WPI-CS-TR-98-14 July 1998

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

by

Kajal T. Claypool
Jin Jing
Elke A. Rundensteiner

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

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

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
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, SERF(**S**chema Evolution using an **E**xtensible **R**e-Usable **F**ramework) 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 on top of PSE (Persistent Storage Engine) Object Design Inc. 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 ODL. In order to design a schema evolver manager for ODMG, we had to develop a taxonomy of schema evolution primitives for the ODMG object model that is *minimally complete.* Related to the implementation of OQL-SERF on top of PSE (Persistent Storage Engine), we also describe the design and implementation issues involved in developing a java-based schema evolution manager as well as an OQL query engine for PSE. Our working prototype, OQL-SERF, demonstrates the capability of our SERF approach to handle a large set of schema transformations.

1 Introduction

With current database technology, object-oriented database systems (OODBs) can support very complex object models like the ODMG object model [Cea97]. These complex object models have paved the road for modelling dynamic applications which by their very nature have frequent schematic changes and upgrades [FFM$^+$95].

*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 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 ODI for not only software contributions but also for providing us with a customized patch of the PSE Pro2.0.2 system that exposed schema-related APIs needed to develop our tool.

Unfortunately, the existing support for schema evolution provided by current OODBs [BKKK87, Tec94, BOS91, IS93, Inc93] 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 [Bré96]. 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 [Tec94, BOS91, IS93, Inc93]. 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 [Bré96, Ler96]. 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 framework [CJR98] addresses this by introducing the powerful concept of *schema transformations*. A *SERF schema transformation* uses a database query language to integrate primitives for schema updates, meta-data retrieval for schema information access, object updates, and object manipulations to formulate a powerful script for schema restructuring. 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 [CJR98] thus gives users:

- The *flexibility* to define the transformation semantics of their choice.
- The *extensibilty* of defining new complex transformations meeting user-specific requirements.
- The *generalization* of these transformations to templates so as to be applicable to any schema.
- The *re-useability* 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 [CJR98], we now set out to develop a working system, called OQL-SERF, 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 [Cea97]. 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. It uses Object Design Inc.'s PSE as its persistent store which is a lightweight persistent storage engine and it is 100% pure Java. However, as PSE has limited facilities in terms of schema evolution, as part of the OQL-SERF implementation effort we first developed a schema evolution manager. This effort also involved the definition of the invariants for preserving the ODMG Object Model and the development of a set of schema evolution primitives that preserve these invariants.

The main contributions of this paper are:

- 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 schema transformations.
- Development of the first 100% Pure Java Schema Evolution Manager - have design and implementation of a dynamic schema evolution facility for *PSE Pro 2.0* [1].
- Development of an OQL Query Engine - have design and development of 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. Thus also showing the portability of the SERF Framework to any ODMG compliant OODB system.

The rest of the paper is organized as follows. Section 2 presents some related work and Section 3 presents the SERF framework. Section 4 gives the SERF architecture and the requirements for an OODB system. Section 5 presents the the relevant parts of the ODMG Standard, i.e., the ODMG Object Model, the MetaData and the Object Query Language. Section 6 describes the invariants for the ODMG object model and describes how evolution of this model can be done. Section 7 describes the implementation of OQL-SERF based on the ODMG Object Model and developed on top of *PSE Pro 2.0*. We conclude in Section 8.

2 Related Work

Schema evolution is a problem that is faced by long-lived data. The goal of schema evolution research is to allow schema evolution mechanisms to change not only the schema but also the underlying objects to have them conform to the modified schema. 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 [IS93], GemStone [BOS91], ObjectStore [Inc93], and O_2 [Tec94] all essentially handle a set of evolution primitives based on their own object models.

In recent years, the advent of more advanced applications has led to the need for support of complex schema evolution operations. [Bré96, Ler96, Cla92] have investigated the issue of more complex operations. [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*, *Move*, *Duplicate*, *Reverse Link* and *Link Addition*.

[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. Like other previous work, the paper however still provides a fixed taxonomy of primitives to the users, instead of giving them the flexibility, extensibility and customization 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.

[1] 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/OOSE/SERF.html.

In summary, all previous research in this area tends to provide the users with a *fixed* set of schema evolution operations [FFM+95, BKKK87]. No provision, other than for the user to write ad-hoc programs for a desired transformation, is made for the situation where this does not meet the user's specific needs. How to add extensibility to schema evolution is now the focus of our effort.

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

Peters and Ozsu [PO95] 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.

Other research has studied the issue of when and how to modify the database objects to address such concerns as efficiency, availability, and impact on existing code. Research on this issue has focused on providing mechanisms to make data and the system itself more available during the schema evolution process [Lau97a], in particular deferred and immediate propagation strategies [FMZ94b, FMZ94a]. In principle, either of these propagation strategies could be implemented for our framework.

Another important issue focuses on providing support for existing applications that depend on the old schema, when other applications change the shared schema according to their own requirements. Research to address this issue has followed along two possible directions, namely, views [RRL97, RR95, RR97, Ber92] and versions [SZ86, Lau97b]. Some of this on-going research may need to be re-examined in order to handle the complex notion of transformations as introduced by our templates.

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. It provides support for *arbitrary user-customized* and possibly *very complex* schema evolution operations. The SERF framework is 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 in a general and re-usable manner through a *transformation* as shown in Figure 1.

Figure 1 illustrates the transformation for *inlining* via an example. *Inlining* is defined as the replacement of a referenced type with its type definition [Ler96]. For example in Figure 2 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. We claim that in general a transformation accomplishes this using the following four key steps [2]:

[2] Note that each of these four steps are not pure but can often be composed of or inter-mingled with the other steps. For example, **Step D** also involves the query of objects. We use these four key steps to denote the primary functionality of the

```
refClass = element (
             select a.attrType
             from MetaAttribute a
             where a.attrName = address
             and   a.classDefinedIn = Person )
                                                       Step A
define AddressAttrs() as
   select c.localAttrList
   from MetaClass c
   where c.metaClassName = refClass;

for all attrs in AddressAttrs():
   add_atomic_attribute (Person, attrs.attrName,       Step B
                         attrs.attrType,
                         attrs.attrValue);

define extents() as
   select c                                            Step C
   from Person c;

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

delete_attribute (Person, address);                    Step B
```

Figure 1: The Inline SERF Transformation.

- **Step A: Query the MetaData.** To make a transformation general and re-usable for any possible schema in the form of a transformation template, it is necessary that a user be able to query the metadata using a query language. This information can then be used to make decisions about changes to the schema. In Figure 1 this step denoted by **Step A** which retrieves all the objects from the metadata that model the properties of the `Address` object.

- **Step B: Change the Schema.** All structural changes, i.e., changes to the schema, are exclusively made through the schema evolution primitives. This restriction helps us in guaranteeing the schema consistency after the application of a transformation [CJR98]. The information gathered in *Step A* can provide the metadata to be changed as well as the information needed for determining how to change the metadata, both serving as input to these SE primitives. For example, **Step B** in Figure 1 shows the addition of extra attributes through the *add_attribute* primitive to the `Person` class.

- **Step C: 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 C**), or objects that get modified themselves (e.g., `Person` objects in **Step D**).

- **Step D: Change the Objects.** The next step to any schema transformation logically is the transforming of the objects to conform to the new schema. Through **Step C**, we already have a handle to the affected object set. **Step D** in Figure 1 shows how a query language like OQL and system-defined

transformation.

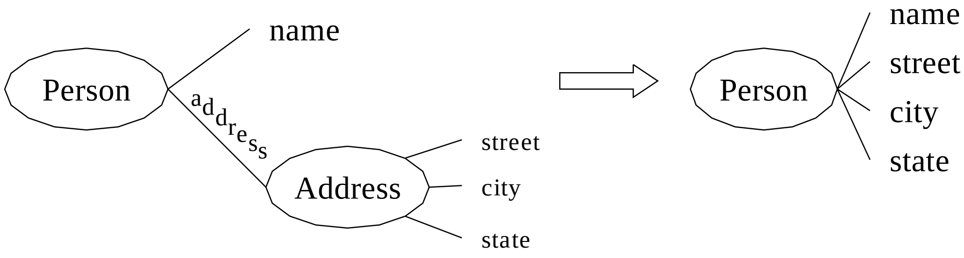

Figure 2: Example of an Inline Transformation

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 schema repository, i.e., the metadata and the application objects, as in **Steps A** and **C**. 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 B** and **D**.

A SERF transformation as given in Figure 1 allows a user to flexibly define schema transformations for a given schema, but to make them re-usable across different schemas SERF goes one step further and introduces the notion of templates [CJR98]. A template is an arbitrarily complex tranformation that has been encapsulated and generalized with a name and a set of parameters. By parameterizing the variables involved in a transformation such as the input and the output classes, e.g., the `Person` and `Address` classes in our example and their properties, e.g., the `address` attribute in our example, and assigning a name to the transformation e.g., *inline* in our example, a transformation becomes a *generalized reusable* module applicable to any application schema. Figure 3 shows the *inline* transformation of Firgure 1 as a template. Vice versa,a when this inline template as shown in Figure 3 is instantiated with the variables `Person` and `address` it results in the SERF transformation shown in Figure 1. 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.

4 Towards an ODMG Compliant SERF

4.1 System Architecture

Figure 4 gives the general architecture of the SERF framework [CJR98]. The components listed on the top half of the figure make up the framework and thus are to be provided by any implementation realizing the SERF framework. The components listed below the line represent system components that we expect any underlying OODB system to provide.

Figure 4 also shows the interaction of the various modules during the execution of a template. In general, a template [3] uses a query language to query over the schema repository, i.e., the metadata [4], and the application objects. The template also uses the query language to invoke the schema evolution primitives for modifying the schema types (schema updates), and system-defined functions for updating the object instances (data updates). These interactions are captured in the Figure 4 by arrows between the

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

[4]More details on the schema repository are presented in Section 4.2.

```
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 3: The Inline Transformation as a SERF Template.

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 for the system modules, as shown below the line in Figure 4, provided by the underlying OODB system :

- **Schema repository.** For a transformation (as shown in Figure 1) 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

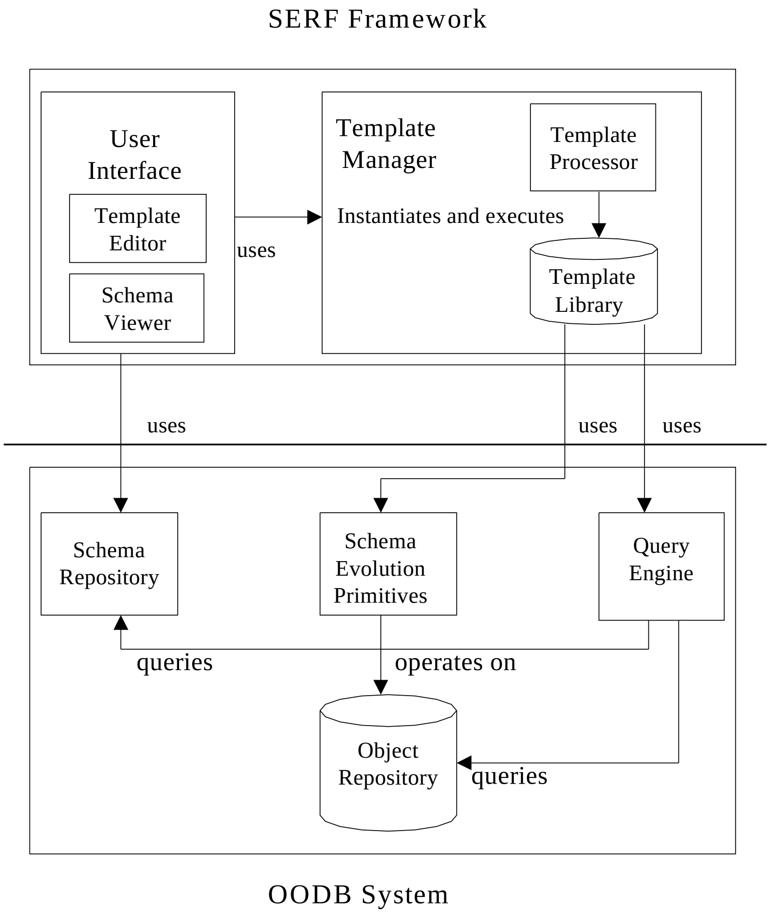

Figure 4: Architecture of the SERF Framework

OODB systems indeed do allow access to the metadata, 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 [5] and consistent [BKKK87]. 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 [BKKK87, Tec94].

- **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. Thus for the SERF Framework we require the query language to provide:

[5] By *complete* we mean a complete set of schema evolution primitives as defined by Banerjee et al. [BKKK87] in terms of achieving all possible basic types of schema graph manipulations.

- 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 inovocation of system-defined update methods,
- support for universal and existential quantification, for example:

```
for all x in Students: x.student_id > 0
```

and

```
exists x in Doe.takes:x.taught_by.name = ``Turing''.
```

In an object-oriented realization of the SERF framework, we also require the query language to provide not just the above but also to provide the object-oriented notions such as complex objects, object identity, path expressions, and polymorphism.

4.3 OQL-SERF: Why ODMG?

The analysis of the requirements for the underlying OODB system revealed the ODMG standard to be a perfect fit for the SERF Framework. The ODMG object model encompasses the most commonly used object models and standardizes the features into its own object model, thus increasing the portability and applicability of our prototype. ODMG also defines metadata and MetaObject Protocols in the shape of the ODL Schema Repository and explicitly states that this Repository should be accessible to tools and applications using the same operations that apply to the user-defined types [Cea97].

ODMG also defines a query language, OQL, as part of its standard that is a superset of SQL-92 interms of its querying capabilities. In addition OQL also provides programming-language-like constructs such as *for loop, iterators etc.*. And, although OQL does not have any built in features for updating objects, it has the capability to invoke system-defined functions. This implies that it can update objects through system-defined set methods and also it can invoke schema evolution primitives. OQL thus meets all the requirements of a query language as set forth by the SERF framework.

In this paper, we present OQL-SERF, an object-oriented implementation of the SERF Framework based on the ODMG standard. However the SERF Framework itself as presented in this paper is not limited to relational or object technology nor is it limited to any particular object model, but the basic SERF principles could be applied to such other environments as long as they meet the four requirements given in Figure 4.2.

5 ODMG Standard

The ODMG standard is based on the continuing work for OODB systems undertaken by the members of the Object Database Management Group (ODMG). 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 [Cea97, Clu98]. For the purpose of the SERF framework we limit our description of the ODMG Object Model to the Java's binding of the object model.

Types, Objects and Literals. The basic modeling primitives for an ODMG compliant database are *objects* and *literals or immutable objects* which are categorized by their *types* implying that there are *object types* [6] and *literal types*. All elements of a given type have a common range of state (i.e., the same set of properties), and common behavior (i.e., the same set of defined operations). Objects are also referred to as the *instances* of their type.

Object Identity. The identity of an object distinguishes it from all other objects in a database i.e., they are unique. The identity of an object is independent of its state and persists through the lifetime of the object. Object identity provides a means to reference objects and thus allows an object to be shared by others.

Literals do not have their own identifiers; they are characterized by their state. As literals are immutable objects a change in the state of the literal creates a new one.

Object Names. In addition to being assigned an object identifier by the OODB system, an object may be given one or more names that are meaningful to the user. The scope of uniqueness of names is the database. This is called *persistence by reachibility* and implies the existance of a garbage collector that deletes objects that are no longer referenced. A literal may also be given a name.

Inheritance. Although ODMG defines multiple inheritance, Java's binding of ODMG Model supports only single inheritance [7]. This implies that a user may define a class to be a subclass of only one other class. A subclass inherits the range of states and behavior from its superclass. Moreover, an object can be considered as an instance of its class as well as its superclass.

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 [8]. The extent of a class is the set of persistent objects belonging to that class. Once a class has been defined with extent, the system is in charge of managing the set of all its persistent instances. This implies that when an object is created the system inserts it into the extent of its direct type and vice versa when it is removed the system removes it from the extent of its direct type. The extent of a class is also implicitly included in the extent of the superclass.

ODMG Schema A schema is composed of a set of object and literal type definitions, a class hierarchy and a set of names for persistence by reachability.

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.

[6] The terms "class" and "type" are used interchangeably in the remainder of this paper.

[7] We deal with the *extends* relationship which does specilization of one class to another. At this point we do not deal with the *implements* relationship which does does inheritance from an abstract type (i.e., interface) to an implementation class.

[8] In the absence of support for extents, a user would have to either declare and maintain an explicit collection for each type or we would have to scan the entire space to retrieve an object of a particular type. Both these options are tedious and very time consuming for the user as well as the sytem. For this we found it necessary to implement extents.

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 which 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 inheritance relationships between the meta-objects which defines the schema graph. These relationships help guarantee the referential integrity of the meta-object graph.

5.3 ODMG's Object Query Language - OQL

As part of its standard, ODMG has defined an object query language OQL which 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 [Cea97].

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 two fields but no object identity.

Path Expressions. ODMG as mentioned in Section 5.1 supports the naming of objects and also the reachability of other objects through this named object (i.e., persistance 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 his/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 are trying to retrieve the `address` of the `oldest-child` of all those `Persons` who live in `Paris`. Here `oldest-child` is a method that takes no parameters but returns an object of type `Person` and thus we are trying to retrieve the `Person.address`. The method `lives-in` takes one parameter of type `String` and returns `true` or `false` depending on whether the `oldest-child` lives in `Paris` or not.

Although OQL does not have any explicit support for updating the objects this capability to invoke methods allows a user to invoke application-specific update methods through the query language.

6 Evolving the ODMG Object Model

Today, some support for schema evolution is provided by most OODB systems [BKKK87, Tec94, BOS91, IS93, Inc93]. This support typically is 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 [BKKK87]. The consistency of a schema is defined by a set of so called *schema invariants* of the given object model [Bré96]. In this section, we present the invariants for the ODMG object model. We have adapted the axiomatic model proposed by Peters and Ozsu [PO95] in order to axiomatize the schema changes for the ODMG Object Model[9].

6.1.1 Axiomatization of Schema Changes

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

[9]This is our adaptation of the model presented in [PO95].

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

Table 1 shows the notation we use for describing the axiomatic model. In the table, *native* properties *Nt* refer to the properties of `type` 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` we consider a `property` to be only an `attribute` [10] in the context of this paper.

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* [11].

Axiom of Closure. Types in $\mathcal{T}$, excluding the *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` is the set of properties that are locally defined within a type.

Axiom of Inheritance. The inherited properties of a type `T` is the union of the inherited and native properties of its supertype.

Axiom of Distinction. All types `T` in $\mathcal{T}$ have distinct names. Every property `p` for a type `T` has a distinct name. The scope of distinction for a property is the set of native properties for a type.

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 should subsume every possible type of 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 simple semantics so as to not embed semantics in the primitives.
- Consistent, i.e., each primitive generates a valid schema when applied to a valid schema.

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

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

Our schema change taxonomy is as follows:

1. Changes to the components of a type
   (a) Changes to class properties
      i. *add-attribute:* Add 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 [12]
      ii. *delete-IS-A-edge:* Delete a supertype/subtype relationship [13]
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

Like the Orion schema evolution taxonomy [BKKK87], 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. 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. We recognize the inefficiency of such an approach and are investigating some optimization techniques that can help us address this problem [Nat98].

We have replaced the schema change *drop-class* from the Orion taxonomy with the *destroy-leaf-class* operation which 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 [Jin98].

6.3 Completeness and Soundness of the Basic Schema Evolution Primitive Set

All the schema evolution primitives in our basic set are updates either to the attributes of a class or to the class as a whole. Each of them is an atomic operation with fixed semantics which cannot be decomposed any further. And, each of them transforms the schema from one structurally consistent state to another structurally consistent one [BKKK87, Zic91].

[12] For Java's binding of the ODMG model because of the single inheritance, this applies for only those types that inherit directly from the *root*.

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

Banerjee et al. [BKKK87] outlined a formal proof of the completeness of their schema evolution taxonomy and correctness of the semantics of schema changes. Their approach is based on a *property inheritance graph (PIG)* which is a single-rooted, directed acyclic graph (DAG) corresponding to a class hierarchy. In this formal model, they define eight PIG operations which correspond essentially to the schema changes in their taxonomy. They prove that every legal PIG is achievable using this set of eight operations (i.e., completeness). They also show that the basic set of operations cannot generate a directed acyclic graph that would violate the syntactic rules which characterize a PIG (i.e., soundness).

The basic PIG operation set used above includes operations we either directly support as primities such as *add-attribute*, *delete-attribute*, *add-edge*, *delete-edge*, *add-class*, *delete-class* and *rename-class* or that we indirectly support as SERF templates such as *change-attribute-name* and *drop-any-class*. The original semantics of all these schema updates have been preserved for our basic primitive set. For this reason, the proof of the completeness and soundness of our basic primitive which follows directly from their proof is omitted from here.

7 Design and Implementation of the OQL-SERF System

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 [Cea97].

7.1 System Architecture of the OQL-SERF System

Figure 4 presents the system design for OQL-SERF using PSE Pro 2.0 as the underlying OODB system. PSE Pro 2.0 is the first persistent storage engine written entirely in Java [Bri97] and runs within the same process as the Java applications or applets. The PSE Java client and the storage layer provide an easy-to-use interface for storing and retrieving persistent objects. As per ODMG, PSE provides *named objects* and offers persistence by reachability. For further details on PSE, we refer the user to [Bri97].

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, 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 enhanced PSE Pro 2.0 by providing:

- an operational ODMG-compliant schema repository.
- a complete schema evolution facility (based on the ODMG object model) that does dynamic class-level changes at run-time and in-place without requiring the lengthy approach of piping the database extent to a file, manually creating the new desired schema, and then reloading all data into the new schema.
- a fully functional subset of the OQL Query Engine for querying the objects stored in the OODB system.

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 realization of the OQL-SERF system consists of two major tasks: namely, the OODB System Modules and the SERF Framework Modules. The implementation of the OODB 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 SERF Framework Modules, i.e., the Template Module and the User Interface. Although at some point in the future we plan to enhance the OQL-SERF to provide help mechanisms such as advanced search mechanisms, more rigid template checking, etc., here we focus on presenting the core of the template module, the steps involved in its processing, its API interface and the user interface (Section 7.3).

7.2 OODB Systems Modules Required as Basis for 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 to some degree it is not fully accessible and modifiable by the SERF system. For this reason 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 which defines the schema of a database and the instances of the `ClassInfo` are used by PSE at runtime to guide its access to the database [Bri97]. 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 *MetaSchema* and *MetaObject*. The *MetaClass* and the *MetaProperty* are specializations of the *MetaObject* and represent application classes 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 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.

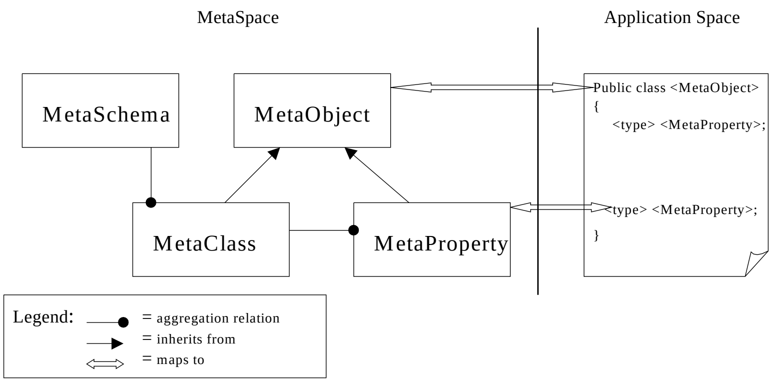

Figure 5: The Design of the Schema Repository for OQL-SERF

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. For our implementation, we have assumed that all additions, deletions and modifications to the schema happen through the interface that we have provided. 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* [14]. 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.

Consider, for example, the addition of the new attribute `DateOfBirth` to an existing class *Person* which should add this new attribute `DateOfBirth` of type `String` to the specified class `Person` and then augment the existing objects of `Person` to contain `DateOfBirth` with the default value. As a first step we update the Schema Repository by creating a copy `PersonTmpCopy` of the class `Person` with the new attribute `DateOfBirth`. A new source file (a Java file) `PersonTmpCopy.java` is created by collating the information in `Person.java` and the now updated Schema Repository. The source code is compiled and annotated to create the `PersonTmpCopy.class` and the `PersonTmpCopy.classinfo` for PSE. The annotated class is registered with the PSE system in compliance with requirements of the *PSE Pro 2.0* and `PersonTmpCopy` gets a copy of all the objects that make the extent of the `Person` class. At this point the `PersonTmpCopy` class includes the requested change and its extent has all the objects transformed so to conform with its definition and hence to conform to the schema change. As a last we swap the OIDs of the `Person` objects with those of the `PersonTmpCopy` objects thereby maintaining the OIDs of the objects through the schema evolution process[15]; and we rename the `PersonTmpCopy` to the `Person` class through an API provided by PSE Pro 2.0.2.

This overall process is almost identical for all primitives that change an existing type.

- Step1: Change the Schema Repository. Create a new temporary instance of the *MetaClass* by augmenting or reducing the capacity of the original object (i.e., the original class) with the desired change.
- Step2: Create the Source. Create the source, i.e., 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 they are 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.
- Step6: Copy the Objects. Copy all the objects of the original class to the temporary class such that the objects of the temporary class are now augmented or reduced by the change.
- Step7: Swap OIDs. Preserve the OIDs by making the new objects have the same OID as the original objects [16].

[14] 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.

[15] We would like to thank the PSE team at Object Design Inc. for going out of their way and spending the time and effort to provide us with the swap OID feature as well as the capability to de-register classes from the PSE system on the fly. These new APIs have been released in PSE Pro 2.0.x

[16] This is part of a patch for PSE Pro2.0.2 provided to us by the PSE team at ODI.

- 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 [17].

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 it does not have any object copying 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 yet have support for joins) and does not meet all of our criteria for SERF transformation support. Thus as part of the implementation effort of OQL-SERF, we have designed and now 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 Pro 2.0,
- an API interface binding consistent with ODMG and Java's binding of ODMG for the applications using the OQL Query Engine.

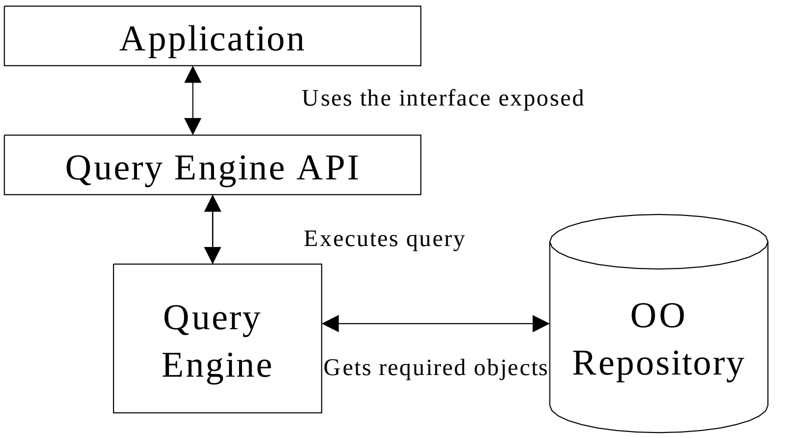

Figure 6:

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. Our OQL Query Engine is based on the OQL grammar given in Chapter 4 of the ODMG Standard 2.0 [Cea97].

The key functionality of the OQL Query Engine is to parse and map a given OQL statement to the right OODB system functions in order to get the required result. The left hand side of Figure 7 shows the steps involved in the building of the OQL Query Engine. To accomplish this, the OQL Query Engine must:

- Parse: parse the OQL statement

[17] This is part of a patch of PSEPro2.0.2 provided to us by the PSE team at ODI.

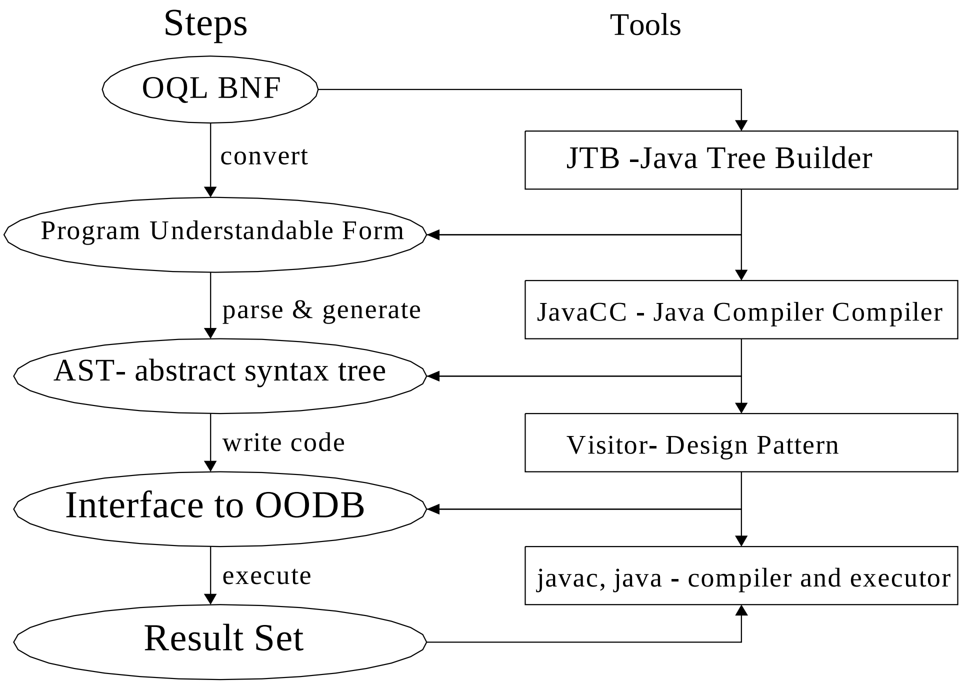

Figure 7: Tools for building the OQL Interface

- Generate: generate the abstract syntax tree
- Iterate: 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 that on the Web we found 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) is currently the most popular parser generator for use with Java applications. A parser generator is a tool that reads a 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. The Java Tree Builder (JTB) is one such tool. It is a syntax-tree builder to be used with the JavaCC. It takes as input a grammar definition in the JavaCC format and automatically generates:

- a set of syntax tree java class files based on the productions in the grammar,
- a properly annotated JavaCC grammar to build the syntax tree during parsing, and
- a *Visitor* superclass whose default methods allow us to visit the children of the current node.

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 [GHJV95]. The issue addressed by the Visitor pattern is the manipulation of composite objects. Without visitors, such manipulation runs into several problems. The Visitor pattern allows us to define a new operation on an object structure without changing the classes of the objects on which it operates. Rather than writing dedicated methods for each programming task and afterwards recompiling, with a visitor the idea is to:

1. automatically insert a so-called accept method in each class, and
2. write the action code in so-called visitors.

The Visitor pattern gives us the flexibility of rapidly retargeting 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. In our first version of the OQLVisitor for PSE, we have extended (e.g., developed a given join operator java class) and used the PSE query interface to allow for an extensive support for the OQL query language.

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 both in terms of the templates and the user interface.

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.

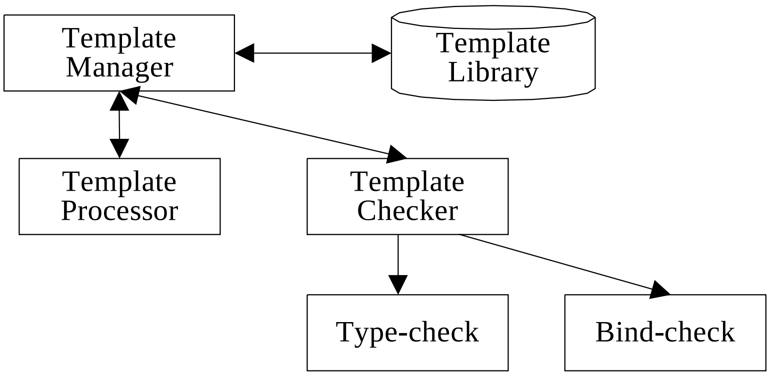

Figure 8: Modules for Realizing the Template

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** 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 [18]. A template object itself is

[18]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.

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 paprameters*, a list of *keywords* and the *source* corresponding to the SERF template. When a user stores a template, he is required to furnish the *name* and the *input parameters* for the template.

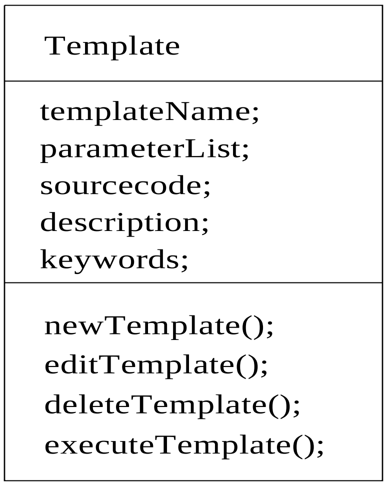

Figure 9: The Template Class

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 actual execution of the SERF Template is handled by the **Template Processor**, after having been passed the proper information needed for execution from the users through the interface of the Template Manager. The Template Processor is responsible for instantiating and executing a template. Figure 10 shows the steps performed by 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 legal 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 usabilty 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 developed a GUI as a frontend 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

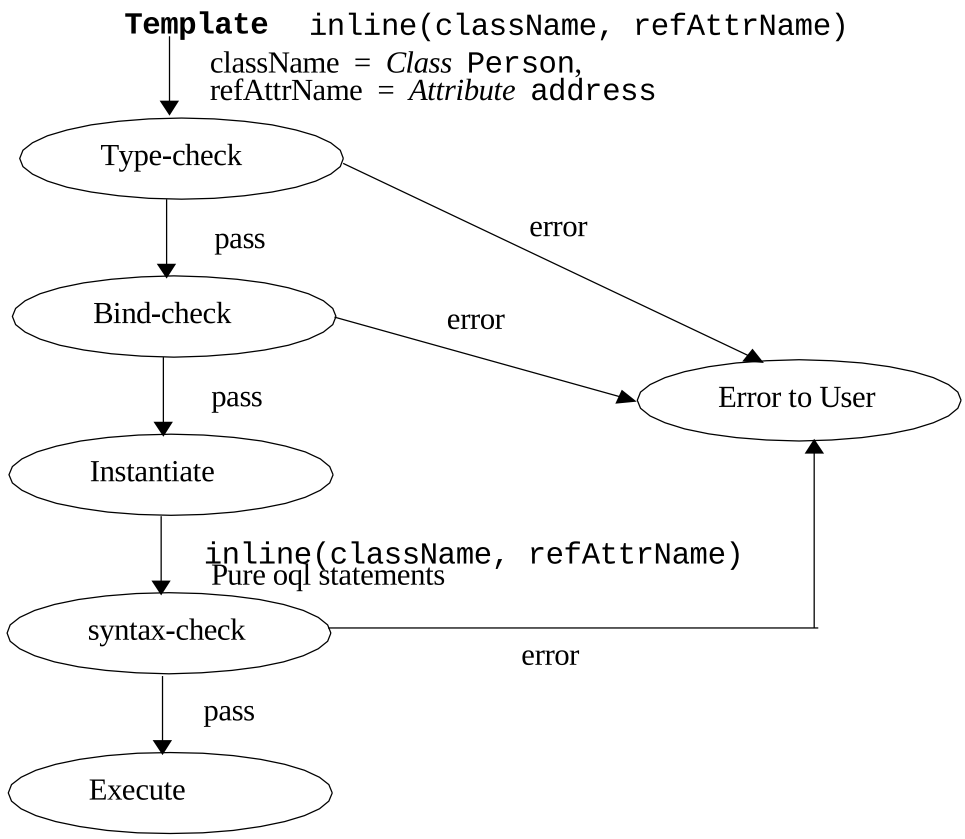

Figure 10: Steps for the Execution of a Template

has been applied to the schema thus giving the user a chance to verify if this selected template was indeed the desired transformation.

8 Conclusions

In this paper we have presented an implementation of OQL-SERF, the template-based scheam evolution framework. We have used the ODMG standard as the foundation for this implementation. As part of this work we have also devised our own taxonomy of schema evolution primitives and have shown it to be minimally complete in the context of the SERF framework.

In summary the main contributions of this paper are:

- Axioms of preservation - we have presented the invariants for preserving the ODMG object model under schema evolution.
- Taxonomy of schema evolution primitives - we have presented a minimally 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. This minimality holds within the context of the SERF framework.

- Development of a Dynamic Schema Evolution Manager - we have designed and implemented a dynamic schema evolution facility for *PSE Pro 2.0* [19].
- Development of an OQL Query Engine - we have designed and developed an OQL Query Engine for *PSE Pro 2.0* using JavaCC, JTB and Visitor design pattern.
- Discussion of Software Engineering Challenges - we presented the choices and decisions we had to make in the process of developing this system so as to make it re-usable and extensible to other systems.
- Design and Implementation of the OQL-SERF System - we have developed 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. Being based on ODMG, it should now be easily portable to any OODB that is ODMG-compliant.

We are currently in the process of completing the implementation of the modules mentioned in this paper. We have enough in place to be able to validate our ideas. We anticipate to have a working prototype later in the year, at which point we plan to release it to public domain via our website *http://www.davis.wpi.edu/OOSE/SERF.html.*

Acknowledgments. The authors would like to thank Gordon Landis, Sam Haradhvala, Pat O'Brien and Breman Thuraising at Object Design Inc. for all their help for producing a patch of PSE 2.0.2 capable of the meta data management needed for the development of our PSE schema evolver tool. We would also like to thank students at the Database Systems Research Group at WPI for their interactions and feedback on this research during our group meetings and discussions, such as in particular Xin Zhang and Andreas Koeller. 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.

References

[Ber92] E. Bertino. A View Mechanism for Object-Oriented Databases. In *3rd Int. Conference on Extending Database Technology*, pages 136–151, March 1992.

[BKKK87] J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *ACM SIGMOD Record*, pages 311–322, 1987.

[BOS91] P. Butterworth, A. Otis, and J. Stein. The Gemstone Object Database Management System. *Communications of the ACM*, 10(1):64–77, 1991.

[Bré96] P. Bréche. Advanced Primitives for Changing Schemas of Object Databases. In *CAISE*, 1996.

[Bri97] P. O. Brien. Making Java Objects Persistent. *Java Report*, (1):49–60, 1997.

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

[CJR98] K. Claypool, J. Jing, and E.A. Rundensteiner. SERF:Schema Evolution through an Extensible, Re-usable and Flexible Framework. Technical Report WPI-CS-TR-98-9, Worcester Polytechnic Institute, May 1998.

[Cla92] S.M. Clamen. Type Evolution and Instance Adaptation. Technical Report CMU-CS-92-133R, Carnegie Mellon University, School of Computer Science, 1992.

[Clu98] S. Cluet. Designing OQL: Allowing Objects to be Queried. In *Journal of Information Systems*, 1998.

[19] The dynamic schema evolution facility, the OQL Query Enigine for *PSE Pro 2.0* and OQL-SERF will be available for download from our web site http://www.davis.wpi.edu/ OOSE/SERF.html by the end of summer.

[FFM+95] F. Ferrandina, G. Ferran, T. Meyer, J. Madec, and R. Zicari. Schema and Database Evolution in the O_2 Object Database System. In *International Conference on Very Large Data Bases*, 1995.

[FMZ94a] F. Ferrandina, T. Meyer, and R. Zicari. Correctness of Lazy Database Updates for an Object Database System. In *Proc. of the 6th Int'l Workshop on Persistent Object Systems*, 1994.

[FMZ94b] F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. In *Proc. of the 20th Int'l Conf. on Very Large Databases*, pages 261–272, 1994.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison Wesley, 1995.

[Inc93] Object Design Inc. *ObjectStore - User Guide: DML. ObjectStore Release 3.0 for UNIX Systems.* Object Design Inc., December 1993.

[IS93] Inc. Itasca Systems. Itasca Systems Technical Report. Technical Report TM-92-001, OODBMS Feature Checklist. Rev 1.1, Itasca Systems, Inc., December 1993.

[Jin98] J. Jin. An extensible schema evolution framework for object-oriented databases using OQL. Master's thesis, Worcester Polytechnic Institute, May 1998.

[Lau97a] S-E Lautemann. A Propagation Mechanism for Populated Schema Versions. In *IEEE International Conference on Data Engineering*, pages 67–78, 1997.

[Lau97b] S-E Lautemann. Schema Versions in Object-Oriented Database Systems. In *International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 323–332, 1997.

[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.

[Nat98] C. Natarajan. CHOP: An Optimizer for Schema Evolution Operation Sequences. Master's thesis, Worcester Polytechnic Institute, June 1998.

[PO95] R. J. Peters and M. T. Ozsu. Axiomatization of Dynamic Schema Evolution in Objectbases. In *IEEE International Conference on Data Engineering*, pages 156–164, 1995.

[RR95] Y. G. Ra and E. A. Rundensteiner. A Transparent Object-Oriented Schema Change Approach Using View Schema Evolution. In *IEEE International Conference on Data Engineering*, pages 165–172, March 1995.

[RR97] 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*, pages 600–624, Aug/Sept 1997.

[RRL97] Y.G. Ra, E. A. Rundensteiner, and A.J. Lee. A Practical Approach to Transparent Schema Evolution. Technical Report WPI-CS-TR-97-3, Worcester Polytechnic Institute, Dept. of Computer Science, 1997.

[SZ86] A. H. Skarra and S. B. Zdonik. The Management of Changing Types in an Object-Oriented Databases. In *Proc. 1st OOPSLA*, pages 483–494, 1986.

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

[Zic91] R. Zicari. Primitives for schema updates in an Object-Oriented Database System: A proposal. *Computer Standards & Interfaces*, (12):271–284, 1991.