

ROVER: Flexible Yet Consistent Evolution of Relationships *

Kajal T. Claypool

Elke A. Rundensteiner

George T. Heineman

Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609-2280
{kajal|rundenst|heineman}@cs.wpi.edu

Abstract

Relationships have been repeatedly identified as an important object-oriented modeling construct. Most emerging modeling standards such as the ODMG object model and UML have some support for relationships. However OODB systems have largely ignored the existence of relationships during schema evolution. We are the first to propose comprehensive support for relationship evolution. A complete schema evolution facility for any OODB system must provide primitives to manipulate all object model constructs; and maintenance strategies for the structural and referential integrity of the database under such evolution. We 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 that allows the user to define new relationship transformations as well as to extend existing ones. Addressing the second problem, namely of updating 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. We also present an approach that can be used to verify the correctness of these complex evolution operations using the de-coupled constraints.

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

1 Introduction

Object-oriented database (OODB) systems today are popular with communities that model complex data types, such as Computer Aided Design (CAD), multimedia applications and e-commerce applications [39]. The very nature of these applications has also brought forth the need for the OODB systems to provide mechanisms for (1) modeling associations between types and (2) for dynamically

*This work was supported in part by the NSF NYI grant #IIS-97-29878, #IIS-97-96264 and #IIS-99-88776. We would also like to thank our industrial sponsors, in particular, IBM for the IBM partnership award and Excelon Inc. for software contribution.

changing not only the data but also the structure of the types that contain them [45]. 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 [27] or in aggregation relationships [7, 13] and has been adopted to some degree in commercial OODB systems [35, 36]. While most OODB systems to date provide some basic evolution support [48, 35, 4], the OODB community at large does not treat the relationship construct as a first-class entity. 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 manipulate and change any the object model construct. To achieve this goal a new set of schema evolution primitives to evolve the new construct(s) is required. The OODB system must also maintain the structural and referential integrity of the database. That is, the existing schema evolution primitives must be changed to assure that they now obey the new constraints of the new object model augmented with the relationship construct. For the relationship construct [13] 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 system 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 literal attributes. And while at the object level most OODB systems perform referential integrity checks to ensure referential consistency, at the schema level, referential relationships are largely mistreated 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 often 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 [32] to 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 a-priori 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 [17] provides an extensible and flexible schema evolution framework for supporting complex relationship evolution.

However, atomicity has a price and while a roll-back based support for consistency might be adequate for some types of application programs, it is a very expensive management strategy for schema evolution programs. Consider a large schema evolution program which may take over 24 hours [22]. A roll-back due to the detection of an erroneous condition in the 23rd hour is not an attractive or a viable option. It would be much preferable to have a guarantee of success *before* executing the evolution program. An alternative approach would thus be to allow for verification of schema evolution programs prior to their execution. We present here this alternative approach, a theorem prover to verify the correctness of an operation prior to its execution.

Contributions. 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. In [16], we made the following contributions:

- We identify the problem that current OODB systems 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;
- We propose a minimal set of *new evolution primitives* needed for supporting the evolution of both uni-directional and bi-directional relationships;
- We propose a set of *compound evolution primitives* for relationships;

- 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
- 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.

In this paper, we further extend this work and show now, how the contracts can indeed be utilized to verify the correctness of a ROVER Wrapper. We thus present here an approach for verification of a ROVER Wrapper via a theorem prover.

2 Background

In this section, we briefly introduce relationship constructs as defined by the Object Database Management Group (ODMG), while full details can be found in [13]. 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 querying the system dictionary as used in the paper.

Table 2 lists the basic schema evolution operations that are supported by most OODBs [35, 48].

3 Minimal Primitives for Relationship Evolution

In this section we present a set of evolution primitives needed for the evolving uni-directional (unary) and 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 [21]. They can be composed together with other evolution primitives to form more complex transformations, as we will discuss in Section 5.

Term	Description
$types(\mathcal{C})$	The set of all types in the system
s, t, T, \perp	Elements of $types(\mathcal{C})$
$\sigma(c)$	Type of class c
$super(t)$	The set of all direct supertypes of type t
$super^*(t)$	The set of all direct and indirect supertypes of type t
$sub(t)$	The set of all direct subtypes of type t
$sub^*(t)$	The set of all direct and indirect subtypes of type t
$in-paths(t)$	The set of all paths $\langle c,r \rangle$ referring to type t
$in-degree(t)$	The count of all paths referring to type t
$out-paths(t)$	The set of all paths $\langle t,r \rangle$ going out of type t
$obj-in-degree(o_i)$	The number of objects referring to the object o_i

Table 1: Axiomatic Notation of Schema Changes

Evolution Primitive	Description	Error Condition
$add-class(c, \mathcal{C})$	Add new class c to \mathcal{C} in the schema \mathbf{S}	Class c already exists in \mathbf{S}
$delete-class(c)$	Delete class c from \mathcal{C} in the schema \mathbf{S}	c has sub-classes
$add-ISA-edge(c_x, c_y)$	Add an inheritance edge from class c_x to c_y	there exists an edge from some class c_z to c_y
$delete-ISA-edge(c_x, c_y)$	Delete the inheritance edge from class c_x to c_y	c_x is root class
$add-attribute(c_x, a_x, t, d)$	Add attribute a_x of type t and default value d to class c_x and to all its subclasses	a_x already exists in c_x
$delete-attribute(c_x, a_x)$	Delete the attribute a_x from the class c_x and removes it from all its subclasses	a_x does not exist in class c_x

Table 2: Taxonomy of Basic Schema Evolution Primitives.

3.1 Evolution of Unary Relationships

As per the ODMG object model and other object models [48], unary relationships are modeled using a reference attribute. For example, Figure 2 shows a 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 a unary relationship.

Add-reference-attribute. 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 reference 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

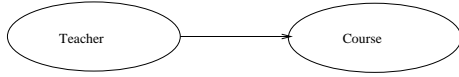


Figure 1: Graphical Schema Description of the Unary Relationship between Classes `Teacher` and `Course`.

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

Figure 2: ODMG Syntax for a Unary Relationship, `teaches`.

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.

Delete-reference-attribute. The `delete-reference-attribute` primitive deletes 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 bi-directional relationship implies special update semantics, i.e., any update of objects in one class `Teacher` will be automatically reflected in the other 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 set) for the source relationship *source-type* (`set<Course>`), and type of storage for the inverse relationship *inverse-type* (`Teacher`).

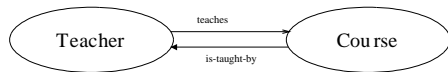


Figure 3: Graphical Schema Description of the `teaches` – `is-taught-by` Bi-directional Relationship.

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

Figure 4: ODMG Syntax for Specifying a Bi-Directional Relationship.

As a bi-directional relationship can be divided into a pair of uni-directional relationships between the two classes, only two additional primitives are needed for manipulating bi-directional relationships: *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 the two uni-directional relationships `Teacher.teaches` and `Course.is-taught-by` are to be modeled as a bi-directional relationship.

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*) decomposes the binary relationship into 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 affect 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
<i>add-reference-attribute</i> (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
<i>delete-reference-attribute</i> (c_x, r_x)	Delete unary relationship in class c_x named r_x
<i>form-relationship</i> (c_x, r_x, c_y, r_y)	Promote the specified two unary relationships to a binary relationship
<i>drop-relationship</i> (c_x, r_x, c_y, r_y)	Demote 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 primitives alone are not sufficient to manipulate all the properties of a relationship. For instance, using only the primitives in Section 3 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 [10] and Lerner [29] we now support atomic system-defined evolution operations that are composed of several (basic) evolution primitives.

```

change-cardinality-m1(cx, rx, td, inverse-cx, inverse-rx) {
  // Drop the relationship to two individual uni-directional relationships
  drop-relationship(cx, rx, inverse-cx, inverse-rx)

  // Create a new attribute in class cx. The attribute
  // tmp-Attr is a system generated name for a temporary attribute
  add-attribute(cx, tmp-attr, td, null);

  // As we are moving a from a multi-valued relationship to a
  // single-valued relationship, we need to provide this conversion
  // for each object in the extent. We use a default here, i.e. find
  // the most-common-value of the set rx and use that as a
  // representative value for the tmp-attr. Other, user-defined
  // functions can be used instead.
  while ((o = extent(cx).nextElement()) != null)
    o.tmp-attr = most-common(o.rx);

  // Remove the attribute rx from class cx
  delete-attribute(cx, rx);

  // Rename the attribute tmp-attr to rx
  rename-attribute(cx, tmp-attr, rx);

  // Form bi-directional relationship from two uni-directional relationships
  form-relationship(cx, rx, inverse-cx, inverse-rx)
}

```

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

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-attribute** adds a temporary attribute **tmp-attr** whose domain is representative of the cardinality change. Let the domain of **tmp-attr** be **t_d**. In the next step, for all objects in the extent of the class **c_x**, we need to convert the set value (**r_x**) to a singleton. We use a system-defined method **most-common(r_x)** that finds the most commonly occurring object **a_i** in a given collection **r_x**. Thus, for all objects in the extent of **c_x**, the single-value attribute **tmp-attr** is assigned **a**, the most common occurring value in the object's collection **r_x**. We then delete the attribute **r_x** and rename the temporary attribute **tmp-attr** to **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.

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

Table 4: Taxonomy of Compound Evolution Operations for Relationships.

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 semantics for a compound change, **change-cardinality-m1** shown in Figure 5. We arbitrarily use the most commonly occurring object value as the single-value representative. However, other conversions 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 user¹ any flexibility. Consider that the user now wishes to build a bi-directional relationship between two types where only one uni-directional relationship exists². The only alternative would be to write an ad-hoc program that combines existing primitives without the atomicity 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 while being guaranteed the atomicity of the operation and the consistency of the database.

5.1 SERF: A Framework for Extensible Schema Evolution

Our solution [17] is based on the hypothesis that complex schema evolution transformations can be broken down into a sequence of basic evolution primitives glued together by a language that can do complex compositions. We choose here a declarative language, i.e., a query language such as OQL [13].

Users can now on-the-fly introduce new transformations 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

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

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

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 them to a bi-directional relationship. This operation is defined by the user within the confines of our framework that can guarantee the atomicity and consistency of the system.

```
// 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);
```

Figure 6: Transformation From Uni-directional to Bi-directional Relationship

SERF Template. An OQL transformation as in Figure 6 *flexibly* allows a user to define different schema transformations. 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 [17]. A template uses the query language’s ability to query over the meta information (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 the meta information 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.

```

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

```

Figure 7: Template for Converting a Uni-directional Relationship to a Bi-directional Relationship

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 [18]. 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 ROVER Wrapper is a template with contracts. 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.

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 **preconditions** and **postconditions**.

The constraints, termed *preconditions*, are placed prior to any body of template code (OQL

statement including system-defined schema evolution primitive). The *preconditions* are separated from the actual OQL statements by means of the keyword **requires**. *Postconditions*, 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 postconditions 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 resulting schema and database was consistent, i.e., **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_s class is not referred to by another class. Moreover, no objects in the database must refer to the objects of the C_s 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 preconditions³. Thus, in this model it is easy to extend or modify the constraints without re-writing the code for the evolution primitive. Figure 9 shows the constraints that must be satisfied after the execution of the template code.

```

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

  template body here
}

```

Figure 8: Preconditions for Delete-Class Primitive in Contractual Form

```

delete-class (  $C_s$  )
{
  template body here

  ensures:
     $C_s \notin \mathcal{C} \wedge$ 
     $\sigma(C_s) \notin \mathbf{types}(\mathcal{C}) \wedge$ 
     $\forall \langle C_x, r_x \rangle \in out-paths(C_s)$ 
       $(\langle C_s \rangle \notin in-paths(C_x)) \wedge$ 
     $\forall C_x \in super(C_s)$ 
       $(C_s \notin sub(C_x) )$ 
}

```

Figure 9: Postconditions for the Delete-Class Primitive Template

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

7 Putting Contracts to Use

ROVER Wrappers provide faster system upgrade 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. With the contracts in place, we can deploy different approaches to verify the correct execution of a ROVER Wrapper. Perhaps, the most straight forward approach is that of checking the preconditions prior and postconditions after the execution of the ROVER Wrapper. If the preconditions are initially satisfied and the postconditions are met after the execution of the template, the ROVER Wrapper is said to be correct. As an example, consider the *inline* example given in Figure 10. The inline template inlines all the attributes (and hence all of the data) of the class referred to by the reference attribute `refClass` into the class `className`. Once all attributes and data have been moved over to `className`, the `refClass` is deleted. Now consider the scenario that the reference attribute is a self-referencing attribute⁴. In such a case after the execution of the `delete-class` primitive, the class `refClass = className` is deleted which is clearly not the desired result as indicated by the postconditions that require that `className` must still exist on completion of ROVER Wrapper execution. All changes made by the *inline* template must be rolled back in such a scenario.

Thus while this approach verifies the correctness of the ROVER Wrapper, its fall-back for a erroneous change is a transaction roll-back which in this case would be a rather expensive operation and hence not very desirable. An alternative to this is to use theorem proving to verify the correctness of the ROVER Wrappers prior to their execution. A theorem prover based on facts postulates the different states of the system for every statement that is executed. Consider that the system is in state S_0 of the system in which all preconditions given in Figure 11 for the *inline* wrapper are met. The theorem prover can now calculate the new state S_1 which results from the addition of an attribute. Let state S_n represent the state of the system after all attributes have been added. In state S_n , a `delete-class` primitive would take the system to its final state S_f . If the state S_f does not meet the desired postconditions as stated in the wrapper, then the wrapper is not allowed to execute. In the following sections, we detail the basic requirements for a theorem prover and show via an example (*inline* template) how the postconditions of the wrapper can be checked prior to execution.

⁴Inline assumes only one level of inlining, so recursion is not an issue here.

```

begin template inline ( Class: className, Attribute: refAttrName)
{
  refClass = element (
    select a.attrType
    from MetaAttribute a
    where a.attrName = $refAttrName
    and a.classDefinedIn = $className; )

  define localAttrs(cName) as
    select c.localAttrList
    from MetaClass c
    where c.metaClassName = cName;

  // get all attributes in refAttrName and add to className
  for all attrs in localAttrs(refClass)
    add_atomic_attribute ($className, attrs.attrName,
      attrs.attrType, attrs.attrValue);

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

  // set: className.Attr = className.refAttrName.Attr
  for all obj in extents($className):
    for all Attr in localAttrs(refClass)
      obj.set (obj.Attr, valueOf(obj.refAttrName.Attr));

  delete_attribute ($className, $refAttrName);
}
end template

```

Figure 10: The Inline Template.

```

begin template inline ( Cs, rs )
{
  requires:
    Cs ∈ C ∧
    σ(Cs) ∈ types(C) ∧
    rs ∈ N(Cs) ∧
    domain(rs) ∈ C ∧
    Cs ≠ domain(rs);

  Body of inline template

  ensures:
    Cs ∈ C ∧
    σ(Cs) ∈ types(C) ∧
    rs ∉ N(Cs) ∧
    domain(rs) ∉ C ∧
    ∀ ax ∈ N(domain(rs))
      (ax ∈ Cs);
}

```

Figure 11: Inline ROVER Wrapper with Set-Theoretic Contracts

7.1 Theorem Prover for SERF Templates

Verification of any program relies on the knowledge that given a start state, the program code will take the system to a desired final state. Thus, before verification can be applied, it becomes essential to describe these states, the start and the final states. In this section, we now first define these states and then walk-through an example to show how the technique can be used to verify a ROVER Wrapper.

Theorem proving approaches verification by formalizing (1) a model of computation, (2) the specification and (3) the rules of inference [8]. The axioms and other knowledge about the environment comprise the model of computation, the pre- and post-conditions are the specification, i.e., the initial state of the system as well as the final targeted state that must be reached. The rules of inference are the functions that help reason about the validity of the path from the initial to the final states of the system. Table 5 shows what these components correspond to in the context of SERF. In the following subsections, we sketch out these three components for our problem domain to show the viability of theorem provers for SERF templates.

Theorem Prover Component	SERF Components
Model of Computation	Object Model, Invariants, System Functions
Specification	ROVER Wrappers (Pre-Post Conditions)
Rules of Inference	Schema Evolution Primitives

Table 5: Theorem Prover Components for the SERF Environment.

7.1.1 Model of Computation for ROVER Wrappers

The model of computation formally describes the environment in which the theorem prover is being applied. The SERF framework is based on the ODMG object model (Section 2) [13]. Hence, the theorem prover must be provided with a formal definition of the ODMG object model, its invariants and the functionality of each of the system dictionary functions as described in Table 1. This model of computation is part of the setup of the theorem prover system and thus would be created once a-priori for the SERF system. It would only need to be modified if and when there is a change in the environment itself, for example if the object model changes. While different theorem provers use different languages [24, 38], for the purpose of this paper, we assume the language of the theorem prover to be set-theoretic.

The Object Model. Table 6 gives a brief description of the components of the object model. In general, a schema that ties all this together is as defined below.

Term	Description
\mathcal{C}	The set of all class names in the system
$\mathbf{types}(\mathcal{C})$	The set of all types in the system
σ	Mapping from \mathcal{C} to $\mathbf{types}(\mathcal{C})$
\prec	The sub-typing relationship on $\mathbf{types}(\mathcal{C})$
\mathcal{R}	The set of all relationships in the system
\mathbf{obj}	The set of all objects in the system
O	An object that is a pair (o,v) where o = OID and v = value
α	Mapping from \mathcal{R} to an ordered pair of types in $\mathbf{types}(\mathcal{C})$
\mathbf{M}	The set of all method signatures in the system

Table 6: Components of the Object Model

Definition 1 A schema is a tuple $\mathcal{S} = (\mathbf{C}, \sigma, \prec, \mathbf{M}, \mathbf{G}, \mathcal{R}, \alpha)$ where

- \mathbf{G} is a set of class names disjoint from \mathbf{C} ,
- σ is a mapping from $\mathbf{C} \cup \mathbf{G}$ to $\mathbf{types}(\mathbf{C})$,
- $(\mathbf{C}, \sigma, \prec)$ is a well-formed class hierarchy,

- \mathbf{M} is a well-formed set of method signatures for $(\mathbf{C}, \sigma, \prec)$,
- \mathcal{R} is a finite set of relation names, and
- α is a mapping from \mathcal{R} to an ordered pair of types.

A more thorough treatment of the formal description of the object model can be found in [1].

System Functions. Table 1 describes some helper functions which are a part of the system definition. For the theorem prover, the behavior of each of these functions must be precisely defined in a set language.

Invariants of the Object Model.

1. **Rootedness.** There is a single type t in \mathcal{T} that is the super-type of all types in \mathcal{T} . The type t is called the *root* of the type lattice.
2. **Closure.** Every type in \mathcal{T} , excluding *root*, has a super-type in \mathcal{T} , giving closure to \mathcal{T} .
3. **Pointedness.** There are one or more types \perp in \mathcal{T} such that \perp has no subtypes in \mathcal{T} . \perp is termed a *leaf* of the type lattice.
4. **Distinction.** Every type t in \mathcal{T} has a distinct name. Every property \mathbf{p} for a type \mathbf{t} has a distinct name. The scope of name distinction for a property is the union of the inherited ($\mathbf{H}(\mathbf{t})$) and native properties ($\mathbf{N}(\mathbf{t})$) for a type.
5. **Singularity.** Every type t in \mathcal{T} has at most one direct super-type \mathbf{t}_s in \mathcal{T} , i.e., $|P(t)| = 1$.

7.1.2 Specification of ROVER Wrappers

A theorem prover requires the specification of the initial state of the system as well as the final state that needs to be verified. The contracts, i.e., the pre- and post- conditions as defined in Section 6, fulfill these requirements by providing an initial state (the pre-conditions) that must be valid and an expected final state (the post-conditions) that must be met. However, the theorem prover expects its inputs to be expressed in a formal language. Hence these contracts need to be converted to the language of the theorem prover, i.e., in our case to a set-theoretic language.

7.1.3 Rules of Inference

The rules of inference are operations that move the system from one given state to another state, i.e., code segments that take the system from an initial specification to a final specification. The

body of a ROVER Wrapper, i.e., the actual schema evolution functions and OQL code, are hence the rules of inference in our system.

For example, Figure 10 depicts a SERF template that inlines the class `refClass` referred to by the reference attribute `refAttrName` in class `className` into the class `className`. Each step (statement, OQL, or schema evolution function) of the inline template as shown in Figure 10 is a rule of inference. Each of these rules is applied one at a time to a given state of the system.

In addition to the primitive evolution programs, we also consider the `for all` OQL statement. This is translated to a repetitive application of the loop body that results in a cumulative effect on the state of the system. For example, `for all x in attributeSet: add-attribute(C, x, default)` results in the application of the `add-attribute` primitive `count(attributeList)` times, where `count` gives the number of elements in a set. The final state of the system will be the cumulative result of applying all `add-attribute` primitives.

7.2 Formal Verification Process: Application to Inline Template

In this section, we illustrate the working of the theorem prover by a step by step verification of a template (namely the `inline` template from Figure 11), thereby showing how theorem provers can be applied to our domain for verification of schema evolution transformations. This is an automated process that assumes the computation model (Section 7.1.1) and the rules of inference (Section 7.1.3) have already been provided as part of the tool. The user only needs to input the specification contracts and the template code in OQL.

Consider the ROVER Wrapper shown in Figure 11. The class C_s and the reference attribute r_s must exist otherwise it is meaningless to proceed with the verification. Given this initial state S_0 , we proceed to apply the first evolution change in the template, `add-attribute` (statement 1). This results in a new state S_1 . As part of our work, we wrap each individual change primitive program using a ROVER Wrapper with their specific contracts. Henceforth we consider the ROVER Wrapper for each evolution primitive program. Thus, in this example the initial state of the system must meet the precondition for `add-attribute`. The post-conditions specified by `add-attribute` represent the final state S_1 and are indicative of the behavior of the `add-attribute` primitive. Figure 12 lists the ROVER Wrapper for `add-attribute`. Figure 8 and 9 list the ROVER Wrapper precondition and postcondition for the `delete-class` primitive which are used in the example here.

A subsequent evolution change, `add-attribute`, uses this state S_1 as its initial state against which its precondition must match. Tables 2 and 3 contain the schema evolution operations that we consider as rules of inference in our system. The theorem prover proves the correctness of the inline transformation shown in Figure 10 by first proving three theorems where each theorem is similar to the `add-attribute` and then composes them together to prove the correctness of the

```

add-attribute (  $C_s, a_x, t, \text{default}$  )
{
  requires:
     $C_s \in \mathcal{C} \wedge$ 
     $\sigma(C_s) \in \mathbf{types}(\mathcal{C}) \wedge$ 
     $a_x \notin N(C_s)$ 

  add-attribute-primitive (  $C_s, a_x, t, \text{default}$  )
}

```

```

{
  ensures:
     $a_x \in N(C_s) \wedge$ 
     $\forall \mathbf{y} \in \text{sub}^*(C_s)$ 
       $a_x \in H(\mathbf{y})$ 
}

```

Figure 12: Add-Attribute Primitive Template with Contracts

inline transformation itself (the fourth theorem). Each of the theorems specifies the properties of one of the evolution programs in the inline transformation.

Schema Evolution Primitive: add-attribute. The precondition from the contract specification in Figure 11 that must hold for **add-attribute**($C_s, a_x, \text{type}, \text{default}$) is given in Equation 1⁵:

$$\left. \begin{array}{l} C_s \in \mathcal{C} \\ \sigma(C_s) \in \mathbf{types}(\mathcal{C}) \\ a_x \notin N(C_s) \end{array} \right\} \wedge \wedge \quad (1)$$

The desired postconditions expected after applying **add-attribute** are:

$$\left. \begin{array}{l} C_s \in \mathcal{C} \\ \sigma(C_s) \in \mathbf{types}(\mathcal{C}) \\ a_x \in N(C_s) \\ \forall \mathbf{y} \in \text{sub}^*(C_s) \\ a_x \in H(\mathbf{y}) \end{array} \right\} \wedge \wedge \quad (2)$$

Theorem 1 *If the add-attribute program is applied to arguments satisfying the precondition given in Equation 1, then the program results satisfy the postcondition given in Equation 2.*

Proof: Assume that the precondition in Equation 1 holds and **add-attribute** adds the attribute a_x to the class C_s , i.e.:

$$N(C_s) = a_x \cup N(C_s) \quad (3)$$

The primitive **add-attribute** also adds a_x to all the subclasses $\text{sub}^*(C_s)$ of C_s (refer Table 2). Hence we have:

$$H(y) = a_x \cup H(\mathbf{y}) \quad (4)$$

⁵These are repeated from Figure 11 for convenience.

Here Equations 3 and 4 show the altered states of the system after the execution of each **add-attribute** function. From Equations 3 and 4, we have the desired postcondition as specified in Equation 2. \square In Figure 10, the **for-all** loop copies all the attributes of the class $C_{refClass}$ ⁶. At the end of the **for-all** loop, with repetitive application of **add-attribute**, the desired state is given by the postcondition in Equation 5.

$$\left. \begin{array}{l} C_s \in \mathcal{C} \\ \sigma(C_s) \in \mathbf{types}(\mathcal{C}) \\ \forall a_k \in N(C_{refClass}) \\ (x \in N(C_s)) \\ \forall y \in sub^*(C_s) \\ (a_k \in H(\mathbf{y})) \end{array} \right\} \begin{array}{l} \wedge \\ \wedge \\ \wedge \\ \wedge \\ \wedge \end{array} \quad (5)$$

Theorem 2 *If the **add-attribute** function is correct as per Theorem 1, then repetitive execution of **add-attribute** for all $\mathbf{a}_k \in C_{refClass}$ results in a cumulative effect such that postcondition given in Equation 5 are satisfied.*

Proof: (Proof By Induction)

Base Case: Assume that the class $C_{refClass}$ has only one attribute a . This reduces the **for all** statement to a simple **add-attribute**($C_s, a, a.attrType, a.defaultValue$). We know by Theorem 1 that if the pre-condition given in Equation 1 holds for these arguments, then the postcondition as given in Equation 2 will also hold, i.e., for one attribute ($n = 1$), postcondition in Equation 5 reduce to postcondition in Equation 2.

Induction Hypothesis: Assume that the theorem holds true when class $C_{refClass}$ has k attributes and they are added to class C_s , i.e., the postcondition given in Equation 5 is satisfied for k **add-attribute** applications.

Induction: Prove that the post-condition in Equation 5 holds when the class $C_{refClass}$ has $k+1$ attributes.

We know that the postcondition (5) holds when class $C_{refClass}$ has k attributes and they are added to the class C_s . Assume we now have attribute \mathbf{a}_{k+1} , the $k + 1^{th}$ attribute, that is unique, i.e., $\mathbf{a}_{k+1} \neq \mathbf{a}_j$, for $1 \leq j \leq k$. To add this to class C_s we do: **add-attribute**($C_s, \mathbf{a}_{k+1}, \mathbf{a}_{k+1}.attrType, \mathbf{a}_{k+1}.defaultValue$). We know by Theorem 1 that if this satisfies the precondition given in Equation 1, then the postcondition in Equation 2 holds true (Base Case). Combining the postcondition for the addition of k attributes (Induction Hypothesis) with the postcondition of the Base Case, we get the postcondition as given in Equation 5. \square

⁶ $C_{refClass}$ is the class that is being referred to by the reference attribute r_x in class C_s .

Schema Evolution Primitive delete-attribute. First we give the initial state of the system, i.e., the precondition in Equation 6 that must be satisfied for the primitive `delete-attribute`(C_s , a_x):

$$\left. \begin{array}{l} C_s \in \mathcal{C} \\ \sigma(C_s) \in \mathbf{types}(\mathcal{C}) \\ a_x \in N(C_s) \end{array} \right\} \wedge \wedge \quad (6)$$

After the execution of `delete-attribute`, the final state of the system is as given in the postcondition in Equation 7.

$$\left. \begin{array}{l} a_x \notin N(C_s) \\ (\forall \mathbf{y} \in \mathit{sub}^*(C_s) \\ a_x \notin H(\mathbf{y})) \end{array} \right\} \wedge \quad (7)$$

Theorem 3 *If `delete-attribute` is applied to arguments satisfying precondition (6), then the result satisfies the postcondition (7).*

Proof: Similar to Theorem 1.

Schema Evolution Primitive delete-class. The necessary precondition that must hold for `delete-class`($C_{refClass}$) is given in Equation 8:

$$\left. \begin{array}{l} C_{refClass} \in \mathcal{C} \\ \sigma(C_{refClass}) \in \mathbf{types}(\mathcal{C}) \\ \mathit{sub}(C_{refClass}) = 0 \\ \mathit{in-degree}(C_{refClass}) = 0 \\ (\forall o_i \in \mathbf{extent}(C_{refClass}) : (\mathit{obj-in-degree}(o_i) = 0)) \end{array} \right\} \wedge \wedge \quad (8)$$

The desired postcondition (9) after the application of `delete-class` is as given in Equation 9.

$$\left. \begin{array}{l} \forall \langle C_x, r_x \rangle \in \mathit{out-paths}(C_{refClass}) \\ (\langle C_{refClass} \rangle \notin \mathit{in-paths}(C_x)) \wedge \\ \forall C_x \in \mathit{super}(C_{refClass}) \\ (C_{refClass} \notin \mathit{sub}(C_x)) \wedge \\ C_{refClass} \notin \mathcal{C} \\ \sigma(C_{refClass}) \notin \mathbf{types}(\mathcal{C}) \end{array} \right\} \quad (9)$$

Theorem 4 *If `delete-class` is applied to arguments satisfying the precondition in Equation 8, then the result satisfies the postcondition in Equation 9.*

Proof: Similar to Theorem 1.

The Inline Transformation. To verify the correctness of the inline transformation, we chain the results of Theorems 1, 2, 3 and 4. The overall precondition for this is given by Equation 10⁷. The execution of the inline transformation must result in the final state as specified by Equation 11⁸.

$$\left. \begin{array}{l} C_s \in \mathcal{C} \quad \wedge \\ a_x \in N(C_s) \quad \wedge \\ C_{refClass} = domain(a_x) \quad \wedge \\ C_{refClass} \in \mathcal{C} \quad \wedge \\ C_s \neq C_{refClass} \end{array} \right\} \quad (10)$$

$$\left. \begin{array}{l} C_s \in \mathcal{C} \quad \wedge \\ C_{refClass} \notin \mathcal{C} \quad \wedge \\ \forall a_k \in NC_{refClass} \\ (a_k \in N(C_s)) \quad \wedge \\ \forall \mathbf{y} \in sub^*(C_s) \\ (a_k \in H(\mathbf{y})) \quad \wedge \\ a_k \notin N(C_s) \end{array} \right\} \quad (11)$$

Theorem 5 *If Theorems 1, 2, 3, and 4 are satisfied in the order specified, then the inline transformation satisfies the postcondition given in Equation 11.*

Proof: The proof for this can be given by a combination of the postcondition in Equations 5, 7 and 9. □

Using theorem proving techniques as shown for the template here it is possible to verify the correctness of any given template. If at any point one of the sub-theorems is not satisfied, i.e., if Theorems 1, 2, 3, or 4 are not satisfied, the verification process is aborted and the template is not permitted to be executed. However, all of this occurs prior to any execution of the ROVER Wrapper. Hence, any failure of the verification process results in no execution of the ROVER Wrapper. While the verification process may incur additional overhead of checking, we believe it can be shown that under erroneous conditions the verification process can save hours when compared to a roll-back strategy.

8 Related Work

Relationships. Semantic modeling research has looked into the modeling of relationships and the different semantics that can be applied for these relationships [9]. In object databases, Kim, Bertino and others [26] have examined the part-whole relationship (composite objects). Bertino et al. [7]

⁷This is the precondition for the `inline` wrapper in Figure 11.

⁸This is the postcondition for the `inline` wrapper given in Figure 11.

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.

Schema Evolution. Schema evolution is a problem that is faced by long-lived data. The goal of schema evolution research is to allow schema evolution mechanisms to change not only the schema but also the underlying objects to have them conform to the modified schema. Many researchers have looked at the problem of schema evolution and at providing basic primitives to handle these schema manipulations [4, 46, 37, 45]. Most commercial systems today such as Itasca [25], GemStone [11], ObjectStore [35], and O₂ [48] provide similar evolution support for their underlying object model.

In recent years, the advent of more advanced applications has led to the need for support of complex schema evolution operations. [44, 23, 10, 29, 14] have investigated the issue of more complex operations. Much of the initial work [44, 23] focused on converting and re-structuring data from flat files to a database which often required complex conversions. More recently, [30] has introduced compound type changes in a software environment, i.e., focusing only on the type changes and not the changes to the object instances associated with the modified type. The proposed compound type changes included *Inline*, *Encapsulate*, *Merge*, *Move*, *Duplicate*, *Reverse Link* and *Link Addition*. Breche [10] proposed a similar list of complex evolution operations for O₂, i.e., which considered schema as well as object changes. [10] shows that these advanced primitives can be formulated by composing the basic primitives that are provided by the O₂ system, and has shown the consistency of these advanced primitives. Lastly, while our previous work on SERF [17] 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, ROVER addresses the problem of evolution (be it simple or complex) in the context of object models with relationships. We extend previous work in two ways - we now provide evolution of relationships in the object model and we now also provide the users with a framework that allows extensibility to schema evolution in general.

Peters and Ozsu [40] have introduced an axiomatic model that can be used to formalize and compare schema evolution modules of OODBs. They develop a formal basis for schema evolution research, and we use their notation and model to represent the concepts in this paper.

Another important issue in evolution is to provide support for existing applications that depend on the old schema, when other applications change the shared schema according to their own requirements. Method to address problem include, views [43, 41, 6, 5] and versions [46, 28, 33]. However, such work has focused on simple schema evolution operations only, and hence may need to be re-examined in order to handle the complex notion of transformations as introduced by our templates. We do not address this issue in this work.

Consistency Management. 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 [19, 34]. 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.

Other approaches for consistency management in ODBs utilize programming language support [49, 2] such as assertions and exception handling mechanisms in languages like C++, Java and Ada. Relational database systems (RDBMS) offer some additional support in the form of *triggers* but provide only roll-back semantics, i.e., if a constraint is not satisfied at the end of a transaction, then the entire transaction is rolled back [20]. Active database systems [12, 31, 3] provide event-condition-action (ECA) rules as a mechanism for detecting the occurrence of some event and responding to it by some action.

Research has also been done in consistency management for software process languages. Tarr et al. [47] have developed a consistency management system which allows for the specification of consistency conditions and the degree of inconsistency tolerable by the user. Much of the work in literature concentrates on detecting consistency violations and on the specification of consistency constraints. We now try to utilize theorem provers as a preventive measure to ensure that consistency violations do not occur.

9 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 [18]. The core SERF system for flexible transformation was demonstrated at SIGMOD 2000 [42]. Our approach as we describe in this work is focused on allowing consistent, yet flexible mechanism for schema evolution of relationships. However, our approach, and hence the consistency checking mechanism is tied to the ODMG model. As part of the future work, we are now looking at developing a meta-model, graph based approach which can further generalize these concepts and make them model-independent [15]. Also, more detailed investigation of the theorem prover approach is required to see if it is really a viable and a general approach for various types of constraint checks that may be required.

References

- [1] S. Abiteboul, R. Hull, and Vianu V. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.
- [2] T. Andrews and C. Harris. Combining Language and Database Advances in an Object-Oriented Development Environment. In *Readings in Object-Oriented Database Systems*, pages 186–196, 1990.
- [3] F. Bancilhon and W. Kim. Object-Oriented Database Systems: In Transition. In *SIGMOD RECORD*, volume 19, December 90.
- [4] 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.
- [5] Z. Bellahsene. A View Mechanism for Schema Evolution. In *British National Conference on Databases*, pages 18–35, 1996.
- [6] E. Bertino. A View Mechanism for Object-Oriented Databases. In *3rd Int. Conference on Extending Database Technology*, pages 136–151, March 1992.
- [7] E. Bertino and G. Guerrini. Extending the ODMG Object Model with Composite Objects. In *OOPSLA*, pages 259–270, 1998.
- [8] P.J. Black, K.M. Hall, M.D. Jones, T.N. Larson, and P.J. Windley. A Brief Introduction to Formal Methods. In *Proceedings of CICC*, pages 377–380, 1996.
- [9] G. Booch. *Object-Oriented Analysis and Design*. Benjamin Cummings Pub., 1994.
- [10] P. Bréche. Advanced Primitives for Changing Schemas of Object Databases. In *Conference on Advanced Information Systems Engineering*, pages 476–495, 1996.
- [11] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Management System. In *Object-Oriented Concepts, Databases and Applications*, pages 283–308. ACM Press, 1989.
- [12] A. Buchmann, R. Carrera, and M. Vazquez-Galindo. A Generalized Constraint and Exception Handler for Object-Oriented CAD-DBMS. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 38–49, 1986.
- [13] R.G.G Cattell and et al. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Inc., 1997.
- [14] S.M. Clamen. Type Evolution and Instance Adaptation. Technical Report CMU-CS-92-133R, Carnegie Mellon University, School of Computer Science, 1992.
- [15] K. T. Claypool and E. A. Rundensteiner. **Sangam**: Modeling Transformations For Integrating Now and Tomorrow. Technical Report WPI-CS-TR-01-04, Worcester Polytechnic Institute, March 2001.
- [16] K. T. Claypool, E. A. Rundensteiner, and G.T. Heineman. ROVER: A Framework for the Evolution of Relationships. In *Nineteenth International Conference on Conceptual Modeling*, October 2000.
- [17] 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.
- [18] 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.
- [19] 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.
- [20] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, Inc., 1996.
- [21] F. Ferrandina, G. Ferran, T. Meyer, J. Madec, and R. Zicari. Schema and Database Evolution in the O₂ Object Database System. In *Int. Conference on Very Large Data Bases*, 1995.
- [22] F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. In *VLDB*, pages 261–272, 1994.
- [23] J.P. Fry, R.L. Frank, and E.A. Hershey III. A Development Model for Data Translation. In A. L. Dean., editor, *Proceedings of 1972 ACM-SIGFIDET Workshop on Data Description, Access and Control, Denver, Colorado, November 29 - December 1, 1972*, pages 77–105. ACM, 1972.
- [24] M.J.C Gordon and T.F. Melham. Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, 1993.
- [25] Itasca Systems Inc. Itasca Systems Technical Report. Technical Report TM-92-001, OODBMS Feature Checklist. Rev 1.1, Itasca Systems, Inc., December 1993.

- [26] W. Kim, E. Bertino, and J. F. Garza. Composite objects revisited. *SIGMOD*, pages 337–347, 1989.
- [27] 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.
- [28] S.-E. Lautemann. Schema Versions in Object-Oriented Database Systems. In *Int. Conference on Database Systems for Advanced Applications (DASFAA)*, pages 323–332, 1997.
- [29] B.S. Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. Technical Report UM-CS-96-044, University of Massachusetts, Amherst, 1996.
- [30] B.S. Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. *ACM Transactions on Database Systems*, 25(1):83–127, 2000.
- [31] G. Lohman, B. Lindsay, H. Pirahesh, and K.B. Schiefer. Extensions to StarBurst: Objects, Types and Rules. *Communications of the ACM*, 34(10):95–109, 1991.
- [32] B. Meyer. Applying “Design By Contract”. *IEEE Computer*, 25(10):20–32, 1992.
- [33] S. Monk and I. Sommerville. Schema Evolution in OODBs Using Class Versioning. In *SIGMOD RECORD, VOL. 22, NO.3*, September 1993.
- [34] 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.
- [35] Object Design Inc. *ObjectStore - User Guide: DML. ObjectStore Release 3.0 for UNIX Systems*. Object Design Inc., December 1993.
- [36] Objectivity Inc. White Paper, Schema Evolution in Objectivity, February 1994.
- [37] Erik Odberg. MultiPerspectives: the classification dimension of schema modification management for object-oriented databases. In *TOOLS USA '94 (Technology of Object-Oriented Languages and Systems)*, Santa Barbara, California, USA, August 1994.
- [38] S. Owre, J.M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *11th CADE, Lecture Notes in Artificial Intelligence*, pages 748–752, 1992.
- [39] 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.
- [40] R.J. Peters and M.T. Ozsu. Axiomatization of Dynamic Schema Evolution in Objectbases. In *IEEE Int. Conf. on Data Engineering*, pages 156–164, 1995.
- [41] Y. G. Ra and E. A. Rundensteiner. A Transparent Schema Evolution System Based on Object-Oriented View Technology. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):600–624, September 1997.
- [42] 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.
- [43] E.A. Rundensteiner, A. Lee, and Y.-G. Ra. Capacity-Augmenting Schema Changes on Object-Oriented Databases: Towards Increased Interoperability. In *Object-Oriented Information Systems*, 1998.
- [44] N.C. Shu, B.C. Housel, R.W. Taylor, S.P. Ghosh, and V.Y. Lum. EXPRESS: A Data EXtraction, Processing, and REStructuring System. *TODS*, 2(2):134–174, 1977.
- [45] D. Sjoberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1):35–54, January 1993.
- [46] A. H. Skarra and S. B. Zdonik. The Management of Changing Types in an Object-Oriented Databases. In *Proc. 1st OOPSLA*, pages 483–494, 1986.
- [47] P. Tarr and L. Clarke. Consistency management for complex applications. In *International Conference on Software Engineering*, pages 230–239, 1998.
- [48] O₂ Technology. *O₂ Reference Manual, Version 4.5, Release November 1994*. O₂ Technology, Versailles, France, November 1994.
- [49] S.L. Vandenberg and D.J. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity and Inheritance. In *SIGMOD*, pages 158–167, 1991.