OQL_SERF: An ODMG Implementation of the Template-Based Schema Evolution Framework *

Kajal T. Claypool, Jing Jin, and Elke A. Rundensteiner
Department of Computer Science, Database Systems Research Group
Worcester Polytechnic Institute, Worcester, MA 01609–2280
{kajal|jing|rundenst}@cs.wpi.edu

Abstract

With rapid progress in application development and technologies, there is an increasing need to specify and handle complex schema changes of databases. The existing support for schema evolution in current OODB systems is limited to a pre-defined taxonomy of simple schema evolution operations with fixed semantics. We have proposed an extensible framework for schema transformations to address this open problem. The SERF framework succeeds in giving the user the *flexibility* to define the semantics of their choice, the *extensibility* of defining new complex transformations, and the power of *re-using* these transformations through the notion of templates. In this paper, we now report on OQL-SERF, the realization of our concepts based on the ODMG standard. We have utilized not only the ODMG object model, but have also used OQL as the database transformation language, the ODMG Metadata Repository for providing meta information utilized by the templates and Java's binding of OQL. We have developed a taxonomy of schema evolution primitives for the ODMG object model that can be shown to be *minimally complete* in the context of the SERF framework. The use of the ODMG standard as foundation for this implementation makes the implementation itself portable across different platforms and OODB systems.

*This work was supported in part by several grants from NSF, namely, the NSF NYI grant #IRI 94-57609, the NSF CISE Instrumentation grant #IRIS 97-29878, and the NSF grant #IIS 97-32897. We would also like to thank our industrial sponsors, in particular, IBM for the IBM partnership award and for the IBM Corporate Fellowship to one of the students in our DSRG research group for mentorship through the IBM Toronto CAS center. Special thanks also goes to the PSE Team at Object Design Inc., specifically, Gordon Landis, Sam Haradhvala, Pat O'Brien and Breman Thuraising 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

With current database technology, object-oriented database systems (OODBs) can support very complex object models like the ODMG object model [4]. These complex object models have paved the road for modelling dynamic applications which by their very nature have frequent schematic changes and upgrades [17].

Unfortunately, the existing support for schema evolution provided by current OODBs [1, 18, 3, 9, 8] is limited to a pre-defined taxonomy of *simple fixed-semantic* schema evolution operations. However, such simple changes, typically to individual types only, are not sufficient for many advanced applications [2]. More radical changes of the schema, such as combining two types or redefining the relationship between two types, are very difficult or even impossible to achieve with the current commercial database technology [18, 3, 9, 8]. In fact, it typically requires the user to write throw-away programs to accomplish these. Research has been done recently towards providing evolution support for such complex changes [2, 13]. However, this new work is again limited to providing a *fixed* set of now more complex operations with *fixed* semantics.

The provision of any fixed set, may it be simple or complex, is not satisfactory, as it would be very difficult for any one user or system to pre-define all possible semantics and all possible transformations that could be required by a user in the future. Our SERF [1] framework [5] addresses this by introducing the concept of *schema transformations* that use a database

[1]SERF stands for **S**chema evolution through an **E**xtensible, **R**e-usable and **F**lexible framework.

transformation language to combine schema evolution primitives into more complex type manipulations and to describe object transformations. These *schema transformations* are then generalized in our framework as *transformation templates* such that they are applicable to any schema and thus are re-usable for building new transformations. The SERF framework [5] gives users:

- The *flexibility* to define the transformation semantics of their choice.
- The *extensibility* of defining new complex transformations meeting user-specific requirements.
- The *generalisation* of these transformations to templates so as to be applicable to any schema.
- The *re-usability* of a template from within another template.
- The *ease* of template specification by programmers and non-programmers alike.
- The *soundness* of these user-defined transformations in terms of assuring schema consistency.
- The *portability* of these transformations across OODBs as libraries.

In order to validate this proposed concept of SERF transformations [5], we now set out to develop a working system, called OQL-SERF. OQL-SERF will serve both as proof of concept as well as to explore the suitability of the ODMG standard as the foundation for a template-based schema evolution framework. Our OQL-SERF development is based on the ODMG standard which today is the only source for a reliable basis to develop open OODB applications. ODMG holds 90% of the existing commercial OODB market and is fast becoming the standard for OODB systems [4]. The ODMG standard defines an Object Model, a Schema Repository, an Object Query Language (OQL) as well as a transaction model for OODB systems (see Section 5).

As demonstrated in this paper, OQL-SERF uses the ODMG standard in its entirety. It uses an extension of Java's binding of the ODMG model as its object model, our binding of the Schema Repository for its Metadata Dictionary, OQL as its database transformation language. OQL-SERF uses Object Design Inc.'s PSE as its persistent store. PSE, however, has limited facilities in terms of schema evolution. Thus, as part of the OQL-SERF implementation effort we have defined the invariants for preserving the ODMG Object Model and also a set of schema evolution primitives that preserve these invariants.

In summary, in this paper we present:

- Axioms of Preservation - the invariants for preserving the ODMG object model under schema evolution.
- Taxonomy of Schema Evolution Primitives - we have a complete set of ODMG-based schema evolution primitives such that they have minimal semantics and a combination of them is able to describe a large set of transformations.
- Development of the first 100% Pure Java Schema Evolution Manager - we have designed and implemented a dynamic schema evolution facility for *PSE Pro 2.0* that in-place modifies the types and the objects [2].
- Development of an OQL Query Engine - have designed and are in the process of implementing an OQL Query Engine for *PSE Pro 2.0* using JavaCC, JTB and the Visitor design pattern.
- Software Engineering Challenges - the requirement analysis of the SERF concept, the choices and the design decisions we had to make in the process of developing this system so as to make it re-usable and extensible to other systems.
- Development of OQL-SERF - design and implementation of OQL-SERF fully based on ODMG, that is, the Object model, OQL and the Schema Repository, as a proof of concept for the SERF framework. This also shows the portability of the SERF Framework to any ODMG compliant OODB system.

[2] The dynamic schema evolution facility, the OQL Query Engine for *PSE Pro 2.0* and OQL-SERF will be available for download from our web site http://davis.wpi.edu/dsrg/OOSE/SERF .

The rest of the paper is organized as follows. Section 2 presents some related work. Section 3 presents the SERF framework. Section 4 gives the SERF architecture and the requirements for an OODB system. Section 5 presents the relevant parts of the ODMG standard, i.e., the ODMG Object Model, the Schema Repository and the Object Query Language. Section 6 describes the invariants for the ODMG object model and its evolution. Section 7 describes the implementation of OQL-SERF using the ODMG Object Model and *PSE Pro 2.0.* We conclude in Section 8.

2 Related Work

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. [1]. They defined consistency and correctness of these primitives in the context of the Orion system. Until now, current commercial OODBs such as Itasca [9], GemStone [3], ObjectStore [8], and O_2 [18] all essentially handle a similar set of fixed evolution primitives; though based on their own respective object models.

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 [2, 13] have investigated the issue of more complex operations. Lerner [13] 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 [2] proposed a similar list of complex evolution operations for O_2, i.e., now considering both schema as well as object changes. [2] 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.

In summary, all previous research in this area tends to provide the users with a *fixed* set of schema evolution operations [1]. Very limited provisions, such as custom programs, are made for the situation where this does not meet the user's specific needs. How to add extensibility to schema evolution is the focus of our effort.

In 1991, Cattell set up the Object Database Management Group (ODMG) to standardize the object models used by different object database vendors. Today, although there are over 20 members in ODMG consortium, to the best of our knowledge we are the first to look at providing an extensible ODMG-compliant schema evolution framework. OQL-SERF focuses on an implementation of the SERF framework based on the ODMG standard.

Peters and Ozsu [16] have introduced a sound and complete axiomatic model that can be used to formalize and compare schema evolution modules of OODBs. This is the first effort in developing a formal basis for schema evolution research, and we utilize their notations for the description of our invariants and primitives.

3 The SERF Framework

The SERF Framework addresses the limitation of current OODB technology that restricts schema evolution to a *predefined* set of simple schema evolution operations with *fixed* semantics. With the SERF framework we can now offer *arbitrary user-customized* and possibly *very complex* schema evolution operations such as *merge*, *inline* and *split*. Moreover, for each transformation itself there can be many different semantics. For example, two classes can be merged by doing a union, an intersection or a difference of their attributes. Similarly the deletion of a class that has a super-class and several sub-classes can be accomplished by either propagating the delete of the inherited attributes through all sub-classes, or by moving the attributes to the super-class or by moving them down to all the sub-classes or any composition of the above semantics. The SERF framework achieves these *complex* schema evolution transformations based on the idea that three key ingredients, namely:

- the set of schema evolution primitives provided by the underlying OODB system,
- the Schema Repository exposed by the underlying OODB system and
- the query language supported by the OODB system

can be combined together to transform both the schema and the objects through a *transformation* as shown in Figure 2.

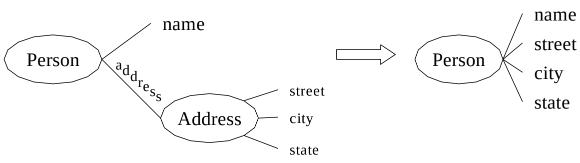

Figure 1: Example of an Inline Transformation

```
// Add the required attributes to the Person class
add_attribute (Person, Street, String," ");
add_attribute (Person, City, String," ");            Step A
add_attribute (Person, State, String," ");

// Get all the objects for the Person class
define extents() as
   select c                                          Step B
   from Person c;

// Update all the objects
for all obj in extents():
   obj.set (obj.Street, valueOf(obj.address.Street));   Step C
   obj.set (obj.City, valueOf(obj.address.City));
   obj.set (obj.State, valueOf(obj.address.State))

// Delete the address attribute                      Step B
delete_attribute (Person, address);
```

Figure 2: Inline Transformation

We illustrate the steps involved in a schema evolution transformation using the example of *Inlining* which is defined as the replacement of a referenced type with its type definition [13]. For example in Figure 1 the `Address` type is inlined into the `Person` class, where all the attributes defined for the `Address` type (the referenced type) are now added to the `Person` type resulting in a more complex `Person` class. Figure 2 shows the *Inline* transformation expressed in our framework using OQL, schema modification primitives, and system-defined update methods. In this example, the object.set () methods are the system-provided update methods.

In general a transformation has three steps [3]:

[3] Each step can be composed of or inter-mingled with the other steps. For example, **Step C** also involves the query of objects. We use the three key steps to denote the primary functionality of the step.

- **Step A: Change the Schema.** We require that all structural changes, i.e., changes to the schema, are exclusively made through the schema evolution primitives. This helps us in guaranteeing the schema consistency after the application of a transformation [5]. For example, **Step A** in Figure 2 shows the addition of the attributes `Street, City` and `State` via the *add_aattribute* schema evolution (SE) primitive to the `Person` class.
- **Step B: Query the Objects.** As a preliminary to performing object transformations, we need to obtain the handle for objects involved in the transformation process. This may be objects from which we copy object values (e.g., `Address` objects in **Step B**), or objects that get modified themselves (e.g., `Person` objects in **Step C**).
- **Step C: Change the Objects.** The next step to any schema transformation logically is the transformation of the objects to conform to the new schema. Through **Step B**, we already have a handle to the affected object set. **Step C** in Figure 2 shows how a query language like OQL and system-defined update methods, like *obj.set*(...), can be used to perform object transformations.

In general, a transformation in our SERF framework uses a query language to query over the application objects, as in **Step B**. The transformation also uses the query language to invoke the schema evolution primitives for schema structure changes and the system-defined functions for updating the objects, as in **Steps A** and **C**.

A SERF transformation as given in Figure 2 allows a user to define different semantics for any type of schema transformation. However, these transformations are not re-usable across different classes or different schemas. For example, the *inline* transformation shown in Figure 2 is valid only for the class `Person` and the class `Address`. To address this, in the SERF framework we have introduced the notion of templates [5]. A template uses the query language's ability to query over the meta data (as stated in the ODMG Standard), a name and a set of parameters in an effort to generalize the template.

Figure 3 shows a templated form of the transformation presented in Figure 2. The section of the template marked as **Step D** shows the steps required to achieve the effect of **Step A** in Figure 2 in a general form. Thus when this inline template shown in Figure 3 is instantiated with the variables `Person` and `address` it results in the SERF transformation shown in Figure 2.

A template is thus an arbitrarily complex transformation that has been encapsulated and generalized via the use of the Schema Repository, a name and a set of parameters. These templates can then be collected in a template library, guaranteeing the availability of these templates to any user at any time, just as the fixed set of schema evolution operations is available to the users in any regular schema evolution system.

```
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_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
```

Step D

Figure 3: Inline Template

4 Towards an ODMG Compliant SERF System

4.1 System Architecture

Figure 4 gives the general architecture of the SERF framework [5]. The components listed on the top half of the figure make up the SERF framework and thus are to be provided by any implementation realizing the framework. The components listed below the line represent system components that we expect any underlying OODB system to provide.

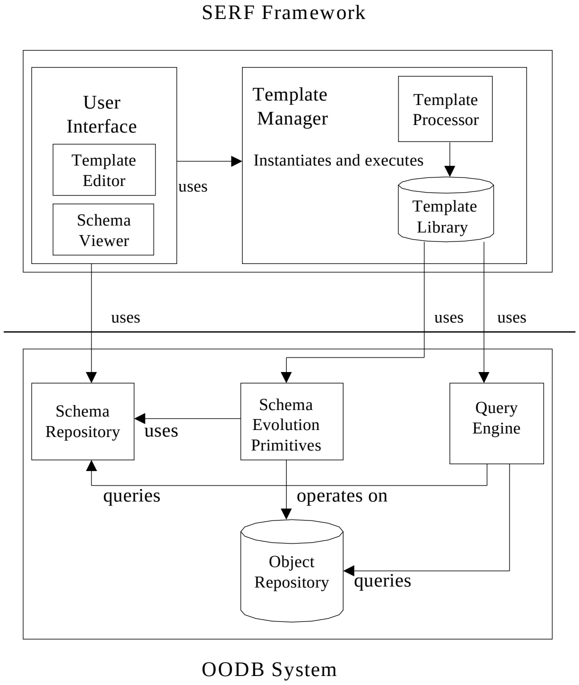

Figure 4: Architecture of the SERF Framework

Figure 4 also shows the interaction of the various modules during the execution of a template. In general, a template [4] uses a query language to query over the schema repository, i.e., the metadata [5] and the application objects. The template also uses the query language to invoke the schema evolution primitives for modifying the schema types, and system-defined functions for updating the object instances. These interactions are captured in Figure 4 by the directed arcs between the *Template Module*, the *Query Engine*, the *Schema Repository*, the *Schema Evolution Manager* and the *Object Repository*.

4.2 System Requirements for the SERF Framework

The SERF framework imposes the following minimum requirements on the underlying

[4] Although we distinguish between a transformation and a template, unless explicitly stated we use the term template to refer to both.

[5] More details on the schema repository are presented in Section 7.2.1

OODB system (see also the bottom half of Figure 4):

- **Schema repository.** For a transformation (as shown in Figure 2) to be generalizable to a template (as shown in Figure 3) we need to be able to query and access the metadata in some manner. For example, to merge two source classes into a single merge class by doing a union of the properties of the source classes, we need to get the properties of the source classes and add them to the merge class. In general to be able to get this information in a template, our framework requires access to the metadata. Most OODB systems indeed do allow access to the meta data, i.e., the data dictionary, and in most cases this is via some high-level declarative interface like a query language instead of just a procedural low-level API.

- **Taxonomy of schema evolution primitives.** The OODB needs to provide a set of schema evolution primitives that is complete [6] and consistent [7] [1]. Most OODB systems provide a taxonomy of schema evolution primitives and have invariants defined for preserving the consistency of the schema graph. The set of schema evolution primitives used is dependent on the underlying object model [1, 18].

- **Query language.** A transformation is a sequence of statements that gather metadata information, execute schema evolution primitives and invoke system-defined methods for object manipulations. There is therefore a need for a uniform language to access, query and modify both the metadata information and the application objects. For the SERF framework, we propose the use of a *declarative language*, namely the query language of the OODB system itself, to accomplish this task, thus requiring the query language to have the expressibility power to do all of the above. In particular, for the SERF Framework we require the query language to provide:
  - simple-to-use access to the OODB system,
  - constructs to invoke the schema evolution primitives,
  - support for creating, deleting and modifying objects either through some built-in features as in SQL or through the invocation of system-defined update methods, and
  - support for universal and existential quantification.

[6] By *complete* we mean a complete set of schema evolution primitives as defined by Banerjee et al. [1] in terms of achieving all possible basic types of schema graph manipulations.

[7] Consistent for SE primitives implies that the SE primitives preserve the set of invariants defined for the data model.

In an object-oriented realization of the SERF framework, we in addition require the query language to support object-oriented features such as complex objects, object identity, path expressions, polymorphism, etc.

In the remainder of this paper, we present OQL-SERF, an object-oriented implementation of the SERF Framework based on the ODMG standard. Section 5 highlights the features of the ODMG Standard that meet the system requirements of the SERF framework as outlined above. However the SERF Framework is not limited to relational or object technology nor is it limited to any particular object model, that is, it's application to other data models is possible as along as the above requirements are being met.

5 ODMG Standard

We have selected the ODMG standard, based on the continuing work for OODB systems undertaken by the members of the Object Database Management Group (ODMG) as the model basis of our work because of the its wide acceptability as a standard for object databases. We will show later that it meets all the requirements of the SERF framework. The major components of the ODMG standard as applied to the SERF framework are described in the subsequent sections.

5.1 ODMG Object Model

The ODMG Object Model is based on the OMG Object Model for object request brokers, object databases and object programming languages [4, 6]. For the purpose of the SERF framework we limit our description of the ODMG Object Model to Java's binding of the object model.

Types, Objects and Literals. The basic modelling primitives for an ODMG compliant database are *objects* and *literals or immutable objects* that are categorized by their *types* implying that there are *object types* and *literal types*. Each object has a unique object identifier that persists through the lifetime of the object and serves as a means of reference for other objects. Literals, on the other hand, do not have object identifiers and a change to a literal results in a new literal. An object can optionally also have one or more user-assigned name(s) and this is termed as *persistence by reachability*.

Inheritance. Although ODMG defines multiple inheritance, Java's binding of the ODMG model supports only single inheritance. A subclass therefore inherits the range of states and behaviour from its superclass. Moreover, an object can be considered as an instance of its class as well as its superclass(es).

Extent of Types Although Java's binding of the ODMG model does not as yet support the notion of extents, we have found it to be a necessary extension to the binding.

ODMG Schema A schema is composed of a set of object and literal type definitions, a class hierarchy and a set of named objects.

5.2 ODMG Schema Repository

Metadata is descriptive information that defines the schema of a database. It is used by the OODB system at initialization time to define the structure of the database and at run-time to guide its access to the database. Metadata is stored in a *Schema Repository*, which is also accessible to tools and applications using the same operations that apply to user-defined types, like OQL.

The Schema Repository is stored in the form of *meta-objects* interconnected by relationships that define the schema graph. A database schema, the types and the properties of these types all exist in the Schema Repository as meta-objects. For example, a class `Person` and an attribute `name` are meta-objects. Most meta-objects have a defining scope that gives the naming scope for the meta-objects in the repository. For example, the defining scope for `Person` would be its defining schema and the defining scope for `name` would be `Person`. In addition to this, the schema repository also contains the relationships between the meta-objects that define the schema graph.

5.3 ODMG's Object Query Language - OQL

As part of its standard, ODMG has defined an object query language OQL that supports the ODMG data model. OQL is similar in format and features to SQL 92 but has extensions for some object-oriented notions like complex objects, object identity, path expressions, polymorphism, operation invocation and late binding. In this section we describe a small subset of the language that is used for the examples in the paper. For a complete description of OQL the reader is referred to [4].

Selection. As a stand-alone query language, OQL supports the querying over *any* kind of object (i.e., individual object instances, collections and even the metadata in the schema repository) starting from their names which act as entry points to the database. OQL supports querying with and without object identifiers. For example, if the schema defines the types `Person` and `Employee` with extents `Persons` and `Employees` then we can query `Persons` as follows:

```
select distinct x.age
from Persons x
where x.name = ''Pat''
```

This selects the set of ages of all persons named Pat, returning a literal of type set<integer>.

```
select x
from Persons x
where x.name = ''Pat''
```

This selects all persons with the name Pat, returning a literal of type set<Person> where each Person object in the resultant set has the same object identifier as that in the database.

Creation. OQL supports the creation of objects both with and without identity. For example, `Person(name: "Pat", birthdate:"3/28/95",salary:10000)` creates an instance of the type `Person` using the `Person` type constructor. This constructs a new `Person` object with a new object identifier. Similarly, `struct (name:"Pat", birthdate:"3/28/95", salary:10000 )` yields a structure with three fields but no object identity.

Path Expressions. The ODMG object model as mentioned in Section 5.1 supports the naming of objects and also the reachability of other objects through this named object (i.e., persistence by reachability). From OQL, one therefore needs a way to *navigate* from a named object and reach the right data. For example, the query `p.spouse.address.city.name` starts from a `Person`, gets her spouse, a `Person` again, goes inside the complex attribute of type `Address` to get the `City` object whose `name` is then accessed.

Method Invocation. OQL can call a method with or without parameters anywhere the result type of the method matches the expected type in the query. For example,

```
select p.oldest-child.address
from Persons p
where p.lives-in(``Paris'')
```

In this statement we retrieve the `address` of the `oldest-child` of all `Persons` who live in `Paris`. Here `oldest-child` is a method that takes no parameters and returns an object of type `Person`. The method `lives-in` takes one parameter of the type `String` and returns `true` if the `oldest-child` lives in `Paris` and false otherwise.

6 Evolving the ODMG Object Model

Today, some support for schema evolution is provided by most OODB systems [1, 18, 3, 9, 8]. This support is generally in the form of a pre-defined taxonomy of *simple fixed-semantic* schema evolution operations. While *PSE Pro 2.0* supports the ODMG model and does provide support for schema evolution through a stream mechanism, it does not offer the dynamic evolution of the ODMG object model. As a step towards extending the schema evolution for *PSE Pro 2.0*, we have developed a taxonomy of schema evolution primitives for the ODMG object model such that they preserve the object model. In this section we first present the invariants for preserving the ODMG object model and then the schema evolution primitives that preserve these invariants and hence the object model.

6.1 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 [1]. The consistency of a schema is defined by a set of so called *schema invariants* of the given object data model [2]. In this section, we present the invariants for the ODMG object model. We have adapted the axiomatic model proposed by Peters and Ozsu [16] in order to axiomatize the schema changes for the ODMG object model.

As described in Section 5.1 a *type* in an object model defines the *properties* of its associated objects. Most object models support the notion of *sub-typing* of these *types*. Typical schema changes like adding and dropping of types, adding and dropping of subtype relationships, and adding and dropping type properties can affect the system integrity. To maintain the schema integrity through these changes we now define some axioms that must be maintained by any schema evolution primitive attempting to change the structure of the schema.

Table 1 shows the notation we use for describing the axiomatic model. In the table, *native* properties *Nt* 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. For the ODMG model, a `type` defines `properties` of objects. Although ODMG defines a `property` as `attributes or relationships` here we consider a `property` to

Term	Description
$\mathcal{T}$	All the types in the system
$s,\ t,\ T,\ \perp$	elements of $\mathcal{T}$
Pt	The immediate supertype of type t
Nt	The native properties of type t
Ht	The inherited properties of type t

Table 1: Notation for Axiomatization of Schema Changes

be only an `attribute` [8].

Axiom of Rootedness. There is a single type `T` in $\mathcal{T}$ that is the supertype of all types in $\mathcal{T}$. The type `T` is called the *root* [9].

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

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

Axiom of Nativeness. The native properties of a type `T`, *Nt*, is the set of properties that are locally defined within a type.

Axiom of Inheritance. The inherited properties of a type `T`, *Ht*, is the union of the inherited and native properties of its immediate supertype *Pt* .

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

6.2 Taxonomy of Schema Evolution Primitives

In this section we present the taxonomy of schema evolution primitives that we have designed for the ODMG object model such that they preserve the invariants introduced in Section 6.1. Our goal is to achieve a set of schema evolution primitives that is:

- Complete, i.e., our primitive set subsumes every possible type of structural schema change.
- Minimal, i.e., none of the primitives can be achieved by a combination of the other primitives.
- Simple, i.e., each primitive has minimal semantics such that most decisions are elevated to the user level.
- Consistent, i.e., each primitive generates only a valid schema.

Our schema change taxonomy is as follows:

1. Changes to the components of a type
   (a) Changes to class properties
      i. *add-attribute:* Adds a new property to the type.
      ii. *delete-attribute:* Delete a property from the type.
   (b) Changes to the inheritance graph
      i. *add-IS-A-edge:* Add a new supertype/subtype relationship [10].
      ii. *delete-IS-A-edge:* Delete a supertype/subtype relationship [11].
2. Changes to the types
   (a) *create-class:* Add a new type
   (b) *drop-leaf-class:* Delete a type
   (c) *rename-class:* Change the name of type

Compared to the Orion schema evolution taxonomy [1], we have kept the schema changes *create-class*, *rename-class*, *add-attribute*, *delete-attribute*, *add-IS_A-edge* and *delete-IS_A-edge* as primitives in our basic set. We have excluded the schema change *change-name-of-attribute* from our primitive set due to our minimality requirement, because it can be achieved by the composition of two other schema evolution primitives. Namely, *change-name-of attribute* can be accomplished by a sequence of first *add-attribute* with the new name and then *delete-attribute* with the old name and with the intermediate operation of copying values from one attribute to another.

[8] This is because the Java binding of ODMG does not support relationships as yet.

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

[10] Because of the single inheritance enforced by Java's binding of the ODMG model , this applies for only those types that inherit directly from the *root*.

[11] When the inheritance relationship between two types is removed, the subtype is added to the *root*. This is preservance of the *Axiom of Rootedness*.

For this reason, the *change-name-of-attribute* is not a fixed predefined primitive in our framework. For the same reason, we have excluded *retype-attribute* from the taxonomy. This can be achieved by a sequence of first *add-attribute* with a new name and a new type and then copying or casting the values from the attribute of the old type to the new attribute with the new type, followed by *delete-attribute* with the old attribute, creating another attribute with the old attribute name and the desired type and then performing a copy of values. As a last step the temporary new attribute and all its values are deleted. While minimal, we recognize the inefficiency of such an approach and are investigating optimization techniques for templates in general that can help us address this problem [14].

We have replaced the schema change *drop-class* from the Orion taxonomy with the *destroy-leaf-class* operation that removes a leaf class that has no local attributes as per our minimal-intrusion requirement. The *destroy-leaf-class* can be achieved through a SERF template by applying the *delete-attribute* primitive to all locally defined attributes followed by our *drop-class* primitive [11]. Using the above arguments together with the base proof given by Banerjee [1] it can be shown that the above taxonomy of schema evolution primitives is *minimally complete* in the context of the SERF framework.

7 Design and Implementation of OQL-SERF

In this section we present our implementation of the SERF Framework - OQL-SERF. OQL-SERF is built using Object Design Inc.'s Persistent Storage Engine Pro 2.0 (PSE Pro 2.0) as the underlying OODB system. It is based on the ODMG standard and is written in 100% Pure Java. In particular, we have used an extension of Java's binding of the ODMG object model and we have built our own binding of the ODMG Schema Repository using Java [4].

7.1 System Architecture

Figure 4 presents the system design for OQL-SERF using PSE Pro 2.0 as the underlying OODB system [15]. PSE, a lightweight OODB system written in Java [15], is an ideal choice for SERF due to its portability to different platforms and the availability of a free version. PSE also conforms to Java's binding of the ODMG object model.

In a persistent storage system, like PSE, it is assumed that the schema representation, data, applications and the links between them are all held as objects in persistent storage. While PSE offers most OODB features, it does not explicitly define a Schema Repository as per the ODMG standard. It also does not have the requisite schema evolution support and nor does its query interface meet the requirements of a query language for the SERF framework.

Thus as part of the OQL-SERF implementation we have addressed these limitations and built:

- A fully operational ODMG compliant schema repository.
- A complete schema evolution facility that preserves the ODMG object model and does changes at a lower granularity, thus allowing for in-place evolution.
- A fully functional subset of the OQL Query Engine for querying the objects in PSE.

In Section 7.2, we present the challenges that we faced in doing the design and implementation of these modules and the lessons we learned from the experience. The implementation of the System Modules provides the OODB support as per the SERF requirements (refer Section 4.2). With this in place, the main implementation of the SERF system focuses on the Framework Modules, i.e., the Template Module and the User Interface as described in Section 7.3.

7.2 OO Basis of OQL-SERF

7.2.1 Schema Repository

PSE Pro 2.0 provides some data dictionary support. It has a `ClassInfo` for each persistence capable class and each of these classes needs to be registered with the PSE database at the time of creation. Although we can access and use the `ClassInfo` class it is does not have support for schema evolution. As we are unable to modify the `ClassInfo`, we have implemented our own binding of the ODMG data dictionary called the Schema Repository and have consolidated some of its functionality with the PSE data dictionary. In our implementation the Schema Repository stores and manages metadata that defines the schema of a database. The instances of the `ClassInfo` are used by PSE at runtime to guide its access to the database [15].

Figure 5 shows the OQL-SERF Schema Repository which is our binding to the ODMG specification for the same. There are two main kinds of system classes, i.e., called the *MetaSchema* and the *MetaObject*. The *MetaClass* and the *MetaProperty* are specialisations of the *MetaObject* and represent an application class and properties respectively. An instance of the *MetaSchema* is synonymous with an application schema. Instances of the *MetaClass* model the application classes and the defining scope of these classes is the application schema instance for which they exist. Instances of the *MetaProperty* are the properties of the application classes and the defining scope for these is an instance of the *MetaClass*. All instances of the *MetaSchema, MetaClass* and *MetaProperty* are stored in the PSE database and all of them together define a particular application schema.

Figure 5 also shows the partitioning of the database space into the *MetaSpace* and the *Application Space*. The *MetaSpace*, represented by the Schema Repository in OQL-SERF, manages the *Application Space*, i.e., each instance of the *MetaClass* translates into a java class definition and instances of the *MetaProperty* bind to the properties that are part of a class definition.

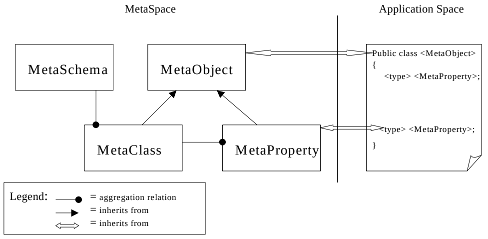

Figure 5: The Design of the Schema Repository for OQL-SERF.

7.2.2 Schema Evolution Manager

The Schema Evolution Manager provides an interface for the execution of the set of schema evolution primitives as described in Section 6.2. It interacts with the schema repository which contains information on each class and its placement in the class hierarchy. It also updates the PSE `ClassInfo` so as to keep it in sync with our Schema Repository as shown in Figure 5. As in other OODB systems, we have assumed that all additions, deletions and modifications to the database schema happen through the schema evolution primitives that we have provided [12]. This implies that as part of the schema evolution process, the schema evolution manager creates, modifies or deletes the source code, i.e., the Java files corresponding to the instances of the *MetaClass* [13]. The schema manager is also responsible for the migration of objects from the existing (old) class definition to the changed class definition, thus keeping them updated and consistent with the schema change.

Thus the schema manager must modify both the structure and the objects according to the change specified. The schema evolution process for all primitives that change an existing type is done using the following steps:

- Step1: Change the Schema Repository. Create a new temporary instance of the *MetaClass* by augmenting or reducing the capacity of the original meta object (i.e., the original class) with the desired change.
- Step2: Create the Source. Create the Java class file for the temporary class. The .java file is a subset or superset of the original .java file.
- Step3: Compile the Source. Compile the `temporary.java` file.
- Step4: Annotate the Source. Annotate the `temporary.class` file to create the `temporary.classInfo` file such that it is compliant to the PSE requirements.
- Step5: Change the PSE dictionary. Register the existence of the `temporary.class` in the PSE data dictionary through the `temporary.classInfo` file.

[12] For OODB systems where the user can directly manipulate the source files, the OODB system still translates them into primitive changes on the database schema.

[13] We do not limit the user from editing these files manually for adding, deleting or modifying method definitions. The only limitation that we impose is of using our interface for the addition, deletion and modification of properties.

- Step6: Copy the Objects. Copy all objects of the original class to the temporary class such that the objects are now properly augmented or reduced by the change.
- Step7: Swap OIDs. Preserve the OIDs by making the new objects have the same OID as the respective original objects.
- Step8: Rename Class. Remove the original class and rename the new class to the original class name in both the Schema Repository and the PSE data dictionary.

The evolution steps for primitives at the class level (like *create-class, rename-class*, etc.) are similar but do not involve any object copying. The *drop-leaf-class* primitive although does not have any object copying but it has object deletions.

7.2.3 OQL Query Engine

Although *PSE Pro 2.0* provides a mechanism for querying collection objects, it is very basic (does not have joins) and does not meet our criteria for SERF transformation support. Thus as part of the implementation effort of OQL-SERF, we are implementing an OQL interface for PSE. Our goal in building an OQL Query Engine is to provide:

- a general-purpose OQL Query Engine for object databases,
- a binding of the OQL Query Engine for PSE Pro2.0,
- an API interface binding consistent with ODMG and Java's binding of ODMG for the applications using the OQL Query Engine.

Figure 6 shows how we envision the query engine being used by an application to query the underlying OODB system. An application would use the exposed interface of the query engine to formulate and execute OQL queries on the underlying database. The left hand side of Figure 7 shows the steps involved in the building of the OQL Query Engine. Our OQL Query Engine is based on the OQL grammar given in Chapter 4 of the ODMG Standard 2.0 [4].

The key functionality of the OQL Query Engine is to parse and map a given OQL statement

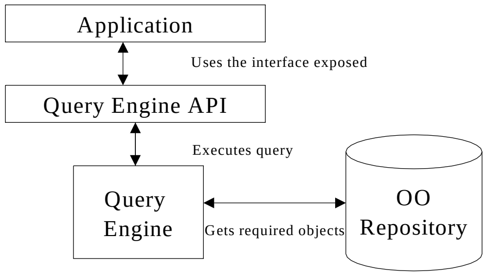

Figure 6: Design of OQL Query Engine for OQL-SERF.

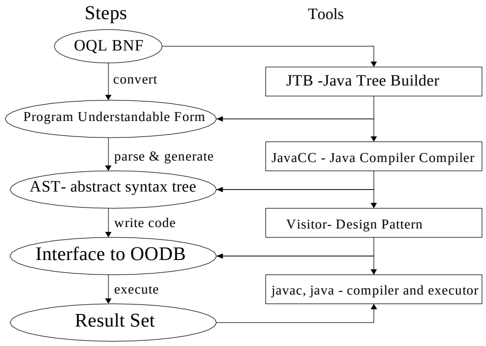

Figure 7: Steps and Corresponding Tools for Building the OQL Interface.

to the right OODB system functions in order to get the required result. To accomplish this, the OQL Query Engine must:

- **Parse** the OQL statement
- **Generate** the abstract syntax tree
- **Iterate** over the abstract syntax tree and collect the information from the query tree into appropriate data structures.
- **Execute** using the information in the data structures, call the system functions to retrieve the set of requested objects from the OODB.

We were fortunate to find a wealth of information and tools that provided us with the building blocks for the OQL Query Engine. The right side of the Figure 7 matches each step with the tool that was used to accomplish it. The Java Compiler Compiler (JavaCC) [10] is currently the most popular parser generator for use with Java applications. This tool reads the OQL grammar specification and converts it to a

Java program that can recognize matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building, embedded calls for actions, debugging, etc. In the past year since the release of JavaCC there have been many tools that have been built to enhance and compliment the JavaCC functionality like the Java Tree Builder (JTB). JTB [12] is a syntax-tree builder to be used with the JavaCC and automatically generates not just the set of syntax tree java class files but also a *Visitor* superclass whose methods allow us to visit the nodes of the tree.

Thus after using JTB and JavaCC tools on the OQL grammar, we have a functional OQL Parser. The *Visitor* generated by JTB as a default is our tool for doing the next step of the process, namely, the *iterate and execute* steps. The *Visitor* pattern is one among many design patterns aimed at making object-oriented systems more flexible [7]. The issue addressed by the Visitor pattern is the manipulation of composite objects. Thus, the Visitor pattern gives us the flexibility of rapidly re-targeting the OQL Query Engine to a number of OODB systems. In OQL-SERF, we have written a Visitor for *PSE Pro 2.0* using their existing collection query interface to take advantage of their collection optimization strategies such as indexes.

7.3 SERF Framework Modules

The SERF Framework Modules are the core components that need to be provided by any system realizing the SERF Framework. Sections 7.3.1 and 7.3.2 describes the core functionality and support needed for SERF templates and transformations.

7.3.1 Template Module

The Template Module provides all of the functionality for storing, retrieving of templates and for the execution of templates. Figure 8 shows the architecture for the Template Module in OQL-SERF.

The Template Manager can store, retrieve and edit templates in the Template Library. It provides the users with an interface for storing, retrieving and editing the templates in the Template Libraries. A Template Library

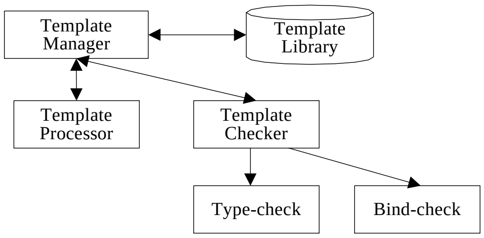

Figure 8: The Template Module.

is a package of templates and since the Template Manager can have more than one library, `libraryName.templateName` gives the complete path for a template object [14]. A template object itself is an instance of the TemplateClass (see Figure 9) and there is a one-to-one correspondence between this instance and the SERF template. An instance of the TemplateClass contains the *name*, *description*, a set of *input parameters*, a list of *keywords* and the *source* corresponding to the SERF template. When a user stores a template, they are required to furnish the *name* and the *input parameters* for the template.

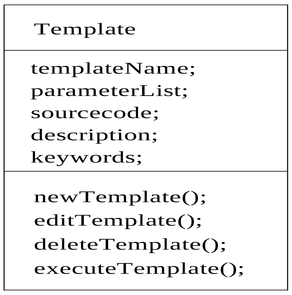

Figure 9: The Template Class.

The Template Manager also provides the users with an interface for searching through the templates. In our current implementation we support *search by name*, *search by complete or partial description*, and *search by keywords*.

The Template Manager also provides the users with an interface for the execution of the SERF template. The actual execution of the SERF Template is handled by the Template Processor. Figure 10 shows the steps performed by

[14] We distinguish between a template and a SERF template. Here a template object implies an instance of the TemplateClass and a SERF template is the source code that is part of this instance.

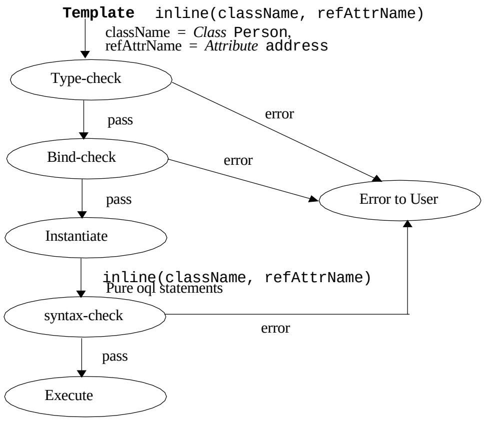

Figure 10: Steps for the Execution of a Template.

the Template Processor for the execution of a SERF template. The template processing begins with the user supplying the input parameters. These parameters are a particular `Class` or `Property` in the application schema to which the user wants to apply the SERF template transformation. A *type-check* ensures that the types of the parameters match as well as the correct number of parameters are supplied by the user. This is followed by a *bind-check* which checks the existence of these actual parameters in the schema on which they are being applied by accessing the Schema Repository. The SERF template is instantiated using these parameters by replacing each variable with its bound parameter after all the checks are completed successfully. The instantiated SERF template now corresponds to pure OQL statements, i.e., we now call it an OQL transformation. The OQL Query Engine provides an interface for the syntax-checking, the *parsing* and the *execution* of the OQL transformation (see Section 7.2.3).

7.3.2 User Interface

No software is usable unless and until there is an interface provided for it. The usability of the software is increased ten-fold with an easy-to-use graphical user interface. For OQL-SERF version 1.0, we have designed and are currently developing a GUI as a front-end to the functionality offered by the Template Manager. It provides a Template Editor with syntax highlighting that allows the user to create a new or edit an existing template. The Save option walks the user through the steps of providing the *name*, the *list of formal parameters*, the *description*, and the *keywords* for the SERF template object before it can be stored and managed by the template library. The GUI also allows the user to view the schema graph before and after a transformation has been applied to the schema thus giving the user a chance to verify if this was indeed the desired transformation.

8 Conclusions

In this paper we have presented the design and implementation of OQL-SERF, the first template-based framework for extensible schema evolution. We have used the ODMG standard as the foundation for this implementation so as to make the implementation itself portable across different platforms and OODB systems.

We have completed the implementation of most of the modules of the SERF Framework and are currently in the process of completing the implementation of the OQL Processor. We plan to release it to public domain via our website *http://davis.wpi.edu/dsrg/OOSE/SERF.*

Acknowledgements. We would like to thank students at the Database Systems Research Group at WPI for their interactions and feedback on this research. In particular, we are grateful to Anuja Gokhale, Parag Mahalley, Swathi Subramanian, Jayesh Govindrajan, Stacia De Lima, Stacia Weiner, Ming Li and Xin Zhang for their work on the implementation of OQL-SERF.

About the Authors.

Kajal T. Claypool and *Jing Jin* are graduate students at Worcester Polytechnic Institute. Their main interests lie in views, schema evolution, distributed systems and OO technology.

Elke Angelika Rundensteiner is currently Associate professor of the Department of Computer Science at the Worcester Polytechnic Institute, after having been a faculty member in the Department of Electrical Engineering and Computer Science at the University of Michigan. She has received a BS degree (Vordiplom) from the Johann Wolfgang Goethe University, Frankfurt, West Germany, an M.S. degree from Florida State University, and a Ph.D. degree from the University of California, Irvine. Dr. Rundensteiner has been active in the database research community for over 10 years now, and her current research interests include object-oriented databases, data warehousing for distributed systems, database and software evolution, multi-media and web database applications, and visual information interfaces. She has more than 100 publications in these areas. Her research has been funded by government agencies including NSF, ARPA, NASA, CRA, DOT; and by industry including IBM, AT&T, Intel, Informix and GE. She is on Program Committees of the key conferences in the database field such as IEEE ICDE, ACM SIGMOD, VLDB, as well as others. Dr. Rundensteiner has received numerous honors and awards, including a Fulbright Scholarship, an NSF Young Investigator Award, and an Intel Young Investigator Engineering Award, and an IBM Partnership Award.

References

[1] J. Banerjee, W. Kim, H. J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *SIGMOD*, pages 311–322, 1987.

[2] P. Bréche. Advanced primitives for changing schemas of object databases. In *Conference on Advanced Information Systems Engineering*, pages 476–495, 1996.

[3] 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*. ACM Press, 1989.

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

[5] K. Claypool, J. Jing, and E.A. Rundensteiner. SERF:Schema Evolution through an Extensible, Re-usable and Flexible Framework. In *Int. Conf. on Information and Knowledge Management*, 1998.

[6] S. Cluet. Designing OQL: Allowing objects to be queried.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

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

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

[10] JavaCC. http://www.sun.com/suntest/JavaCC/, 1996.

[11] J. Jing. An Extensible Schema Evolution Framework for Object-Oriented Databases using OQL. Master's thesis, Worcester Polytechnic Institute, May 1998.

[12] Java Tree Builder. http://www.cs.purdue.edu/homes/taokr/jtb, 1996.

[13] 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.

[14] C. Natarajan. CHOP: An Optimizer for Schema Evolution Operation Sequences. Master's thesis, Worcester Polytechnic Institute, June 1998.

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

[16] 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.

[17] 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.

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