WPI-CS-TR-99-15 March 1999

Extending Schema Evolution to Handle Object Models with Relationships

by

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

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Extending Schema Evolution to Handle Object Models with Relationships *

Kajal T. Claypool, Elke A. Rundensteiner and 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 for advanced applications. Today most emerging modeling standards such as the ODMG object model and UML hence have some support for relationships. Similarly, current OODB systems are begining to support structural definition of relationships as well as object level management. However, no work has been done on schema evolution of an object model that has relationships. In this paper, we present the evolution taxonomy required for evolving relationships. We also examine the effect of relationships on the pre-existing taxonomy of evolution primitives that is supported by current commercial OODB systems. We provide a comprehensive contract-based solution within the SERF framework to handle the problem externally, i.e., in a layer outside the OODB system. Also as part of our work, we extend SERF to now allow for flexible evolution of relationships with user-defined semantics. Lastly, we show that SERF is powerful in that it can provide extended schema evolution support in an OODB after its object model has been extended. In some cases, such as in *aggregate* relationships we can use SERF to provide the basic evolution support for the object model extension.

*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 Informix for software contribution. Special thanks also goes to the PSE Team specifically, Gordon Landis, Sam Haradhvala, Pat O'Brien and Breman Thuraising at Object Design Inc. for not only software contributions but also for providing us with a customized patch of the PSE Pro2.0 system that exposed schema-related APIs needed to develop our tool.

1 Introduction

Object modeling is a critical technology with much industry and research focused on it. Relationships are one class of modeling constructs that have been studied for a long period of time, both in the form of composite objects [KGBW90] and of associations between objects [Boo94]. Today, most OODB systems provide some implicit relationship support in the form of reference attributes as well as explicit relationship support [Obj93, Tec92, Obj94]. Work is also ongoing to provide more specific modeling constructs such as *aggregation* at the OODB system level [BG98]. All work thus far has primarily focused on providing support for the definition of relationships at the schema level [Cea97] and for managing them at the object level [BG98].

Schema Evolution of Relationships. At the same time, changing application requirements have made schema evolution a critical task [Sjo93]. Today, most OODB systems recognize this and provide some basic evolution support [Tec94, Tec92, BMO+89, Obj93, BKKK87, Inc93]. However, while some OODB systems provide object level management for relationships, evolution support for the same is lacking. To address this gap, in this paper we present a set of evolution primitives for relationships. However, we find that this support alone is not sufficient. We need to analyze and address the repercussions of adding relationships to an object model on the existing taxonomy of schema evolution primitives. We demonstrate that in current OODB systems, where relationships are often implemented as reference attributes, a myriad of problems related to the consistency of a system can arise. For example, a simple `delete-class` primitive may leave dangling references both at the schema level and the object level when deleting a class involved in relationships with other classes. Towards that end, in this paper we characterize evolution related problems that can occur when the object model is extended to support relationships. We also propose a comprehensive solution based on software contracts for maintaining consistency while providing evolution of relationships.

Flexible Schema Evolution of Relationships. Furthermore, providing basic schema evolution support for relationships does not necessarily cover all the changes that a user might potentially want to make to an object schema that contains relationships. It is hard, if not impossible, to provide a set of *pre-defined* changes that would meet every users need. In our previous work, we have proposed a framework, SERF, for allowing users to safely customize semantics of transformations in a *flexible* and *re-usable* manner [CJR98]. While our previous work on SERF was limited to a simple object model without relationships, we find that a large number of transformations involving relationships can still be defined using the current SERF system. However, we have also identified some interesting ones that cannot be addressed. For example, building a bi-directional relationship between two disjoint classes `Employee` and `Department` cannot be achieved by the current SERF technology. The reason for this is that it requires information specific to the instantiated classes, which it makes it hard to generalize it. Thus, in this work, we also address this problem and propose *user functions*, a named transformation that can be invoked from within a SERF template, as key step of our solution. In this paper we present the complete system extensions needed to support *user functions* in SERF.

Extending the Object Model Easily. The last part of this paper presents a major step forward, even beyond evolution support for relationship constructs. We investigate the feasibility of using the SERF framework as a mechanism to simplify the process of offering evolution support to an OODB as it is extended by additional semantic constructs, such as relationships, aggregation relationships, key constraints, etc. In general, when the object model in the underlying OODB system is extended, the OODB system must accomplish two tasks: (1) provide new schema evolution primitives for the extension and (2) update its existing taxonomy of schema evolution primitives. This is a tedious and extremely expensive process. In Section 8, we show that under some scenarios this costly endeavour can be circumvented by using SERF to provide the schema evolution for the extensions.

Contributions. In this work we bring together and extend techniques from several existing areas to develop a framework for the schema evolution of relationships. In summary in this paper:

- We identify the problem that OODBs which offer 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.
- We present a comprehensive solution, called *Contracts in SERF*, to the above problems that to the best of our knowledge represents the first evolution system handling relationships in OODB systems. As part of this solution we present here:
  - The minimal set of new evolution primitives needed for supporting the evolution of uni-directional and bi-directional relationships.
  - The analyses of the effect of relationships in the object model on the existing taxonomy of schema evolution primitives supported by most current OODB systems.
  - The development of a contract model for the specification of schema constraints and for describing the behavior of schema evolution primitives.
  - The integration of the contracts into SERF templates enabling us to provide pre-execution analysis as well as post-execution verification of the evolution primitives.
- Driven by our work on relationships we extend the power of the SERF system to effectively support flexible evolution of relationships in particular and more powerful transformations in general by supporting:
  - User functions that allow for user-specific transformations in a generalized template,
  - Embedded SERF templates and
  - Typing for SERF templates.
- We show SERF can be used to minimize development effort for upgrading a schema evolution system of an OODB system in the face of an extension of its underlying object model.

Overview of Rest of the Paper. Sections 2 and 3 gives the object model extended with relationships and introduces the readers to the SERF framework. In Section 4 we present the basic evolution of relationships that must be provided by an OODB system and in Section 5 we analyse the consistency problems at the evolution level that are introduced by extending the object model with relationships. Section 6 shows how we can use an extended SERF system to resolve the consistency problem. In Section 7 we present the extensions that are needed to now enable us to do relationship transformations beyond the basic primitives provided by the system. Section 8 goes beyond relationship support (basic and flexible) by demonstrating how SERF can be exploited to make it easy to extend an object model. Section 9 talks about some related work and Section 10 gives our conclusions and some possible extensions for the future.

2 Object Model with Relationships

In this section, we thus introduce a formal treatment of relationships as defined by ODMG [Cea97] using the formal model presented by Abiteboul et al. [AHV95]. 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.

In general, we define a class association list as a five tuple ($\mathcal{C}$, σ, $\mathcal{R}$, α), where $\mathcal{C}$ is a finite set of class names, σ is a mapping from $\mathcal{C}$ to **types**, $\mathcal{R}$ is a finite set of relation names, and α is a mapping from $\mathcal{R}$ to an ordered pair of types.

Definition 1 (Referential Relationship.) *Two elements* c_1, $c_2 \in \sigma(\mathcal{C})$ *are in* referential relationship *if* $\exists$ $r \in \mathcal{R} : \alpha(r) = <c_1,c_2>$. r *is the name of the referential relationship. The referential relationship is denoted as* $c_1 \overrightarrow{\ r\ } c_2$.

Definition 2 (Bi-directional Relationship.) *Two elements* c_1, $c_2 \in \sigma(\mathcal{C})$ *are in* bi-directional relationship *if* $\exists$ $r_1, r_2 \in \mathcal{R}$*:* $\alpha(r_1) = <c_1,c_2> \wedge \alpha(r_2) = <c_2,c_1>$. r_1 *and* r_2 *are the names of the relationships and are termed as a* pair. r_1 *is also termed as the* inverse-of r_2 *and vice versa. The bi-directional relationship is denoted as* $c_1 \overset{r_1/r_2}{\longleftrightarrow} c_2$.

2.1 Invariants for the ODMG Object Model

A schema update can cause inconsistencies in the structure of the schema, referred to as structural inconsistency. An important property imposed on schema operations is thus that their application always results in a *consistent* new schema [BKKK87]. The consistency of a schema is defined by a set of so called *schema invariants* of the given object data model [Bré96]. In this section, we present a summary of the invariants for the ODMG object model.

Term	Description
types($\mathcal{C}$)	All the types in the system
s, t, T, $\perp$	elements of **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*
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 <t,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*
H-out-paths(t)	The set of all inherited out-paths of type *t*
out-degree(t)	The count of all paths going out of type*t*
self-degree(t)	The count of all self paths of type*t*
H-out-degree (t)	The count of all *H-out-paths(t)* that are not self-referential
T-IN(t)	The total in-degree: in-degree(*t*) + self-degree(*t*)
T-OUT(t)	The total out-degree: out-degree(*t*) + self-degree(*t*) + H-out-degree(*t*)
$\mathcal{R}$	The set of all relations in the system
N(t)	The native(local) properties of type *t*
H(t)	The inherited properties of type *t*

Table 1: Notation for Axiomatization of Schema Changes

Table 1 shows the notation we use for describing the axiomatic model. In the table, *native* properties *N(t)* refer to the properties of a `type` *t* that are defined locally in the type. *Inherited* properties of a `type` *t* refer to the union of all the properties defined by all the supertypes of `type` *t*. The *in-paths* and the *out-paths* are a set of pairs of <`type, name`>, i.e., each in-path and out-path is represented as a pair $<$ `c`$_1$`, r`$_1$ $>$ where `c`$_1$ is the name of the class referring to *t* or the class being referred to by *t* and `r`$_1$ is the name of the reference attribute. The *in-degree* and the *out-degree* of a type is given by the count of all the types other than itself referring to the type and vice versa. The *self-degree* is the count of self-references for a type. Thus the total in-degree, *T-IN* of a type is given by the sum of the in-degree and the self-degree. Similarly, the total out-degree, *T-OUT* is the sum of the out-degree, the self-degree and the inherited out-degree.

Axiom of Rootedness. There is a single type `T` in $\mathcal{C}$ that is the supertype of all types in $\mathcal{C}$. The type `T` is called the *root* [1].

[1] ODMG defines this *root* as an *object*.

Axiom of Closure. Types in $\mathcal{C}$, excluding *root*, have supertypes in $\mathcal{C}$, giving closure to $\mathcal{C}$.

Axiom of Pointedness. There are many types $\perp$ in $\mathcal{C}$ such that $\perp$ has no subtypes in $\mathcal{C}$. $\perp$ is termed a *leaf*.

Axiom of Nativeness. The native properties [2] of a type `T`, *N(t)*, is the set of properties that are locally defined within a type.

Axiom of Inheritance. The inherited properties of a type `T`, *H(t)*, is the union of the inherited and native properties of its immediate supertype *P(t)* .

Axiom of Distinction. All types `T` in $\mathcal{C}$ have distinct names. Every property `p` for a type `T` has a distinct name. All relationships r in $\mathcal{R}$ have a distinct name.

Axiom of Degree. The ratio of total in-degree, *T-IN* of the schema , to the total out-degree, *T-OUT* of the schema is an invariant.

3 Review of the SERF Framework

In this section we now present a brief overview of the SERF framework [CJR98]. The concepts introduced here form a basis for the discussion in the following sections. Schema evolution support today does not necessarily cover all the changes that a user might potentially want to make to an object schema. It is hard, if not impossible, to provide a set of *pre-defined* changes that would meet every users' needs. In SERF we address this limitation and allow users to safely customize semantics of transformations in a *flexible* and *re-usable* manner [CJR98]. Our approach is based on the hypothesis that complex schema evolution transformations can be decomposed into a sequence of basic evolution primitives, where each basic primitive is an invariant-preserving atomic operation with fixed semantics provided by the underlying OODB system. To effectively combine these primitives and perform arbitrary transformations on objects within a complex transformation, we rely on a standard query language, namely OQL [Cea97]. In our work, we have demonstrated that a language such as OQL is sufficient for accomplishing schema evolution, thereby re-using existing technology and showing its adaptability for our system. The SERF system is proposed as a value-added re-structuring layer on top of existing database systems.

A SERF transformation *flexibly* allows a user to define different semantics for any type of schema transformation (see Figure 1). However, these transformations are *not re-usable* across different classes or different schemas. To address this, we have introduced the notion of templates in the SERF framework [CJR98]. A template uses the query language's ability to query over the meta data (as stated in the ODMG Standard) and is augmented by a name and a set of parameters to make transformations *generic* and *re-usable* (Figures 2). Thus, when the example SERF template in Figure 2 is instantiated with actual schema elements it results in the SERF transformation shown in Figure 1. A template is thus an arbitrarily complex transformation that has been encapsulated and parameterized.

An implementation of the SERF framework, called OQL-SERF, is currently being developed at Worcester Polytechnic Institute. It is based on the ODMG standard and uses the ODMG object model, the ODMG Schema Repository definition, and OQL. The system is being implemented entirely in Java and uses Object Design's Persistent Storage Engine (PSE) for Java as its back-end database [O'B97]. The system is being demonstrated at SIGMOD'99 [RCL+99] in May 1999 and will be released to public domain in the summer of 1999.

4 Minimal Primitives for Relationship Evolution

In any OODB system, when an object model is extended as in the case of relationships, the schema evolution support for those extensions must also be provided. In this section we present the essential set of evolution primitives that are needed for the evolution of uni-directional (unary) relationships as well as for bi-directional

[2] As per ODMG definition, the set of properties includes all the set of relationships and attributes.

```
add_atomic_attribute (Person, Street
                        String," ");
add_atomic_attribute (Person, City
                        String," ");
add_atomic_attribute (Person, State
                        String," ");
define extents() as
   select c
   from Person c;

for all obj in extents():
   for all AA in AddressAttrs ()
       obj.set (obj.AA, valueOf(obj.address.AA))

delete_attribute (Person, address);
```

Figure 1: Inline Transformation Expressed in OQL with Embedded Evolution Primitives.

```
begin template inline (className, 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

Legend: cName: OQL variables
        $className: template variables
        refClass: user variables
```

Figure 2: The Inline Template.

(binary) relationships beyond the basic primitives required for the ODMG object model. The primitives presented here are *essential* and *minimal* in that they cannot be decomposed into a sequence of any other evolution primitives. They can however be composed together with other primitives [3].

4.1 Evolution of Unary Relationships

As per the ODMG object model as well as most other common object models such as O_2, IQL, Orion, etc. [Tec94, AS95, KGBW90], unary relationships are generally modeled via the use of a reference attribute. For example, Figure 4 shows an unary relationship between classes `Teacher` and `Course` via the reference attribute `teaches`. Given that unary relationships are modeled as reference attributes we have two evolution primitives *add-reference-attribute* and *delete-reference-attribute* that allow us to add and delete a unary relationship.

Figure 3: Graphical Schema Description of the Uni-Directional teaches Relationship

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

Figure 4: A Uni-Directional Relationship between Two Classes `Teacher` and `Course`.

The add-reference-attribute Primitive. This primitive allows us to add 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` as

[3]The complete taxonomy of schema evolution primitives can be found in Appendix A.

shown in Figure 4. Figure 5 shows the primitive as a SERF template with *contracts* (refer Section 6) used to describe its behavior and the *pre-conditions* that must be satisfied prior to its execution.

add-reference-attribute (C_s, r, C_d, default)
{
    requires:

C_s, $C_d \in \mathcal{C}$ $\wedge$
$\sigma(C_s)$, $\sigma(C_s) \in$ **types**$(\mathcal{C})$ $\wedge$
r $\notin N(C_s)$

add-reference-attribute-primitive
(C_s, r, C_d, default)

ensures:
r $\in N(C_s)$ $\wedge$
$<C_s,\mathbf{r}> \in$ *in-path*(C_d) $\wedge$
$<C_d,\mathbf{r}> \in$ *out-path*(C_s) $\wedge$
$\forall$ **x** $\in$ *sub*$^*(C_s)$
$<C_d,\mathbf{r}> \in$ *out-path*(**x**)
}

Figure 5: Add-Reference-Attribute Primitive Template with Contracts

delete-reference-attribute (C_s, r)
{
    requires:

$C_s \in \mathcal{C}$ $\wedge$
$\sigma(C_s) \in$ **types**$(\mathcal{C})$ $\wedge$
r $\in N(C_s)$ $\wedge$
domain(**r**) $\in \mathcal{C}$ $\wedge$
$\sigma($*domain*(**r**)$) \in$ **types**$(\mathcal{C})$ $\wedge$
$\alpha(\mathbf{r}) \in$ **types**$(\mathcal{C}) \times$ **types**$(\mathcal{C})$

delete-reference-attribute-primitive
(C_s, r, C_d, default)

ensures:
r $\notin N(C_s)$ $\wedge$
$<C_s,\mathbf{r}> \notin$ *in-path*(*domain*(**r**)) $\wedge$
$<$*domain*(**r**),**r**$> \in$ *out-path*(C_s) $\wedge$
$\forall$ **x** $\in$ *sub*$^*(C_s)$
$<$*domain*(**r**),**r**$> \notin$ *out-path*(**x**) $\wedge$
$\alpha(\mathbf{r}) \notin$ **types**$(\mathcal{C}) \times$ **types**$(\mathcal{C})$
}

Figure 6: Delete-Reference-Attribute Primitive Template with Contracts.

The delete-reference-attribute Primitive. The *delete-reference-attribute* as depicted in Figure 6 allows the deletion of an existing uni-directional relationship between two types. This primitive is an inverse operation of the *add-reference-attribute* and so removes the traversal paths from the *in-paths()* and the *out-paths()* respectively. Similar to the *add-reference-attribute* we use *contracts* to specify the constraints for the primitive as shown in Figure 6.

4.2 Evolution of Bi-directional Relationships

Bi-directional relationships on the other hand are modeled as an explicit declaration of an inverse relationship in both the involved classes. A binary relationship is defined by the declaration of *traversal paths* that enable applications to use the logical connections between the objects participating in the relationship. These traversal paths are declared in pairs, one for each direction of traversal of the binary relationship. Thus, a relationship pair is denoted as *relationship X inverse Y*[4].

For example, the relationship syntax for the classes in Figure 7 is given in Figure 8. In the class `Teacher`, the attribute `teaches` is a *reference* attribute of type `Course` and the inverse of this relationship is given by 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`).

Note that a bi-directional relationship can be broken down into a pair of uni-directional relationships between the two types and hence we propose that the only two primitives needed for the manipulation of bi-directional relationships are: *form-relationship* and *drop-relationship*. The primitive *form-relationship*

[4]Not all the parameters of a relationship are shown.

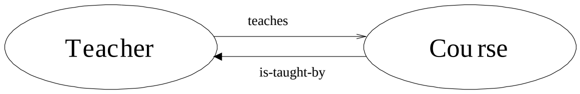

Figure 7: Graphical Schema Description of the teaches - is-taught-by Relationship.

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

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

Figure 8: ODMG Syntax for Specifying a Bi-Directional Relationship.

elevates the status of two already existing uni-directional relationships between two types. This primitive also ensures the referential integrity of all the objects that are elements of the extent of the two involved classes (see Section 4.1 for details). The *drop-relationship* deprecates a bi-directional relationship to a pair of uni-directional relationships. Figures 9 and 10 show the two primitives with the contracts in place respectively.

5 Effects of Relationships on Existing Evolution Primitives: The Consistency Problem

When an object model is extended, it is not sufficient to simply provide extra evolution support for the extensions (as done in Section 4 but also to closely re-examine the basic evolution primitives for the given object model (as listed in Figure 3) to determine how they are impacted. In this section we describe the effects of relationships on the existing taxonomy of primitives.

While the core functionality of existing evolution primitives is unaffected, the constraints that need to be checked to determine when they can be applied may have greatly changed. Consider for example the *delete-class*(C_i) evolution primitive [PS87]. This primitive can only be applied when the class $\mathtt{C}_i$ is a *leaf* class (refer to Figure 11), i.e:

$$sub(C_i) \quad = \quad \emptyset. \tag{1}$$

However, while this is a necessary and sufficient constraint for the delete of the `HomeAddress` class specified in the schema depicted in Figure 12, it is no longer a sufficient stipulation for a schema that contains relationships as in Figure 13.

For example, the delete of the *leaf* class `Address` in the schema in Figure 13 is a valid evolution operation as per the constraints specified in Figure 11 and Equation 1. This however causes dangling references and hence compromises the consistency of the system by violating both the structural integrity (schema-level) and the referential integrity (object-level) of the system. It therefore becomes essential to re-implement the *delete-class* primitive and introduce a constraint such that a class cannot be deleted if it has other classes referring to it. Using the notation in Table 1 this could be expressed as:

$$in - degree(C_i) \quad = \quad 0 \tag{2}$$

However, while the conditions in Equations 1 and 2 ensure the structural integrity of the schema, they still cannot ensure the referential integrity. Consider for example the schema shown in Figure 14. In this example, the class `Person` has a direct relationship with the class `Address`, while the class `Home-Address` is inherited from the class `Address`. The class `Person` and all its subclasses `Student` and `Teaching-Assistant` inherit

form-relationship (C_s, $\mathbf{r}_s$, C_d, $\mathbf{r}_d$)
{
requires:

$\mathrm{C}_s,\ \mathrm{C}_d \in \mathcal{C}\ \wedge$
$\sigma(\mathrm{C}_s),\ \sigma(\mathrm{C}_d) \in \mathbf{types}(\mathcal{C})\ \wedge$
$\mathrm{r}_s \in N(\mathrm{C}_s)\ \wedge$
$\mathrm{r}_d \in N(\mathrm{C}_d)\ \wedge$
$<\mathrm{C}_s,\ r_s> \in \textit{in-path}(C_d)\ \wedge$
$<\mathrm{C}_d,\ r_d> \in \textit{in-path}(C_s)\ \wedge$
$\alpha(r_s) \in \mathbf{types}(\mathcal{C}) \times \mathbf{types}(\mathcal{C})\ \wedge$
$\alpha(r_d) \in \mathbf{types}(\mathcal{C}) \times \mathbf{types}(\mathcal{C})$

form-relationship-primitive
(C_s, r_s, C_d, r_d);

ensures:
$\alpha(r_s) =< C_s, C_d > \wedge$
$\alpha(r_d) =< C_d, C_s >$

}

Figure 9: Form-Relationship Primitive Template with Contracts

drop-relationship (C_s, $\mathbf{r}_s$, C_d, $\mathbf{r}_d$)
{
requires:

$\mathrm{C}_s,\ \mathrm{C}_d \in \mathcal{C}\ \wedge$
$\sigma(\mathrm{C}_s),\ \sigma(\mathrm{C}_d) \in \mathbf{types}(\mathcal{C})\ \wedge$
$\mathrm{r}_s \in N(\mathrm{C}_s)\ \wedge$
$\mathrm{r}_d \in N(\mathrm{C}_d)\ \wedge$
$<\mathrm{C}_s,\ r_s> \in \textit{in-path}(C_d)\ \wedge$
$<\mathrm{C}_d,\ r_d> \in \textit{in-path}(C_s)\ \wedge$
$\alpha(r_s) \in \mathbf{types}(\mathcal{C}) \times \mathbf{types}(\mathcal{C})\ \wedge$
$\alpha(r_d) \in \mathbf{types}(\mathcal{C}) \times \mathbf{types}(\mathcal{C})\ \wedge$
$\alpha^{-1}(\mathrm{r}_d) = \mathrm{r}_s$ and $\alpha(\mathrm{r}_s) = \mathrm{r}_d$

drop-relationship-primitive
(C_s, r_s, C_d, r_d);

ensures:
$\neg(\alpha(r_s) =< C_s, C_d > \wedge$
$\alpha(r_d) =< C_d, C_s >)$

}

Figure 10: Drop-Relationship Primitive Template with Contracts

the relationship to the class `Address`. However, when instantiating the class `Person` or any of its subclasses it is possible at the object level to instantiate a relationship with an object of the type `Home-Address` rather than an object of type `Address` [5]. Thus, while under the conditions in Equations 1 and 2, a *delete-class*(`Home-Address`) would be allowed and the structural integrity of the system would not be violated, we could potentially violate the referential integrity of the system.

To capture consistency violations at the object level, we now extend the notation presented in Table 1 and introduce the notion of *in-degree()* and *out-degree()* at the object level as follows:

- *obj-in-degree(o_i)*: The number of objects referring to the object o_i.
- *obj-out-degree(o_i)*: The number of objects being referred to by the object o_i.

[5] Type casting is allowed in most object-oriented languages.

```
boolean delete-class (Class c)
{
   if (c.subclasses().count() == 0 )
   {
       delete all objects of c
       destroy the class c
        return true;
    }
    else return false;

}
```

Figure 11: Pseudo-Code for *delete-class* Primitive

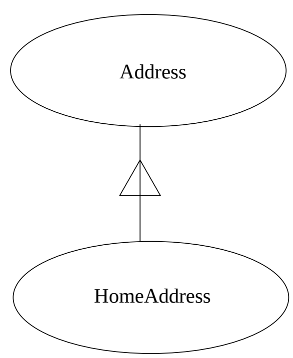

Figure 12: An Example Schema Showing No Relationships

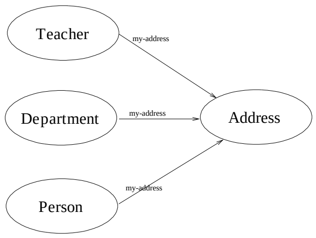

Figure 13: An Example Schema Showing Relationships

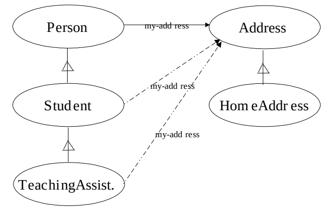

Figure 14: A Sample Schema Containing Relationships via Inheritance

We thus define a third constraint for the *delete-class* primitive that must hold before the deletion of a class can occur:

$$\forall o_i \in \textbf{extent}(t): \quad obj - in - degree(o_i) = 0 \quad for \quad t = type(C_i) \tag{3}$$

The constraints expressed in Equations 1, 2 and 3 ensure the consistency of the database both in terms of the structural as well as the referential integrity.

6 Contract-Based Solution for Consistent Relationship Evolution

Adding hard-coded constraints as they are thought of or as the object model is updated to support more expressive modeling constructs is not only tedious but also a very expensive process. This entails the database vendors updating their OODB and releasing a new version on the market. While not all the cost of the process can be removed, we present a solution here which would enable us to greatly trim these costs and in some scenarios even completely eliminate them.

Our approach is based on the hypothesis (as shown in Section 5) that the actual functionality of these primitives is not being changed. Rather, the change is in the constraints under which these primitives can be applied caused by the presence of additional semantic constructs in the model. Furthermore we present the extensions that need to be added to the base SERF framework to support this solution.

Our Approach. We use the SERF *template* to provide the externally updatable view of an evolution primitive. However, the existing SERF system as presented in Section 3 is not powerful enough to express the constraints in Equations 1, 2 and 3. For this purpose we now introduce the notion of software *contracts* for SERF templates [Mey92]. These contracts provide a declarative description of the behavior of a primitive as well as a mechanism for expressing the constraints that must be satisfied prior to the execution of the actual evolution primitive.

Pre-Conditions for a Contract Template. The constraints, termed *pre-conditions*, occur prior to any code (OQL statement) in a template. Figure 15 shows the constraints for the *delete-class* primitive as pre-conditions [6].

The *pre-conditions* are separated from the actual OQL statements by means of the keyword **requires** and are expressed using set notation shown in Table 1. This mechanism allows the SERF system to verify the truth of each of the listed pre-conditions against the system dictionary [7]. If all of the pre-conditions are satisfied, the actual primitive code can be executed using the OODB interface provided for the schema evolution primitive.

[6] The notation used in here is not the final contract language but an easy to understand set-theoretic notation. While the final contract language we are employing will have a more OQL-like flavor as it is assumed that the template developer will be well-versed in a declarative languages such as OQL.

[7] We require the OODB system to give us access to the system-dictionary. This is listed as part of the system requirements for SERF [CJR98].

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

`template body here`
}

Figure 15: Pre-Conditions for Delete-Class Primitive in Contractual Form

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

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

Figure 16: Post-Conditions for the Delete-Class Primitive Template

Post-Conditions. The behavior of the primitives is declared by *post-conditions*, a set of contracts that occur after the execution of the actual schema evolution operation. These post-conditions describe the exact changes that are made to the schema by the evolution primitive and hence its behavior. We extend the pre-condition verification process to do the post-condition verification and validate that the primitive actually accomplished what it set out to do. Figure 16 shows the post-conditions of the *delete-class* primitive.

Thus, together the *pre-conditions* and the *post-conditions* declaratively define both the constraints of the schema evolution primitives that must be satisfied prior to execution of the primitive as well as the behavior of the evolution primitive itself. However, as the schema evolution primitives are OODB system dependent, it is not possible to have a generic template library of all the evolution primitives. Instead a user of the SERF system would have to provide a basic library of the schema evolution primitives for their OODB system and describe the contracts for each of the primitives.

Advantages of Contracts. SERF templates with contracts provide faster updates to the OODB system when the underlying object model is updated for example with relationships as we would simply now add additional declarative constraints and behavior to the schema evolution primitive template rather than updating 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* help verify the truth of the execution of the primitive and hence aid in the verification process. With embedded templates (refer to Section 8) this advantage has even bigger benefits beyond the evolution primitive templates.

7 SERF Extensions for Flexible Evolution of Relationships

The primary advantage of the basic SERF framework is the ability for users to specify transformation semantics that go far beyond the *fixed* set provided by a given OODB system. Once we added relationships to the underlying object model, the SERF framework allows users to specify additional transformations now involving relationships. Figure 17 shows a template that creates a bi-directional relationship between two classes when at least one referential relationship exists between the classes. This shows how we can take advantage of the single referential relationship and perform the object transformations for maintaining the correctness of the bi-directional relationship all within our template structure (Appendix C shows some more examples of such templates).

User Functions. However, in our case study of exploring complex transformations for relationship evolution, we found that we were unable to design several complex transformations such as building a bi-directional relationship between two disjoint classes within the current SERF framework. While at the schema level it is easy to add these relationships, at the object level it is impossible to achieve a variety of desired semantics in a general manner and hence by a SERF template. In order to accomplish this, we now propose to extend

```
// This template adds a relationship between two partially disjoint
// classes i.e., one class has a one-sided relationship to the
// other. Object transformations in this scenario are very possible.

// A One-to-many relationship => source has many of inverse

begin template add_1m_relationship ( source-Class,
                                     source-attrib-Name,
                                     inverse-Class,
                                     inverse-attrib-Name,
                                     inverse-attrib-Type,
                                     inverse-attib-Value )

{
    // Add the inverse-attrib-Name to inverse-Class
    add_atomic_attribute ($inverse-Class, $inverse-attrib-Name,
                          $inverse-attrib-Type, $inverse-attrib-Value);

    // Fix up the objects
    // Get the extent of the source-Class
    define extents ($cName) as
        select c
        from $cName c;

    // set: inverse-class.inverse-attrib-Name = source-Object
    for all source-Object in extents ($source-Class):
        for all obj in $source-Object.source-attrib-Name
            obj.set(obj.inverse-attrib-Name, source-Object)

    // promote to relationship
    promote_to_relationship ($source-Class, $source-class-Name,
                             $inverse-Class, $inverse-class-Name);
}
```

Figure 17: Converting a Uni-directional Relationship to a Bi-directional Relationship

```
// This template adds a relationship between two completely disjoint
// classes. No object transformations are performed at this point.

begin template add_relationship ( Class source-Class,
                                  Attribute source-attrib-Name,
                                  String source-attrib-Type,
                                  String source-attrib-Value,
                                  Class inverse-Class,
                                  Attribute inverse-attrib-Name,
                                  String inverse-attrib-Type,
                                  String inverse-attrib-Value,
                                  String transformer )


{
    // Add the two uni-directional relationships

    add_reference_attribute ($source-Class, $source-attrib-Name,
                             $source-attrib-Type, $source-attrib-Value);

    add_reference_attribute ($inverse-Class, $inverse-attrib-Name,
                             $inverse-attrib-Type, $inverse-attrib-Value);

    // transform objects by invoking a user function
    $transformer ();

    // promote to relationship

    form_relationship ($source-Class, $source-class-Name,
                       $inverse-Class, $inverse-class-Name);

}
```

Figure 18: Adding a Relationship between two Disjoint Classes

the SERF Framework to also offer **User Functions**. User functions are defined to be SERF transformations that allow us in general to store and invoke at a later date SERF transformations. In the context of flexible evolution of relationships, user functions allow us to invoke a SERF transformation, a very specific piece of code, from within a SERF template, a general piece of code. In this manner we can build the relationship at the schema level in a generalized manner, but can invoke the specific user function to do the object level transformations in a customized manner.

However the current SERF system is limited, in that it does not allow embedded templates, i.e., it does not have any support for invoking a user function from within a template. Moreover, as a user function may return an output that could be used in the next step of the template, we find it necessary to also support a more sophisticated type system than the current, i.e., `String`. Thus, to fully support user functions we have further enhanced the SERF system to support embedded templates and a typed system. A list of the supported types in SERF templates can be found in Appendix B. As a part of this typed system we also allow the user to pass in the name of a user function that is to be executed from within the template.

Figure 18 shows an example of a template that builds a bi-directional relationship between two disjoint classes now using typed parameters. Thus, a variable of the type `Method` specifies that the variable is a user function. In this example, `$transformer` specifies the user function and must be provided by the user during the instantiation process and must be bound to a user function in the template library at run-time. While the template builds a bi-directional relationship between two arbitrary classes `sourceClass` and `inverseClass`, the user function provided by the user is specific and hence might do object transformations for two classes `Person` and `Address`. Thus, this user function is specific to a set of parameters and is valid and correct only for the specific set. Thus, user functions allow us to to have a disjoint relationship template which in their absence would not be possible.

8 SERF: Making Extending the Object Model Easy!

In the previous section, we have shown how with basic evolution support (Section 4) for relationships and some extensions in the SERF framework (Section 7) we can achieve schema evolution transformations beyond the primitives that may be supplied by the system. In this section we show how SERF can help us achieve the primitive schema evolution for extensions to the object model such as *aggregate* relationships, keys, Java's *implements* relation etc. (Figure 19). The SERF system thus helps us bridge the gap between the design methodologies, the support for such models at the database level, and the impact of these new features on schema evolution. It allows us to define basic evolution support for a certain class of object extensions.

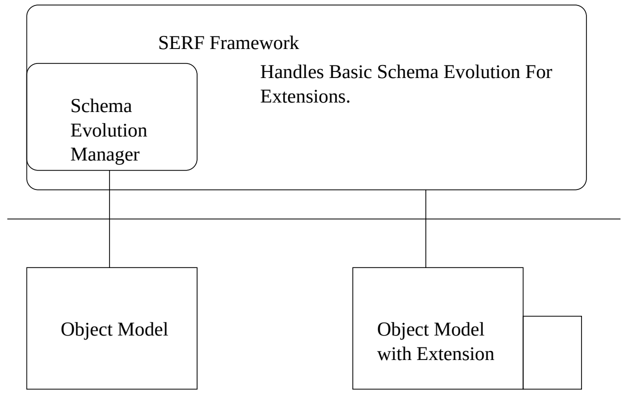

Figure 19: Using SERF to Provide Schema Evolution for the Extended Features of an OODB Model

As an example, in Section 8.1 we present an extension of the object model by *aggregation* relationships and list the support needed by SERF from the OODB system. In Section 8.2 based on the extensions presented in Section 7 we show how SERF can help provide extended schema evolution support without re-writing the existing schema evolution support for a system.

8.1 System Requirements

In order for the SERF system to handle the evolution of such an *aggregation relationship*, including the creation of these relationships, we expect the underlying OODB system to provide us with:

1. An updated object model with the desired semantic constructs,
2. System functions that maintain the desired semantics at the object level,
3. System functions to maintain referential integrity,
4. System functions to maintain and update the system dictionary.

The features 1 through 3 are obvious features that any system would provide when they extend their object model. The fourth feature, providing access to the system dictionary and functions for the maintenance of the same, is a requirement set forth by the SERF system. This enables the SERF system to directly update the system dictionary such that any changes made externally can be still registered with the system and the system dictionary is kept in sync.

Extending the Object Model. An *aggregation* relationship denotes a whole/part hierarchy, with the ability to navigate from the whole (*aggregate*) to its parts (*attributes*) such that if the *aggregate* object is deleted then its *attribute* objects also need to be deleted. To capture this notion in our object model, we use the symbol "$\diamond\!\!-$" [8] to specify that one type aggregates another type. Thus, we extend our basic definition of *referential* relationship "$\longrightarrow$" to now encompass the semantics of an *aggregation* relationship.

Definition 3 *The* aggregation relationship r $\in \mathcal{R}$ *on* $\sigma(\mathcal{C})$ *is a specialized* referential relationship C_1*, denoted by* $C_1 \overset{r}{\diamond\!\!-} C_2$*, implies that the existence of* C_2 *is dependent on the existence of* C_1*, where* $C_1, C_2 \in \sigma(\mathcal{C})$*.*

Extending the System Dictionary. Using the above we now expect the system dictionary to be extended such that it provides support for the features listed in Table 2. In addition, the OODB system also needs to provide a public interface for updating and maintaining these paths such that they are accessible [9].

[8] This symbol is UML notation for aggregation.

[9] [CJR98] lists the system requirements that must be met for SERF by a query language, namely OQL.

Term	Description
agg-paths(t)	The set of all aggregation paths going out of type t
agg-degree(t)	The count of all aggregation paths outgoing from type t
self-agg-paths(t)	The set of all self-referential aggregation paths going out of type t
self-out-degree(t)	The count of all self-referential paths going out of type t

Table 2: Notation for Aggregation Relationships

Assuming that the system already has support for referential relationships, we expect the following functionality to be provided:

- `upgrade-to-aggregation(`C_d`, r)`. This upgrades a referential relationship to an aggregation relationship. The path (C_d, r) is removed from the set of `out-paths` and added to the set of `agg-paths`. No object level manipulations are required by this function.

- `downgrade-aggregation(`C_d`, r)`. This is an inverse function of `upgrade-to-aggregation(`C_d`, r)` and downgrades an aggregation relationship to a referential relationship. The path (C_d, r) is removed from the set of `agg-paths` and added to the set of `out-paths`. No object level manipulations are required by this function.

8.2 SERF Wrapper for Schema Evolution of an Extended Object Model

In this section we show how we can provide the basic schema evolution primitives for the *aggregate* relationship at the SERF level rather than at the system level. Figure 20 shows an *aggregation relationship* between two classes `Person` and `Address` where the `Person` *contains* the `Address` [10]. In order to provide evolution of the *aggregation* relationship, we need to supply primitives to add, delete and modify it.

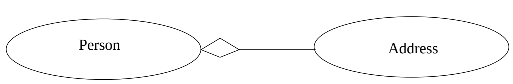

Figure 20: A Sample Schema Showing Aggregate Relationship.

form-aggregation-relation (C_s, r, C_d, default)
{ **requires:**

C_s, $C_d \in \mathcal{C} \wedge$
$\sigma(C_s)$, $\sigma(C_s) \in \mathbf{types}(\mathcal{C}) \wedge$
$\mathbf{r} \notin N(C_s)$

add-reference-attribute-primitive
(C_s, r, C_d, default);
`upgrade-to-aggregation ( `C_d`, r );`

ensures:
$\mathbf{r} \in N(C_s) \wedge$
$<C_s,\mathbf{r}> \in \textit{in-path}(C_d) \wedge$
$<C_d,\mathbf{r}> \in \textit{agg-path}(C_s) \wedge$
$\forall\ \mathbf{x} \in \textit{sub}^*(C_s)$
$<C_d,\mathbf{r}> \in \textit{agg-path}(\mathbf{x})$
}

Figure 21: A Template For Creating an Aggregate Relationship Between Two Classes

SERF Extensibility - Basic Evolution Primitives are not Needed. Figure 21 shows an example of the *aggregation* SERF template that creates an *aggregation* relationship between two classes. Here, we use

[10]We use the UML methodology in the figure to show the aggregation.

the evolution primitive `add-reference-attribute` (Section 4.1) to first create a uni-directional relationship between the two classes. We then use the system dictionary function `upgrade-to-aggregation` to inform the OODB system that we would like to apply the aggregation semantics for this relationship [11]. Similarly we can write templates for doing the deletion and the modification of the *aggregation* relationship.

delete-aggregator (C_i)
{ **requires:**
$C_i \in \mathcal{C} \wedge$
$\sigma(C_i) \in \textbf{types}(\mathcal{C}) \wedge$
$sub(C_i) = 0 \wedge$
$in\text{-}degree(C_i) = 0 \wedge$
$\forall\, o_i \in \textbf{extent}(t)$
$obj\text{-}in\text{-}degree(o_i) = 0 \wedge$
$self\text{-}agg\text{-}degree(C_i) = 0$

agg-List = select c.agg-paths from c in MetaClass where c.name = C_i

for all X in *agg-List*
`downgrade-aggregation(X)`;
delete-reference-attribute (C_i, *X.refAttr*);
delete-class(*X.className*);

delete-class(C_i);

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

Figure 22: Template for Handling the Deletion of an Aggregator.

SERF Flexibility - Update to Existing Taxonomy Not Needed. As we have shown in Section 5 with an updated object model providing evolution support for the extensions is not sufficient. We also now need to consider its impact on the existing taxonomy of schema evolution primitives. For example, the `delete-class` primitive template that we introduced in Section 6 is no longer sufficient for handling the deletion of a class C_i that has an *aggregation* relationship with another class C_j. The `delete-class` primitive now needs to be able to handle the propagation of the delete of an *aggregator* to all of the *aggregated* classes.

In Figure 22 we show a delete template that can handle the propagation of the delete to the *aggregated* classes. In this template, we first downgrade the aggregation relationship to a referential relationship using the system provided function `downgrade-aggregation`. The evolution primitive `delete-reference-attribute` deletes all the downgraded aggregation relationships and the `delete-class` template then deletes all of aggregated classes themselves [12]. The final step to delete the aggregator itself is accomplished by the last invocation of the `delete-class` template. In all of these cases we make use of the `delete-class` template rather than directly invoking the `delete-class-primitive` as all of the consistency violations that could have occured before can also occur in this scenario.

[11] Note that this function is very different from a schema evolution function, as we are really not doing any object manipulations.

[12] There is a possibility of failure of the `delete-class` template for an aggregated class as it is possible that the aggregated class participates in a relationship with some other class. However for simplicity we ignore this situation.

SERF Contracts - Maintaining Consistency. Moreover, using contracts in these templates we can now add safe-guards so that we can maintain schema consistency at a level higher than that of the primitive templates. For example, in the `delete-aggregator` template in Figure 22 we add a new constraint that in order for the deletion of the aggregator to occur it is required that there be no self-referential aggregation. Without this constraint the failure of the final `delete-class` template i.e., the delete of the aggregator itself, would fail as it would presumably be deleted prior to the last `delete-class` template invocation. Thus, contracts in SERF templates also aid in preventing failures which might otherwise result in either a corrupt database or massive amounts of recovery neither of which is very desirable!

9 Related Work

Relationships. Semantic modeling research has looked into the modeling of relationships and the different semantics that can be applied for these relationships [Boo94]. In object databases, Kim, Bertino and others [KBG89] have examined the part-whole relationship (composite objects). The composite objects in Orion [KBG89] however do not include the notion of referential integrity constraints. Thus, an object may be deleted even if it is a component of another object. Bertino et al. [BG98] have presented a formal composite object model that now supports referential integrity constraints in the framework of the ODMG object model. However, we find that their work is limited in several ways. One, they focus only on the composite object relationships rather than dealing with the general issues involved with uni-directional and bi-directional relationships. Two, they do not consider structural integrity, i.e., the consistency of the schema itself. Lastly, they do not study the issue of schema evolution on a now extended object model.

Basic Schema Evolution. 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. One key issue in schema evolution is understanding the different ways of changing a schema. The first taxonomy of primitive schema evolution operations was defined by Banerjee et al. [BKKK87]. They defined consistency and correctness of these primitives in the context of the Orion system. Until now, current commercial OODBs such as Itasca [Inc93], GemStone [BMO$^+$89], ObjectStore [Obj93], and O_2 [Tec94] all essentially handle a similar set of fixed evolution primitives; though based on their own respective object models.

However, evolution of relationships (uni-directional relationships) has been so far dealt in an implicit manner by treating these as reference/complex attributes and treating their evolution in a manner similar to that of basic attributes (such as integers, string etc.). Such treatment of evolution of relationships can potentially lead to violations of system consistency both at the structural and the referential level. They also do not examine the effects of the relationships in the object model on the existing set of schema evolution primitives. Thus, the evolution of relationships, uni-directional and bi-directional, and the effects of relationships on other evolution primitives to the best of our knowledge has not been explicitly addressed by any current research or commercial OODB system.

Advanced Schema Evolution. In recent years, the advent of more advanced applications has led to the need for support of complex schema evolution operations. Both Breche and Lerner [Bré96, Ler96] have investigated the issue of more complex operations. Lerner [Ler96] has introduced compound type changes in a software environment, i.e., focusing on the type and not on the object instance changes. She provides compound type changes like *Inline*, *Encapsulate*, *Merge*, etc..

Breche [Bré96] proposed a similar list of complex evolution operations for O_2, i.e., now considering both schema as well as object changes. [Bré96] claims that these advanced primitives can be formulated by composing the basic primitives that are provided by the O_2 system. He shows the consistency of these advanced primitives. Like other previous work, the paper however still provides a fixed taxonomy of primitives to the users, instead of giving them the flexibility and extensibility as offered by our approach. Also for object changes, the user is limited to using the object migration functions written in the programming language of O_2.

The SERF framework [CJR98] addresses the problem of having a fixed taxonomy of schema evolution primitives. In SERF, we present a flexible way of doing transformations. We have also shown how a SERF transformation can be generalized to a re-usable, parameterized SERF template. In this work we now present some extensions to this system.

Consistency Management. Part of our work focused on declaratively describing the behavior of our primitives and our templates. We have also used this mechanism to update and mainatin the schema definition during the schema evolution process. Much of our work in this area has been inspired by the work of Bertrand Meyer [Mey92]. He has coined the phrase *Design by Contract* to denote a software development style which emphasises the importance of formal specifications and interleaves them with actual code. This mechanism has in turn been enhanced and used by other researchers [TC98] to do consistency management of applications in terms of detecting violations and doing verification.

Both relational and object databases support some implicit forms of consistency definition. These constraints are a pre-defined set and their enforcement is out of control of the application. These constraints are generally used for transaction rollbacks. Some object models now have some support for referential integrity constraints. These are again implicit constraints.

Object Model. Our work is based on the ODMG object model [Cea97]. This object model supports the core features of an object data model in terms of the basic concepts such as object, obejct identity, class and hierarchy. In addition, it also provides the general framework for specifying relationships. As per ODMG standard relationships much like regular attributes are not treated as *first-class citizens* i.e., they do not have an object identity associated with them. In our work, we have followed this description of the object model.

Abiteboul et al. have in [AHV95] have presented a formal description of a schema. We have used this definition and much of their formalism in our work. We have extended their schema definition to now explicitly include the concept of relationships.

Peters and Ozsu [PO95] have introduced a sound axiomatic model to formalize and compare schema evolution modules of OODBs. We utilize their notations with our extensions for the description of our invariants and primitives.

10 Conclusion

In this work, we have addressed the issue of evolving relationships. Within that context we have analyzed the effects of relationships on other elements of the primitive change taxonomy. We have also shown that having hard-coded constraints embedded in the object model can be problematic when the requirements change. Thus, in order to make the upgrading of the primitives themselves possible, we incorporate the notion of *contracts* into our SERF templates. We furthermore extended the SERF system needs to fully support flexible transformations involving relationships. Moreover, we have shown that the SERF system can be used as a wrapper for extended schema evolution support when the underlying object model is extended.

We are currently working on defining a first-order predicate language for the contracts and are doing further case studies to apply SERF to provide evolution support for other semantic extensions of an object model beyond relationships.

References

[AHV95] S. Abiteboul, R. Hull, and Vianu V. *Foundations of Databases.* Addison-Wesley Publishing Company, 1995.

[AS95] S. Abiteboul and Cassio Souza Santos. IQL(2): A Model with Ubiquitous Objects. In *Workshop on Database Programming Language*, page 10, 1995.

[BG98] E. Bertino and G. Guerrini. Extending the ODMG Object Model with Composite Objects. In *OOPSLA*, pages 259–270, 1998.

[BKKK87] 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.

[BMO+89] 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.

[Boo94] G. Booch. *Object-Oriented Analysis and Design*. Benjamin Cummings Publications, 1994.

[Bré96] P. Bréche. Advanced Primitives for Changing Schemas of Object Databases. In *Conference on Advanced Information Systems Engineering*, pages 476–495, 1996.

[Cea97] R.G.G Cattell and et al. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Inc., 1997.

[CJR98] 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.

[Inc93] Itasca Systems Inc. Itasca Systems Technical Report. Technical Report TM-92-001, OODBMS Feature Checklist. Rev 1.1, Itasca Systems, Inc., December 1993.

[KBG89] W. Kim, E. Bertino, and J. F. Garza. Composite objects revisited. *SIGMOD*, pages 337–347, 1989.

[KGBW90] 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.

[Ler96] 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.

[Mey92] B. Meyer. Applying "Design By Contract". *IEEE Computer*, 25(10):20–32, 1992.

[O'B97] P. O'Brien. Making Java Objects Persistent. *Java Report*, 1(1):49–60, 1997.

[Obj93] Object Design Inc. *ObjectStore - User Guide: DML. ObjectStore Release 3.0 for UNIX Systems*. Object Design Inc., December 1993.

[Obj94] Objectivity Inc. White Paper, Schema Evolution in Objectivity, February 1994.

[PO95] 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.

[PS87] D. J. Penney and J. Stein. Class modification in the GemStone object-oriented DBMS. In *OOPSLA*, pages 111–117, 1987.

[RCL+99] E.A. Rundensteiner, K.T. Claypool, M. Li, L. Chen, X. Zhang, C. Natarajan, J. Jin, S. De Lima, and S. Weiner. SERF: ODMG-Based Generic Re-structuring Facility. In *Demo Session Proceedings of SIGMOD'99*, 1999.

[Sjo93] D. Sjoberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1):35–54, January 1993.

[TC98] P. Tarr and L. Clarke. Consistency management for complex applications. In *International Conference on Software Engineering*, pages 230–239, 1998.

[Tec92] Versant Object Technology. *Versant User Manual*. Versant Object Technology, 1992.

[Tec94] O_2 Technology. *O_2 Reference Manual, Version 4.5, Release November 1994*. O_2 Technology, Versailles, France, November 1994.

A Taxonomy of Schema Evolution Operations

In this section we give a table of the current schema evolution operations that are supported for the ODMG object model our system. This is the minimal functionality that we expect an underlying system to provide in order to achieve the flexibility of evolution as stated here. Any subset of this taxonomy would result in reduced flexibility of the SERF system. Table 3 gives this essential taxonomy. We also present here the contract-serf templates for each of the primitives that we present here.

Term	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
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
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
add-reference-attribute$(c_x,\ r_x,\ c_y,\ d)$	Add unary relationship from class c_x to class c_y named r_x
delete-reference-attribute$(c_x,\ r_x,\ c_y)$	Delete unary relationship from class c_x to class c_y named r_x

Table 3: Taxonomy of Schema Evolution Primitives

add-class (C_i, $\mathcal{C}$)
{
    requires:
        $\mathrm{C}_i \notin \mathcal{C} \wedge$
        $\sigma(\mathrm{C}_i) \notin \mathbf{types}(\mathcal{C})$

    add-class-primitive (C_i, $\mathcal{C}$)

    ensures:
        $\mathrm{C}_i \in \mathcal{C} \wedge$
        $\sigma(\mathrm{C}_i) \in \mathbf{types}(\mathcal{C})$
            $\mathrm{C}_i \in \mathit{sub}(\mathrm{root}) \wedge$
}

Figure 23: Add-Class Primitive Template with Contracts

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

    delete-class-primitive (C_i)

    ensures:
        $\forall <\mathrm{C}_x, \mathrm{r}_x> \in \mathit{out\text{-}paths}(\mathrm{C}_i)$
            $<\mathrm{C}_i> \notin \mathit{in\text{-}paths}(\mathrm{C}_x) \wedge$
        $\forall\ \mathrm{C}_x \in \mathit{super}(\mathrm{C}_i)$
            $\mathrm{C}_i \notin \mathit{sub}(\mathrm{C}_x) \wedge$
        $\mathrm{C}_i \notin \mathcal{C} \wedge$
        $\sigma(\mathrm{C}_i) \notin \mathbf{types}(\mathcal{C})$
}

Figure 24: Delete-Class Primitive Template with Contracts

add-ISA-edge (C_i, C_j)
{
requires:

$C_i \in \mathcal{C} \wedge$
$C_j \in \mathcal{C} \wedge$
$\sigma(C_i) \in \mathbf{types}(\mathcal{C}) \wedge$
$\sigma(C_j) \in \mathbf{types}(\mathcal{C}) \wedge$
$\mathit{super}(C_i) = \emptyset$

add-ISA-edge-primitive (C_i, C_j)

ensures:
$C_i \in \mathit{sub}(C_j) \wedge$
$C_j \in \mathit{super}(C_i) \wedge$
$C_i \notin \mathit{sub}(\text{root}) \wedge$
$H(C_i) = N(C_j) \bigcup H(C_j) \wedge$
$\mathit{in\text{-}paths}(C_j) \subseteq \mathit{in\text{-}paths}(C_i)$
}

Figure 25: Add-ISA-Edge Primitive Template with Contracts

delete-ISA-edge (C_i, C_j)
{
requires:

$C_i \in \mathcal{C} \wedge$
$C_j \in \mathcal{C} \wedge$
$\sigma(C_i) \in \mathbf{types}(\mathcal{C}) \wedge$
$\sigma(C_j) \in \mathbf{types}(\mathcal{C}) \wedge$
$C_j \in \mathit{super}(C_i) \wedge$
$C_i \in \mathit{sub}(C_j)$

delete-ISA-edge-primitive (C_i, C_j)

ensures:
$C_i \notin \mathit{sub}(C_j) \wedge$
$C_j \notin \mathit{super}(C_i) \wedge$
$C_i \in \mathit{sub}(\text{root}) \wedge$
$H(C_i) \neq N(C_j) \bigcup H(C_j) \wedge$
$\mathit{in\text{-}paths}(C_j) \not\subseteq \mathit{in\text{-}paths}(C_i)$
}

Figure 26: Delete-ISA-Edge Primitive Template with Contracts

form-relationship (C_s, $\mathbf{r}_s$, C_d, $\mathbf{r}_d$)
{
requires:

$C_s, C_d \in \mathcal{C} \wedge$
$\sigma(C_s), \sigma(C_d) \in \mathbf{types}(\mathcal{C}) \wedge$
$r_s \in N(C_s) \wedge$
$r_d \in N(C_d) \wedge$
$<C_s, r_s> \in \mathit{in\text{-}path}(C_d) \wedge$
$<C_d, r_d> \in \mathit{in\text{-}path}(C_s) \wedge$
$\alpha(r_s) \in \mathbf{types}(\mathcal{C}) \times \mathbf{types}(\mathcal{C}) \wedge$
$\alpha(r_d) \in \mathbf{types}(\mathcal{C}) \times \mathbf{types}(\mathcal{C})$

delete-reference-attribute-primitive (C_s, r, C_d, default)

ensures:
$\alpha^{-1}(r_d) = r_s$ and $\alpha(r_s) = r_d$
}

Figure 27: Form-Relationship Primitive Template with Contracts

drop-relationship (C_s, $\mathbf{r}_s$, C_d, $\mathbf{r}_d$)
{
requires:

$C_s, C_d \in \mathcal{C} \wedge$
$\sigma(C_s), \sigma(C_d) \in \mathbf{types}(\mathcal{C}) \wedge$
$r_s \in N(C_s) \wedge$
$r_d \in N(C_d) \wedge$
$<C_s, r_s> \in \mathit{in\text{-}path}(C_d) \wedge$
$<C_d, r_d> \in \mathit{in\text{-}path}(C_s) \wedge$
$\alpha(r_s) \in \mathbf{types}(\mathcal{C}) \times \mathbf{types}(\mathcal{C}) \wedge$
$\alpha(r_d) \in \mathbf{types}(\mathcal{C}) \times \mathbf{types}(\mathcal{C}) \wedge$
$\alpha^{-1}(r_d) = r_s$ and $\alpha(r_s) = r_d$

delete-reference-attribute-primitive (C_s, r, C_d, default)

ensures:
$\neg(\alpha^{-1}(r_d) = r_s$ and $\alpha(r_s) = r_d$)
}

Figure 28: Drop-Relationship Primitive Template with Contracts

add-attribute (C_s, a_x, t, 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, default)

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

Figure 29: Add-Attribute Primitive Template with Contracts

delete-attribute (C_s, a_x)
{
requires:

$C_s \in \mathcal{C} \wedge$
$\sigma(C_s) \in \mathbf{types}(\mathcal{C}) \wedge$
$a_x \in N(C_s)$

delete-attribute-primitive (C_s, a_x, t, default)

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

Figure 30: Add-Attribute Primitive Template with Contracts

add-reference-attribute (C_s, r, C_d, default)
{
requires:

$C_s, C_d \in \mathcal{C} \wedge$
$\sigma(C_s), \sigma(C_s) \in \mathbf{types}(\mathcal{C}) \wedge$
$\mathbf{r} \notin N(C_s)$

add-reference-attribute-primitive (C_s, r, C_d, default)

ensures:
$\mathbf{r} \in N(C_s) \wedge$
$<C_s,\mathbf{r}> \in \textit{in-path}(C_d) \wedge$
$<C_d,\mathbf{r}> \in \textit{out-path}(C_s) \wedge$
$\forall\ \mathbf{x} \in sub^*(C_s)$
$<C_d,\mathbf{r}> \in \textit{out-path}(\mathbf{x})$
}

Figure 31: Add-Reference-Attribute Primitive Template with Contracts

delete-reference-attribute (C_s, r)
{
requires:

$C_s \in \mathcal{C} \wedge$
$\sigma(C_s) \in \mathbf{types}(\mathcal{C}) \wedge$
$\mathbf{r} \in N(C_s) \wedge$
$domain(\mathbf{r}) \in \mathcal{C} \wedge$
$\sigma(domain(\mathbf{r})) \in \mathbf{types}(\mathcal{C}) \wedge$
$\alpha(\mathbf{r}) \in \mathbf{types}(\mathcal{C}) \times \mathbf{types}(\mathcal{C})$

delete-reference-attribute-primitive (C_s, r, C_d, default)

ensures:
$\mathbf{r} \notin N(C_s) \wedge$
$<C_s,\mathbf{r}> \notin \textit{in-path}(domain(\mathbf{r})) \wedge$
$<domain(\mathbf{r}),\mathbf{r}> \in \textit{out-path}(C_s) \wedge$
$\forall\ \mathbf{x} \in sub^*(C_s)$
$<domain(\mathbf{r}),\mathbf{r}> \notin \textit{out-path}(\mathbf{x}) \wedge$
$\alpha(\mathbf{r}) \notin \mathbf{types}(\mathcal{C}) \times \mathbf{types}(\mathcal{C})$
}

Figure 32: Delete-Reference-Attribute Primitive Template with Contracts

B Type Hierarchy

Figure 33 shows the types that are supported for the SERF templates.

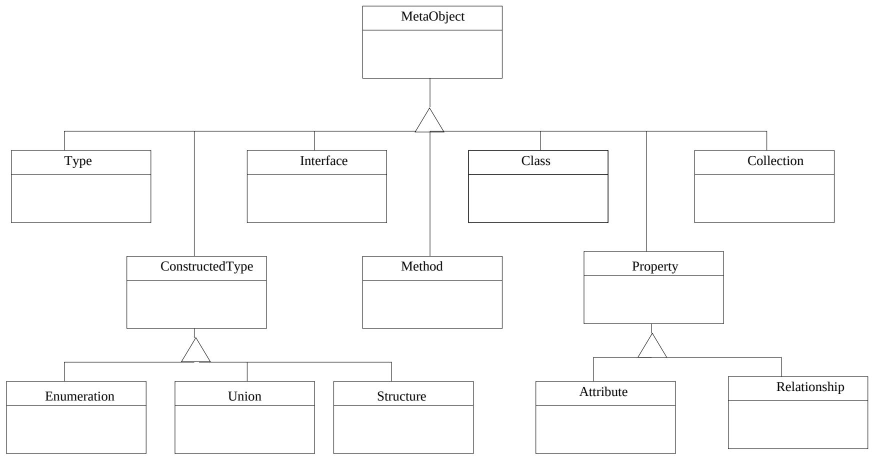

Figure 33: The SERF Template Type Hierarchy

The BNF for the SERF template extended with the these types is given as follows:

$$
\begin{array}{rcl}
template & ::= & \mathbf{begin} - \mathbf{template} \quad template\_name \\
 & & ([parameter]^*) \\
 & & template\_statements \\
 & & \mathbf{end} - \mathbf{template}; \\
template\_statements & ::= & template\_statement \quad | \\
 & & template\_statement \\
 & & template\_statements \\
template\_statement & ::= & define\_query \quad | \quad query | \\
 & & template \\
define\_query & ::= & \mathbf{define} \quad identifier \quad \mathbf{as} \quad query \\
query & ::= & query \\
 & & restricted\_query \\
restricted\_query & ::= & query([function]^*) \quad | \\
 & & \lambda \\
function & ::= & system\_function(basic\_query^*) \quad | \\
 & & schema\_primitive(parameter^*) \\
parameter & ::= & p\_type \quad string\_literal \\
basic\_query & ::= & nil \quad | \quad true \quad | \quad false \quad | \quad literal \\
p\_type & ::= & metaobject \quad | \quad meta\_class| \\
 & & meta\_collection \quad | \quad meta\_structure \\
meta\_property & ::= & meta\_attribute \quad | \quad meta_relationship
\end{array}
$$

C The Relationship Templates

In this section we present some example of relationship templates that can be written using the SERF system. These templates show the flexibility with which we can achieve the building of relationships between two classes.

Add_Many_to_One_Relationship

```
// This template adds a relationship between two partially disjoint
// classes i.e., one class has a one-sided relationship to the
// other. Object transformations in this scenario are very possible.

// A many-to-one relationship => source has one of inverse and inverse
// has many of source

begin template add_m1_relationship ( Class source-Class,
                                     Attribute source-attrib-Name,
                                     Class inverse-Class,
                                     Attribute inverse-attrib-Name,
                                     String inverse-attrib-Type,
                                     String inverse-attrib-Value )
{
    // Add the inverse-attrib-Name to inverse-Class
    add_atomic_attribute ($inverse-Class, $inverse-attrib-Name,
                          $inverse-attrib-Type, $inverse-attrib-Value);

    // Fix up the objects
    // Get the extent of the source-Class
    define extents ($cName) as
        select c
        from $cName c;

    // insertElement: inverse-attrib-Name is a collection so insert the
    // source-Object into the inverse-attrib-Name
    for all source-Object in extents $source-Class:
        source-Object.$source-attrib-Name.$inverse-attrib-Name.
        insertElement(source-Object);

    // promote to relationship
    form_relationship ($source-Class, $source-class-Name,
                       $inverse-Class, $inverse-class-Name);
}
```

Add_Many_to_Many_Relationship Template

```
// This template adds a relationship between two partially disjoint
// classes i.e., one class has a one-sided relationship to the
// other. Object transformations in this scenario are very possible.

// A many-to-many relationship => source has many of inverse and vice versa

begin template add_mm_relationship ( Class source-Class,
                                     Attribute source-attrib-Name,
                                     Class inverse-Class,
                                     Attribute inverse-attrib-Name,
                                     String inverse-attrib-Type,
                                     String inverse-attrib-Value )
{
    // Add the inverse-attrib-Name to inverse-Class
    add_atomic_attribute ($inverse-Class, $inverse-attrib-Name,
                          $inverse-attrib-Type, $inverse-attrib-Value);

    // Fix up the objects
    // Get the extent of the source-Class
    define extents ($cName) as
        select c
        from $cName c;

    // insertElement: inverse-attrib-Name is a collection so insert the
    // source-Object into the inverse-attrib-Name
    for all source-Object in extents ($source-Class):
        for all obj in source-Object.$source-attrib-Name:
            obj.$(inverse-attrib-Name).insertElement (source-Object);

    // promote to relationship
    form_relationship ($source-Class, $source-class-Name,
                       $inverse-Class, $inverse-class-Name);
}
```