$
$\wedge$
    $\forall\ C_x \in super(C_i)$
      $(C_i \notin sub(C_x))$
}

Fig. 9. Post-Conditions for the Delete-Class Primitive Template

Advantages of Contracts. ROVER Wrappers provide faster updates to the OODB system when the underlying object model is updated for example with relationships. Namely, we would simply now add additional declarative constraints and behavior to existing schema evolution primitive templates rather than having to update the system code. Moreover these contracts can detect erroneous conditions prior to the execution of the schema evolution primitives. This would help avoid the cost of roll-backs in cases of failure. The *post-conditions* represent the desired final state and can hence be used for testing the correctness of the ROVER Wrapper.

7 Related Work

Semantic modeling research has looked into the modeling of relationships and the different semantics that can be applied for these relationships [3]. In object

[3] The notation used here is a set-theoretic version of the contract language.

databases, Kim, Bertino and others [9] have examined the part-whole relationship (composite objects). Bertino et al. [2] have presented a formal composite object model that now supports referential integrity constraints for the ODMG object model. None of them have studied the issue of schema evolution on such object models with relationships.

Most research and commercial systems provide schema evolution in the form of a fixed set of evolution primitives [1, 14, 19]. In recent years, research has begun to focus on the issues of supporting more complex schema evolution operations. Breche and Lerner [4, 11] studied the design of a set of more complex operations. Lerner [11] has proposed compound type changes like *Inline*, *Encapsulate*, *Merge*, etc. from the aspect of discovering the transformation sequences to map between these given two schemas. Lastly, our previous work on SERF [6] has provided a framework that allows the user to define arbitrarily complex schema changes by composing them out of the basic set of evolution primitives and OQL. None of these previous approaches have looked at the problem of evolution (be it simple or complex) in the context of object models with relationships.

Consistency management is often done at runtime and is normally handled by transaction roll-backs. Work has been done towards providing behavioral consistency, i.e., the consistency of class methods under evolving environments [8, 13]. While there are similarities in that their algorithms are also detecting broken references, their approach is also hard-coded. With our approach we push these constraints/invariants to a higher level thus making them easy to update and also allowing for pre-execution verification checks.

8 Conclusion

In this work, we present the first solution for the evolution of relationships. We provide a flexible mechanism to handle the evolution of relationships. We incorporate the notion of *contracts* for ROVER Wrappers as an alternative to hard-coding the constraints into the schema evolution primitive and thus helping to offset the re-engineering cost. We provide implementation details on ROVER in [7]. The core SERF system for flexible transformation was demonstrated at SIGMOD 2000 [17].

References

1. J. Banerjee, W. Kim, H. J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *SIGMOD*, pages 311–322, 1987.
2. E. Bertino and G. Guerrini. Extending the ODMG Object Model with Composite Objects. In *OOPSLA*, pages 259–270, 1998.
3. G. Booch. *Object-Oriented Analysis and Design*. Benjamin Cummings Publications, 1994.
4. P. Bréche. Advanced Primitives for Changing Schemas of Object Databases. In *Conference on Advanced Information Systems Engineering*, pages 476–495, 1996.

5. Cattell, R.G.G and et al. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, Inc., 1997.
6. K.T. Claypool, J. Jin, and E.A. Rundensteiner. SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework. In *Int. Conf. on Information and Knowledge Management*, pages 314–321, November 1998.
7. K.T. Claypool, E.A. Rundensteiner, and G.T Heineman. Extending Schema Evolution to Handle Object Models with Relationships. Technical Report WPI-CS-TR-99-15, Worcester Polytechnic Institute, March 1999.
8. C. Delcourt and R. Zicari. The Design of an Integrity Consistency Checker (ICC) for an Object Oriented-Database System. In P. America, editor, *ECOOP*, pages 97–117, 1991.
9. W. Kim, E. Bertino, and J. F. Garza. Composite objects revisited. *SIGMOD*, pages 337–347, 1989.
10. W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.
11. B.S. Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. Technical Report UM-CS-96-044, University of Massachusetts, Amherst, Computer Science Department, 1996.
12. B. Meyer. Applying "Design By Contract". *IEEE Computer*, 25(10):20–32, 1992.
13. M.A Morsi, S. Navathe, and Shilling J. On Behavioral Schema Evolution in Object-Oriented-Database System. In *Int. Conference on Extending Database Technology (EDBT)*, pages 173–186, 1994.
14. Object Design Inc. *ObjectStore - User Guide: DML. ObjectStore Release 3.0 for UNIX Systems.* Object Design Inc., December 1993.
15. Objectivity Inc. White Paper, Schema Evolution in Objectivity, February 1994.
16. Joan Peckham, Bonnie MacKellar, and Michael Doherty. Data model for extensible support of explicit relationships in design databases. *VLDB Journal*, 4(2):157–191, 1995.
17. E.A. Rundensteiner, K.T. Claypool, and L. et. al Chen. SERFing the Web: A Comprehensive Approach for Web Site Management. In *Demo Session Proceedings of SIGMOD'00*, 2000.
18. D. Sjoberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1):35–54, January 1993.
19. O_2 Technology. *O_2 Reference Manual, Version 4.5, Release November 1994.* O_2 Technology, Versailles, France, November 1994.