WPI-CS-TR-98-9 Revised: September 1999

SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework

by

Kajal T. Claypool and Elke A. Rundensteiner

SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework [1]

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

September 8, 1999

[1]This work was supported in part by several grants from NSF, namely, the NSF NYI grant #IRI 97–96264, the NSF CISE Instrumentation grant #IRIS 97–29878, and the NSF grant #IIS 97–32897. Dr. Rundensteiner would like to thank our industrial sponsors, in particular, IBM for the IBM partnership award and for the IBM corporate fellowship for one of her graduate students Special thanks also goes to the PSE Team specifically, Gordon Landis, Sam Haradhvala, Pat O'Brien and Breman Thuraising at Object Design Inc. for not only software contributions but also for providing us with a customized patch of the PSE Pro2.0 system that exposed schema-related APIs needed to develop our tool.

Contents

Abstract

With current database technology trends, there is an increasing need to specify and handle complex schema changes. The existing support for schema evolution in current OODB systems is limited to a pre-defined taxonomy of schema evolution operations with fixed semantics. Given the variety of types, complexity, and semantics of transformations, it is sheer impossible to a-priori provide a complete set of all complex changes that are going to meet all user's needs. This paper is the first effort to successfully address this open problem by providing an extensible framework for schema transformations. Our proposed 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 formally show how the templates guarantee the schema consistency under more complex transformations. To verify the feasibility of our SERF approach we have implemented one realization of our concepts in a working prototype system, called OQL-SERF, based on OQL, ODMG MetaData and Java's binding of ODL. To validate the completeness of our approach, we have also conducted a case study demonstrating that our SERF approach can handle all the schema evolution operations we identified from the literature.

Keywords: Schema Evolution, Transformation Templates, Object-Oriented Databases, Modeling Database Dynamics, OQL, ODMG, Schema Consistency.

1 Introduction

Not only is it difficult to pre-determine the database schema for many complex applications during the first pass, but worst yet application requirements typically change over time. For example [Sjo93] documents the extent of schema evolution during the development and the initial use of a health management system at several hospitals. There was an increase of 139% in the number of relations and an increase of 274% in the number of attributes, and every relation in the schema was changed at least once during the nineteen-month period of the study. These studies reveal that schema changes are an inevitable task not only during the development of a project but also once a project has become operational. Moreover, legacy systems are being migrated to newer state-of-the-art systems, users are looking at transforming schema and data from one object model to another, data warehousing communities are looking at transforming and cleansing information to bring to their users a clean and consolidated view.

Re-structuring is thus a critical task for a variety of applications. For this reason, most object-oriented database systems (OODB) today support some form of re-structuring support via schema evolution [Tec94, Tec92, BMO$^+$89, Obj93, BKKK87, Inc93]. This existing support of current OODBs [BKKK87, Tec94, BMO$^+$89, Inc93, Obj93] is limited to a *pre-defined* taxonomy of *simple fixed-semantic* schema evolution operations. However, such simple changes, typically to individual types only, are not sufficient for many advanced applications [Bré96]. More radical changes, such as combining two types or redefining the relationship between two types, are either very difficult or even impossible to achieve with current commercial database technology [KGBW90, Tec94, BMO$^+$89, Inc93, Obj93]. In fact, most OODBs would typically require the user to write ad-hoc programs to accomplish such transformations. In the last few years, research has begun to look into this issue of complex changes [Bré96, Ler96]. However, this new work is again limited by providing a *fixed* set of some selected (even if now more complex) operations.

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 ever exist. In fact, it would be sheer impossible to predict all transformations a user may desire. This problem exists for simple transformations but becomes even more pronounced for complex transformations. For example, consider the merging of two source classes into one single merge class as one illustration of the variety of nuances in the desired semantics that are possible for this one operation. The structure of the new merge class can be defined in many different ways such as doing a union, intersection or a difference of the properties of the two input classes. Moreover, there can also be many different semantics for populating this merge class, like a value-based join on some pair of properties, an oid-based join, etc. Similarly, multiple choices exist for the fate of the source classes and the placement of the merge class in the class hierarchy. It can be seen that the more complex the transformation, the more difficult it becomes to finitely predict all possible semantics to be desired in the future.

To address this limitation of *current* schema evolution technology, we propose the SERF framework that allows users to perform a wide range of complex user-defined schema transformations *flexibly*, *easily* and *correctly*. To the best of our knowledge, our work is the first to address this problem by proposing an extensible schema evolution framework. Our approach is based on the hypothesis that complex schema evolution transformations can be broken down into a sequence of basic evolution primitives, where each basic primitive is an invariant-preserving atomic operation with fixed semantics provided by the underlying system. In order to effectively combine these primitives and to be able to perform arbitrary transformations on objects within a complex transformation, we propose to use a query language. Previous research has resorted to using a programming language to achieve ad-hoc user transformations [KGBW90] or defining a new language [DK97] for specifying the database transformations. In our work, we propose the use of the standard query language for object database systems, OQL [Cat97], and demonstrate it to be sufficient within our framework if combined with meta-data access.

However, SERF transformations, ad-hoc programs and any new language all suffer from the fact that they specify the transformation for a particular schema. In our SERF framework, we go one step further by introducing the new concept of a SERF *template*. A template extends the notion of a SERF transformation to be a named transformation that can include variables and input and output parameters. The SERF transformation code itself is re-written such that it is generic and can be applied based on the provided input parameters. A template can thus be used for different sets of parameters (i.e., applied to different schemas). Furthermore it can also be re-used for building new more complex transformations. These generic templates can then also be applied for different systems, i.e., for different object databases and different object models, thus making them a valuable community wide resource. With that purpose in mind, one of the goals of this work is to also provide a library of these re-structuring templates for different domains as a resource for the re-structuring community.

In summary, our proposed framework gives the user:

- The *flexibility* to define the transformation semantics of their choice.
- The *extensibility* of defining new complex transformations meeting specific requirements.
- The *generalization* of these transformations through the notion of templates.
- The *re-usability* of a template from within another template.
- The *ease* of template specification by programmers and non-programmers alike.
- The *soundness* of the 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, we have set out to develop a working system, called OQL-SERF [RCL$^+$99]. OQL-SERF serves both as proof of concept as well as helps 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. 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 3). 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 and OQL as its database transformation language. OQL-SERF uses Object Design Inc.'s PSE as its persistent store [O'B97]. 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 this paper we present:

- The SERF Framework -
  - the key concepts of the SERF framework,
  - the notion of a SERF template and template language, and
  - the general architecture of the SERF framework.
- Evolution of the ODMG Object Model -
  - 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 that we designed such that they have minimal semantics and a combination of them is able to describe a large set of transformations.
- Formal proof of preservation of consistency by templates.
- Case Study - a study of the various transformations available in the literature has demonstrated the suitability of OQL as a full transformation language, i.e., no language extensions of OQL are needed.
- Implementation - The design and implementation of the OQL-SERF system fully based on ODMG, that is, the Object model, OQL and the Schema Repository, as a proof of concept for the SERF framework. This shows not only the feasibility of these ideas but also shows the portability of the SERF Framework to any ODMG compliant OODB system.

While some of our preliminary ideas of the framework have been presented in [CJR98b], we now fully define the SERF framework as well as illustrate it with detailed examples. In addition we offer the following additional progress:

- A study of the schema consistency under the application of templates, including a proof of consistency preservation.
- A case study showing the feasibility of SERF templates as a transformation framework capable of doing a large number of complex transformations, and
- Complete implementation details for the OQL-SERF system that has been successfully demonstrated at ACM SIGMOD 1999 [RCL$^+$99].

The rest of the paper is organized as follows. Section 2 talks about related research. Section 3 describes the ODMG Standard and our evolution support for the ODMG object model. Section 4 details our framework and Section 5 shows how the consistency of a schema is preserved in our system. In Section 6 we present a case study of complex schema evolution operations supported by the SERF framework. In Section 7 we describe an implementation of our framework. We conclude with a summary and some future work discussion 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 [Inc93], GemStone [BMO$^+$89], ObjectStore [Obj93], and O_2 [Tec94] all essentially handle a set of evolution primitives similar to Orion's.

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 only on the type changes and not the changes of the object instances associated with the to be modified type. She provides compound type changes like *Inline*, *Encapsulate*, *Merge*, *Move*, *Duplicate*, *Reverse Link* and *Link Addition*.

Breche [Bré96] proposed a similar list of complex evolution operations for O_2, i.e., now considering both schema as well as object changes. [Bré96] claims that these advanced primitives can be formulated by composing the basic primitives that are provided by the O_2 system. He shows the consistency of these advanced primitives. Like other previous work, the paper however still provides a fixed taxonomy of possibly complex 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. In summary, all previous research in database schema evolution tends to provide the users with a *fixed* set of schema evolution operations [FFM$^+$95, BKKK87]. No provision 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.

Similar work has been done in the context of *database transformation*. Database transformation in the mapping of instances of one or more source databases to an instance of a target database. The schemas involved can be expressed in a variety of different database models and can be implemented using different DBMSs or other kind of data repositories. Davidson et al. have done work in defining a new language WOL [DK97] for specifying such database transformations. In our work, we combine a standard database query language and schema evolution services into one framework and strive to provide a consistent yet extensible environment for doing both the schema and the data transformations.

We expect the system dictionary (schema repository) implementation of an OODB system to be fully

ODMB compliant. If this is not achieved by a given OODB system, then our template library could not operate on this OODB system without some prior modifications to utilize the specific notations of the proprietary schema repository of the OODB system. Lakshmanan et al. have developed *SchemaSQL* [LSS96], an extension of SQL that allows for manipulation of data and meta-data in relational multi-database systems. This has the advantage of offering interoperability for meta-data management in relational systems through requiring a query (pre-) processor to translate SchemaSQL into SQL that can be understood by the specific SQL engine. The concept of building generic and re-usable transformations, a la our templates, has not been explored.

Research has also been done towards providing behavioral consistency, i.e., the consistency of class methods under primitive schema changes [DZ91, MNJ94]. We have not re-addressed the issue of behavioral consistency in the context of our SERF templates, but as shown in Section 5 we assume that the schema evolution primitives employed in a template are the only operators to ever modify and hence possibly effect the consistency of the schema. Hence, we expect a solution that is guaranteed to preserve not only the structural consistency but also the behavioral consistency will also indirectly result in behavioral consistency preservation for our templates (a la Section 5).

Further research in schema evolution 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, in particular deferred and immediate propagation strategies [FMZ94b, FMZ94a]. Such optimization strategies are orthogonal to the issue of expressiveness and extensibility of the schema evolution support and in principle either of these propagation strategies could be implemented for our framework.

Peters and Ozsu [PO95] have introduced an 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 use their notation and model to represent the concepts presented in this paper.

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 [RLR98, RR97, Ber92] and versions [SZ86, Lau97]. However, such work has focused on simple schema evolution operations only, and hence it may need to be re-examined in order to handle the complex notion of transformations as introduced by our templates.

3 ODMG Standard and Schema Evolution

The SERF framework is based on the ODMG standard [Cat97], more specifically it uses the ODMG object model, the ODMG Schema Repository and the ODMG Object Query Language (OQL). Here we give a brief description of the ODMG object model, and then present our proposal for providing primitive schema evolution support for the ODMG model.

3.1 ODMG Standard: The Object Model

The ODMG Object Model is based on the OMG Object Model for object request brokers, object databases and object programming languages [Cat97, Clu98]. 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 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* and *literal types.* Each object has a unique object identifier which 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 *persistence by reachability.*

Inheritance. Although ODMG defines multiple inheritance, Java's binding of ODMG Model supports only single inheritance. A subclass therefore 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.

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

3.2 Evolving the ODMG Object Model

Some support for schema evolution is provided by most OODB systems [BKKK87, Tec94, BMO$^+$89, Inc93, Obj93]. This support is generally in the form of a pre-defined taxonomy of *simple fixed-semantic* schema evolution operations. However, the current ODMG standard does not yet address the issue of schema evolution. In this section we therefore describe a taxonomy of primitives to provide support for dynamic schema evolution of the ODMG object model based on a representative set found in other OODB systems. Here, 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. The primitives presented here are *minimal* in that they cannot be decomposed into any other evolution primitives and *essential* in that they are all required for the evolution of the given object model. They can however be composed together with other evolution primitives to form more complex transformations, as we will show in this paper.

3.2.1 Invariants for Preserving the ODMG Object Model

A schema update can cause inconsistencies in the structure of the schema, referred to as structural inconsistency. An important property imposed on schema operations is thus that their application always results in a *consistent* new schema [BKKK87]. The consistency of a schema is defined by a set of so called *schema invariants* of the given object data model [Bré96]. In this section, we present the invariants for the ODMG object model, which we have adapted from the axiomatic model proposed by Peters and Ozsu [PO95].

Axiomatization of Schema Changes. 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 which must be maintained by any schema evolution primitive attempting to change the structure of the schema.

Term	Description
$\mathcal{T}$	All the types in the system
$s,\ t,\ T,\ \perp$	Elements of $\mathcal{T}$
P(t)	The immediate supertype of type t
C(t)	The immediate subtypes of type t
N(t)	The native properties of type t
H(t)	The inherited properties of type t

Table 1: Notation for Axiomatization of Schema Changes.

Table 1 shows the notation we use for describing the axiomatic model. In the table, *native* properties *N(t)* refer to the properties of `type` t that are defined locally in the type t. *Inherited* properties *H(t)* 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 be only an `attribute` as the Java binding of ODMG does not support relationships as yet.

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*.

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 are 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 union of the inherited and native properties for a type.

3.2.2 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 3.2.1 (see Table 2 for a listing). 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 two or more primitives.
- Simple, i.e., each primitive has minimal semantics so as to not embed semantics in the primitives.
- Consistent, i.e., each primitive is guaranteed to generate a valid schema when applied to a valid schema.

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

Table 2: The Minimal Taxonomy of Schema Evolution Primitives.

Following the Orion schema evolution taxonomy [BKKK87], we have kept the schema changes *add-class (create-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* and *rename-class* from our tier one of the primitive set due to our minimality requirement. The reason for this is that they can be achieved by the composition of two other schema evolution primitives, i.e., by our SERF framework as non-primitive changes. For example, *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 the old attribute to the new one. For this reason, the *change-name-of-attribute* is not a fixed predefined primitive in our framework. We have also excluded *rename-class* and *retype-attribute* from the taxonomy as they can be achieved in a similar fashion. A library of these basic transformations which are not *minimal* yet may be very common and useful can be provided by the SERF framework. Moreover, if an OODB system were to a hard-coded primitive operation for them, then these could be made available to the users via SERF. 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].

However, we recognize the inefficiency of having basic transformations as a part of the SERF framework instead of implemented as primitives at the system level. For this we are investigating optimization techniques for templates in general that can help us address this problem [CNR99]. Secondly, commonly used templates can be provided as a hard-coded system level implementation as an alternative optimized solution, and wrapped as SERF templates, thereby offering a transparency from a user's perspective.

3.3 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 and hence our SERF system 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 both 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 relationships between the meta-objects which define the schema graph. A class `Person` related to the class `Address` and the inheritance between two classes are examples of such relationships. These relationships help guarantee the referential integrity of the meta-object graph. The user has direct access to all of the information in the schema repository via the query language, OQL.

4 The SERF Framework

In this section we present the fundamental principles of our proposed transformation framework. In particular, we will demonstrate how our proposed framework succeeds in giving the user the *flexibility* to define the semantics of their choice, the *extensibility* of defining new complex transformations, and the *re-usability* of these transformations through the notion of templates.

4.1 Features of the Framework

Our proposed framework aims at addressing the limitations of current OODB technology that restrict schema evolution to a *predefined* set of operations with *fixed* semantics. In particular, our goal is to instead support *arbitrary user-customized* and possibly *very complex* schema evolution operations. Similar to [Bré96], our first step in this direction is to allow users to build new schema transformations with customized semantics using a fixed set of schema evolution primitives provided by the underlying OODB system (see Section 3.2.2 for the taxonomy we assume.).

While our work is based on describing complex schema transformations using a fixed set of basic schema evolution primitives, we find that a pure sequence of these schema evolution primitives is not always powerful enough to express a sufficiently large class of transformations. For example, Figure 1 shows an example of a schema transformation that creates a new class `JournalPaper` based on the two input classes `Author` and `Paper`. The class `JournalPaper` is constructed by collecting some of the attributes that exist in the two classes **Author** and **Paper**. For example, we might only want to add the attribute **AuthorName** if it exists in both the classes **Author** and **Paper**. And in this case, we want to add it only once. There is no pure sequence of schema evolution operations that could characterize this logic of attribute selection for the **Journal-Paper** class. Thus, we recognize the need to have some language as "glue logic". Moreover, such a language is also needed to achieve value-based transformations of objects across types. For example, if we are adding the attribute **AuthorName** from the class **Author** we might also want to take the value of the attribute **AuthorName** from the instances of the class **Author**. In current OODB systems, in order to achieve these types of transformations users have to resort to writing ad-hoc code using a programming language. This has the drawback of being both programming language and system dependent and also of not having any guarantee of schema consistency.

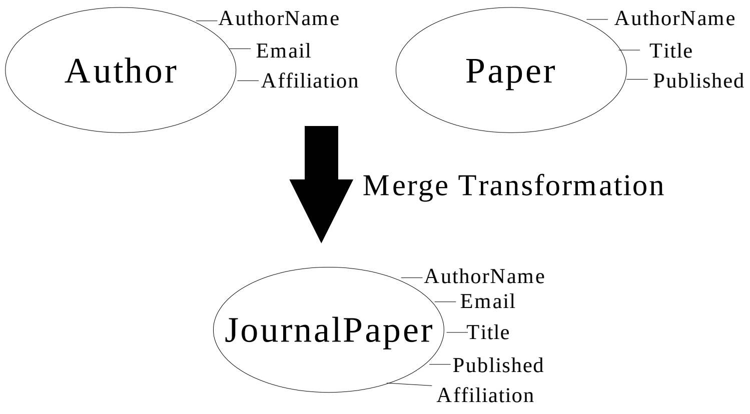


Figure 1: An Example Schema Graph for a MERGE Transformation

In our framework, we therefore advocate the use of a declarative query language like OQL [Cat97] as ideally suited to meet the needs of our transformation language. The query language must have an interface for invoking the schema evolution primitives, as those provide the basic mechanism to syntactically change the type structure. The query language must also have the expressive power for realizing any arbitrary object manipulations to transform objects from one object type to any other object type. In Section 5, we have shown that this approach also guarantees consistency for all schema transformations.

In our framework, these arbitrarily complex transformations can be encapsulated and generalized by assigning a name and a set of parameters to them. From here on these are called *transformation templates* or *templates* for short. By parameterizing the variables involved in a transformation such as the input and the output classes and their properties, a transformation becomes a *generalized reusable* module applicable to any application schema. By assigning a name to such a template, it can also now be *re-used* from within other transformations. This leads us to the idea of collecting these templates in a *template library* and thus guaranteeing the availability of these templates to any user at any time, just as the fixed set of schema evolution operations are available to the users in any regular schema evolution system.

In summary, a schema transformation in our framework lets the user combine an object transformation language with a standard set of schema evolution primitives to produce arbitrarily complex transformations. Moreover, these transformations are generalized and stored in a standard library for later re-use.

Transformations in this general form are called *templates* in the framework and the library, the *template library*.

4.2 Schema Transformations

A schema transformation can be used to express different semantics for primitives as well as to create new possibly complex schema evolution operations.

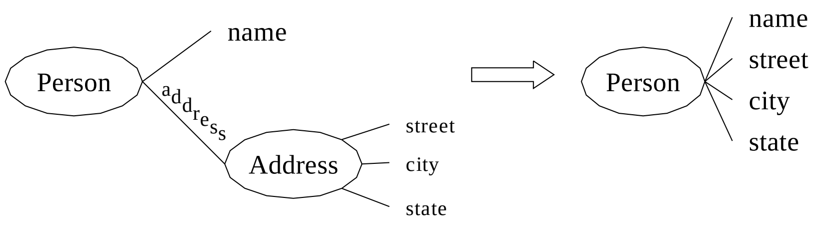


Figure 2: Example of the Inline Transformation.

```
// 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                                        Step B
   select c
   from Person c;

// Update all the objects
for all obj in extents():                                  Step C
   obj.set (obj.Street, valueOf(obj.address.Street));
   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 3: SERF Representation of the Inline Transformation using OQL

We illustrate the steps involved in a schema evolution transformation using the example of *Inline* which 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. For this, 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 3 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 types of operations [1]:

[1]Each type of operation can be composed of or inter-mingled with the other types of operations. For ease of readability, we denote each type of operation as a step. Thus, 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 (refer to Section 5). For example, **Step A** in Figure 3 shows the addition of the attributes `street, city` and `state` via the *add_attribute* 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 themselves get modified (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 3 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 (such as `add_atomic_attribute()` in **Step A**) and the system-defined functions for updating the objects (such as `obj.set()` in **Step C**).

In this example, we simply wanted to add attribute values to the existing objects. In some transformations, existing objects might need to be deleted or new objects might be created. OQL allows for the creation of new objects through a constructor-like statement. Deletion of objects has to be done by some system-defined method.

4.3 Transformation Templates

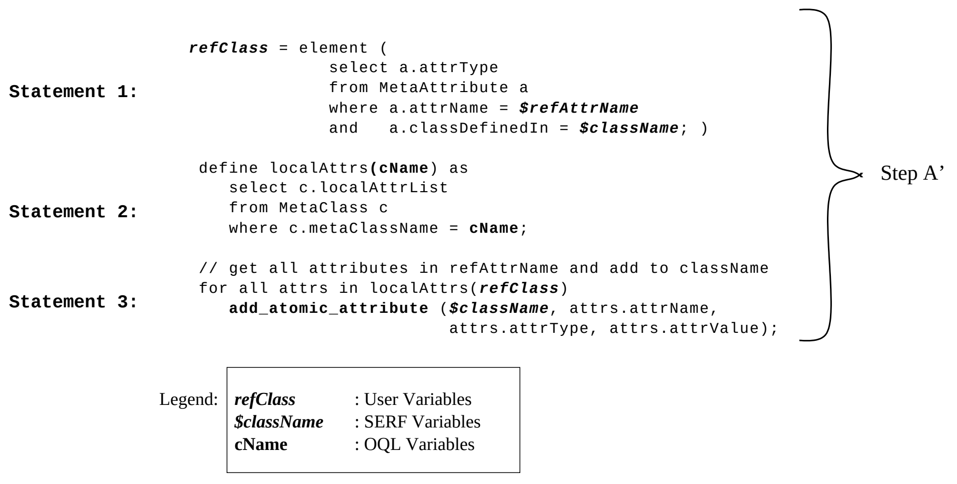
In Figure 3, we have shown how a schema transformation can be written for inlining the `Address` class into the `Person` class using OQL, schema primitives, and system-defined object manipulating methods. Embedding schema evolution operations in OQL now allows the user to achieve any desired transformation, i.e., we have achieved flexibility in terms of the class of transformations that a user can express. We thus consider this to be a big step above the *fixed* set of schema evolution primitives. However, this can be compared to the functionality of a custom program in a high level language. Adapting and re-using the notion of templates, we provide greater re-use for SERF transformations by now introducing the notion of *templates* [CJR98b]. For example, it can be observed that the *inline* transformation as shown in Figure 3 is a useful transformation that could perhaps be applied for inlining any Class A into any Class B. To achieve this we enhance the SERF transformation such that it now has a name and input parameters. The transformation itself has to be re-written such that it is independent of any specific class or attribute.

Naming a Transformation. A SERF transformation is a sequence of pure OQL statements meant for a one time use and as such cannot be re-invoked. In the SERF template, we introduce the notion of a *name* attached to a SERF transformation so that it can be invoked at a later time and thus can be re-used.

Parameters of a Transformation. However, we find that while having a name allows us to invoke the set of OQL statements that form a transformation it is not sufficient to address the problem of re-use in terms of other classes and attributes to which it can be applied. Hence, for a SERF template we have added input parameters that allow a transformation to be applicable to a different set of classes or attributes. We have enhanced the input parameters such that they are *typed.* A list of the parameter types appears in [CRH99]. These typed parameters provide us with some degree of type-checking and hence a measure of syntactic consistency checking for SERF templates.

Generalization of a Transformation. However, the transformation in Figure 3 is specific to a particular class and attribute. Input parameters and a name alone are not sufficient to make this transformation re-usable. Thus in order to achieve complete re-usability we assume that the system dictionary for the database is accessible via the query language. Hence in a SERF template we use OQL's ability to query over the meta data (as stated in the ODMG Standard) and we also introduce variables [2]. Figure 4 shows how the meta data can be used to achieve a generalized *inline* transformation that can now be applied for any set of parameters. Here we use the system dictionary to not only discover the class being referred to by the reference attribute `refAttrName` but also then to get the set of attributes that belong to the reference class.

Step A' in Figure 4 is an equivalent of **Step A** in Figure 3. Here **Statement 1** defines the retrieval of the class referred to by the parameter `refAttrName` [3], **Statement 2** gets all the attributes that have been defined in the `refClass` and **Statement 3** iterates over all the attributes of `refClass` and adds them to class referred to by the parameter `className`.



```
                           refClass = element (
                                         select a.attrType
Statement 1:                             from MetaAttribute a
                                         where a.attrName = $refAttrName
                                         and   a.classDefinedIn = $className; )

                            define localAttrs(cName) as
                               select c.localAttrList
Statement 2:                   from MetaClass c
                               where c.metaClassName = cName;

                            // get all attributes in refAttrName and add to className
                            for all attrs in localAttrs(refClass)
Statement 3:                   add_atomic_attribute ($className, attrs.attrName,
                                                     attrs.attrType, attrs.attrValue);
```

Legend:

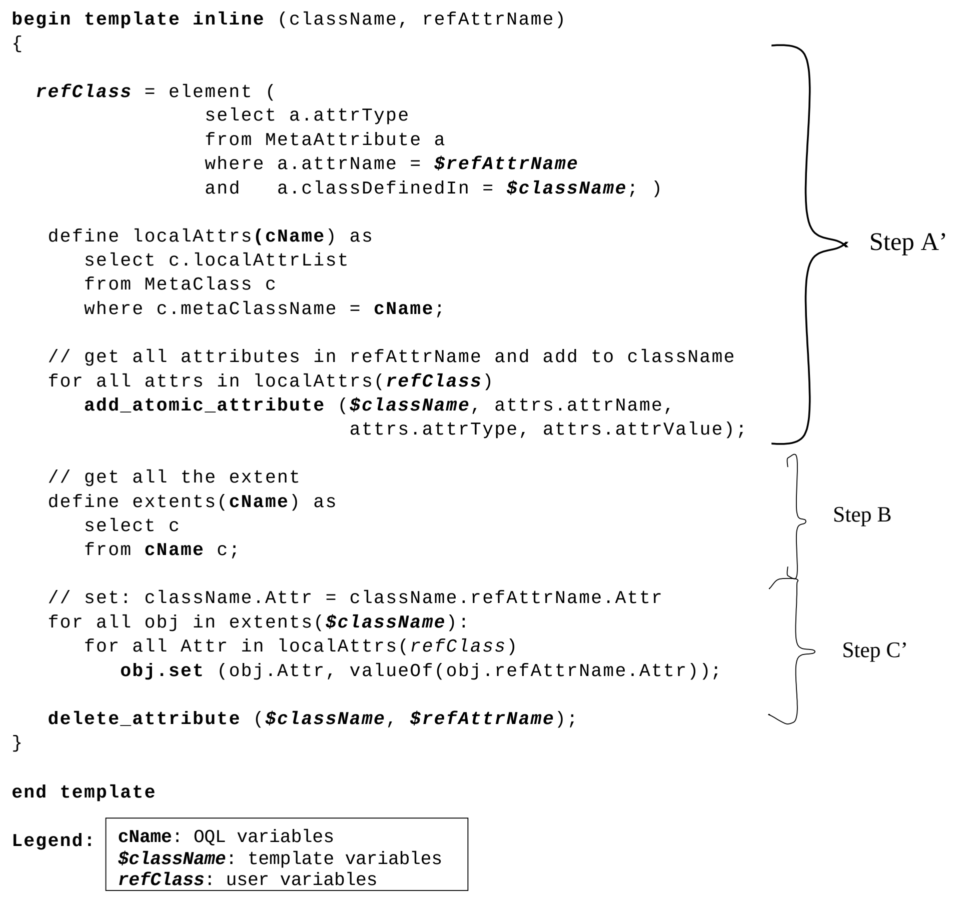
refClass	: User Variables
$className	: SERF Variables
cName	: OQL Variables

Figure 4: Generalizing the Inline Transformation using the ODMG Schema Repository.

Figure 5 shows a templated version of the *inline* transformation shown in Figure 3. This *inline* template when instantiated with the variables `Person` and `address` will achieve the same effect as the *inline*

[2]Currently, OQL does not support variables and hence we have enhanced the Template bnf to now include variables.
[3]This is `define` statement, in that it defines a statement that can be executed at a later time.

transformation shown in Figure 3.



```
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 5: Generalized Inline Template based on the Inline Transformation.

4.3.1 SERF Template Language

A SERF template is thus a named sequence of OQL statements extended with a name, parameters and variables that can be translated down to pure OQL statements during the process of instantiation. The `BNF` for a SERF template [4] is given in Figure 6.

4.3.2 SERF Template Instantiation and Processing

During the instantiation process the SERF template specification is translated to pure OQL statements. A consistency and bind check is performed both before and after the instantiation of the template. Figure 7

[4]In the BNF restricted_query denotes the OQL query limited to invoking only object updates and the schema evolution primitives.

$$
\begin{array}{rcl}
\textit{template} & ::= & \textbf{begin} - \textbf{template} \quad \textit{template\_name} \\
 & & ([\textit{parameter}]^*) \\
 & & \textit{template\_statements} \\
 & & \textbf{end} - \textbf{template}; \\
\textit{template\_statements} & ::= & \textit{template\_statement} \quad | \\
 & & \textit{template\_statement} \\
 & & \textit{template\_statements} \\
\textit{template\_statement} & ::= & \textit{define\_query} \quad | \quad \textit{query} \\
\textit{define\_query} & ::= & \textbf{define} \quad \textit{identifier} \quad \textbf{as} \quad \textit{query} \\
\textit{query} & ::= & \textit{query} \quad | \quad \textit{restricted\_query} \\
\textit{restricted\_query} & ::= & \textit{query}([\textit{function}]^*) \quad | \quad \lambda \\
\textit{function} & ::= & \textit{system\_function}(\textit{basic\_query}^*) \quad | \\
 & & \textit{schema\_primitive}(\textit{parameter}^*) \\
\textit{parameter} & ::= & \textit{string\_literal} \quad \textit{parameter} \quad | \\
 & & \textit{string\_literal} \\
\textit{basic\_query} & ::= & \textit{nil} \quad | \quad \textit{true} \quad | \quad \textit{false} \quad | \quad \textit{literal}
\end{array}
$$

Figure 6: The BNF for a SERF Template.

shows the steps for the execution of a template. When a template is created, the user can assign a name to it and also specify its parameters. A compile-time syntax check is performed for both the OQL queries and the *system_function* in the template. At the time of instantiation of the template, the user needs to provide the bindings for the parameters. A second check performs the type-checking, i.e., ensures that the parameters provided are of the type specified for the template and also validates the bindings, i.e., ensures that the provided parameters exist in the database. A successful check leads to the actual execution of the template. At this point, as a template is a sequence of pure OQL statements we use the OODB system's OQL Query Engine to execute the template against the underlying database.

In summary, the templates provide users not only with the advantages achieved by our schema transformations, i.e, a user can specify their own semantics for transformations, but also allow *re-usability* of these schema transformations by parameterizing them. This thus makes the templates reusable and applicable to any application schema. Moreover, our SERF template BNF allows us to guarantee the consistency of the schema after the application of our schema transformations (For more details see Section 5.).

4.4 The SERF Framework Architecture

4.4.1 System Architecture

Figure 8 gives the general architecture of our proposed SERF framework. As we will explain further, the components listed on the top half of the figure make up the framework and thus are to be provided by any implementation realizing our framework. The components listed below the line represent system components that we expect any underlying OODB system to provide.

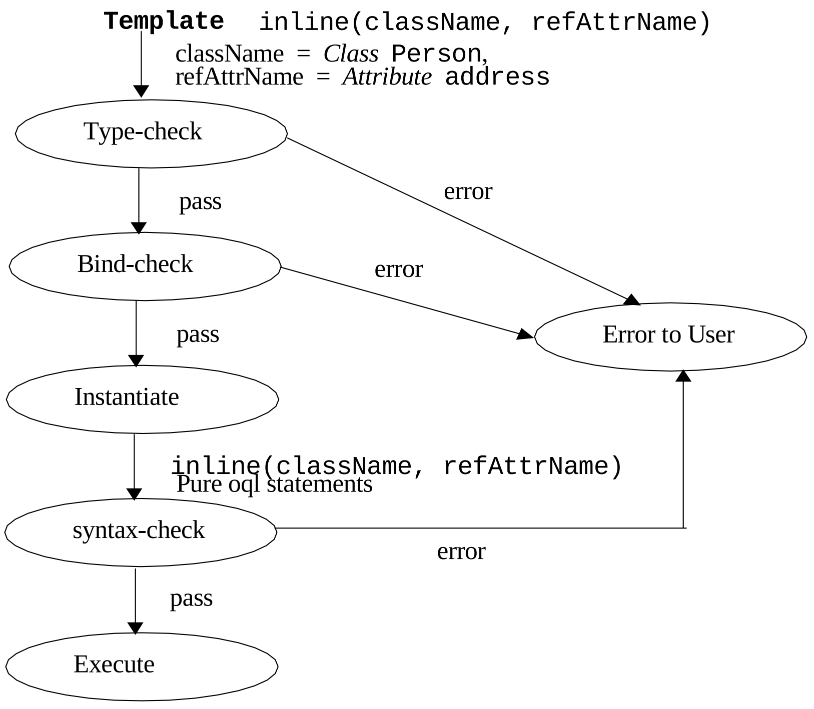

Template inline(className, refAttrName)
className = Class Person,
refAttrName = Attribute address
Type-check
error
pass
Bind-check
error
Error to User
pass
Instantiate
inline(className, refAttrName)
Pure oql statements
syntax-check
error
pass
Execute

Figure 7: Steps for the Execution of a Template

Figure 8 also shows the flow of the composition of a template. In general, a template [5] uses a query language to query over the schema repository, i.e., the meta data [6], 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. Although any query language could be used for this, from here on we will talk about OQL as our query language, since it is part of the ODMG standard and is perfectly suited for our needs as we will show below.

4.4.2 System Requirements

For our framework, we assume that the underlying OODB system provides us the following components:

- **Schema repository.** For a transformation (as shown in Figure 3) to be generalizable to a template (as shown in Figure 5) we need to be able to query and access the meta data 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 meta data. Most OODB systems indeed do allow access to the meta data, i.e., the schema repository, and in most cases this is via some high-level declarative interface like a query language instead of just a procedural low-level API.

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

[6]More details on the schema repository are presented in Section 4.4.2.

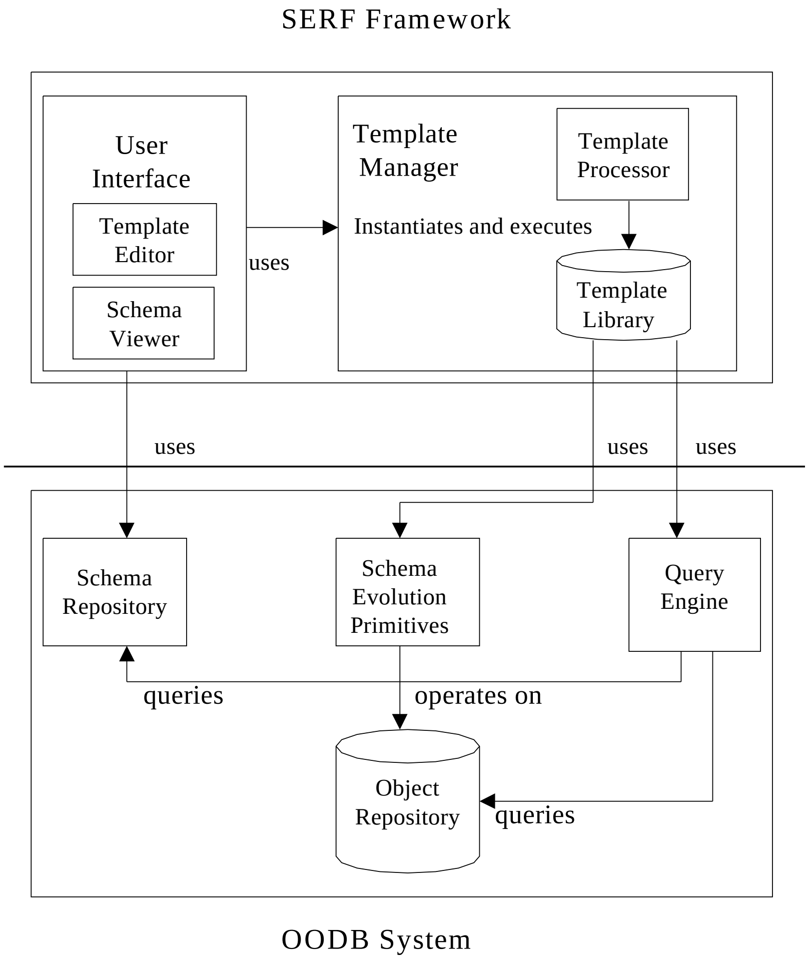


Figure 8: Architecture of the SERF Framework

- **Taxonomy of schema evolution primitives.** The OODB needs to provide a set of schema evolution primitives that is complete [7] 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 meta data 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 meta data 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:

[7]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 the invocation of system-defined update methods, and
- support for universal and existential quantification.

4.5 Template Library

Most databases today provide some support for schema evolution. This support is very specific to the underlying OODB system and while the primitives might achieve the same functionality, they are not by any means portable. Thus as part of this work we propose the development of a library of schema evolution templates. These templates could then be ported across many different systems and many different object models as long as the system requirements as set forth in Section 4.4.2 were met.

Thus SERF templates for a particular domain can be collected in a Template Library and search of a template in this library could be supported via simple key-word search on all stored parameters of the template such as the input and output parameters, the name, and its description. *Inheritance* and other semantic relationships between templates could be defined to facilitate a multi-level organization of the template library. We envision that a template library could become an important resource in the schema evolution community much like the standard libraries in the software engineering q environment.

5 Preservation of Schema Consistency by Templates

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 a given object data model [Bré96] [8]. So it follows that similar to schema evolution primitives, the templates must also preserve the invariants described for an OODB. This leads to the following definition.

Definition 1 *A schema is defined to be* **consistent** *if and only if it preserves the schema invariants defined for the underlying object model.*

Definition 2 *A schema evolution primitive, P_i, with $P_i \in \mathcal{P}$, where $\mathcal{P}$ is the set of schema evolution primitives, is said to* **preserve the consistency** *of the schema if and only if it preserves the invariants when applied to any consistent schema.*

Definition 3 *Each schema evolution primitive is defined to be a consistent* **atomic** *operation, if it is either successfully executed changing both the schema and the corresponding objects, or no change is made to the*

[8]While the invariants proposed in the literature are largely very similar, there are some slight differences based on the underlying object model of the OODB [Tec94, BKKK87]. For example, some object models support multiple inheritance and others support only single inheritance, and hence the invariants have to be slightly adjusted to account for this difference.

schema and the objects[9]*. A change to a schema can be made only using the schema evolution primitives defined by the underlying OODB system.*

Given the above definitions, we will show how a template also preserves the consistency of a schema. For this, we first formally define a template.

Definition 4 (Template) *A template is a* **named, parameterized, atomic** *transformation with the BNF as defined in Section 6.*

In our framework, we have restricted the query language to only invoke a limited set of operations and functions. We denote this restricted query language by *restricted_query* in the BNF. This set of operations includes the schema evolution primitives and some system-defined operations for setting the values of objects. We denote these in the BNF by *schema_primitive* and *system_function*, respectively. While this may limit the ability of users to employ some powerful foreign functions [10], it is essential for assuring the well-foundedness of our template framework. In particular, it assures some level of portability of our templates if method calls are restricted to a standard generic set of object methods, such as, *setValue* or simply *set*. Secondly, this also implies that no method other than a schema evolution primitive can change the structure of the application schema. In order to show that a template also preserves the consistency of the schema we exploit this key idea.

In the remainder of this discussion we will refer to :

- *query* ::= *restricted_query* as restricted query.
- *query* ::= *schema_primitive*(*parameter*) as the SE primitives.
- *query* ::= *system_function*([*basic_query*]) as object manipulation.

To prove that a template preserves the consistency of a schema, we will show that each type of `query` in a template, from here on called a *component*, leaves the schema in a consistent state.

Restricted query component. Restricted queries are used to get information from the schema repository, such as the attributes of a given class or the names of all classes in a schema. They are also used to retrieve the extents of the types defined in the application schema. These queries only retrieve information from the OODB. They are not able to change the structure of the schema in any way and hence have no effect on the consistency of the schema. This implies that if the schema was consistent before the execution of *restricted_query*, then it will be consistent after the execution of the *restricted_query*.

Lemma 1 *The application of any restricted_query to a consistent schema will leave the output schema in a* **consistent** *state.*

[9]An atomic change to objects is at times conceptual, since deferred propagation to instances is allowed.

[10]A given realization of our framework could relax this restriction by carefully permitting the use of some foreign functions within the constraints of the well- foundedness of our framework

SE primitives component. By **Definition 3**, the only means by which a template can change a schema is by invoking schema evolution primitives. Using **Definition 2**, we will now show that a sequence of schema evolution primitives leaves the schema in a consistent state.

Definition 5 (Sequence of Schema Evolution Primitives) *A sequence of schema evolution primitives $\mathcal{S}$ such that $S = p_1, p_2, \ldots p_n \quad | \quad p_i \in P, \quad \forall \quad i, \quad 0 < i \leq n$ is defined as a finite permutation of elements of $\mathcal{P}$, where $\mathcal{P}$ is a set of schema evolution primitives provided by the OODB system.*

By **Definition 3**, we know a schema evolution primitive, $p_i \in \mathcal{P}$, leaves the schema in a consistent state. Since the schema is consistent after the execution of each primitive, then it follows that it should still be consistent after the execution of the last primitive in the sequence. This observation is formally stated next.

Lemma 2 *If $\forall\ p_i \in S$, with $p_i \in P$, the primitive p_i is consistent by Definition 2, then the sequence of schema evolution primitives $\mathcal{S}$ as defined in* **Definition 5** *also preserves schema consistency.*

This lemma can be proven in a straightforward manner using mathematical induction and hence the proof is omitted here.

Object manipulation component. A template can do object manipulation in two ways. It can use OQL to create new objects by using the constructor or it can update the objects by invoking the *system_function* for updating the objects. Neither the constructor nor the *system_function* change the structure of the schema or the structure of the object as defined by its schema. From this we conclude that the structure of the schema itself is left unchanged through any *system_function* or OQL constructor creation.

Lemma 3 *A system_function, if applied to a consistent input schema, will generate a* **consistent** *output schema.*

From the above Lemmas 1, 2, and 3, we see that the only component of a template that can cause a change in the structure of the schema are the schema evolution primitives. Based on Lemma 2 we know that applying any sequence of these schema evolution primitives will assure consistency. This leads us to the following theorem.

Theorem 1 (Template Consistency) *A template as defined in Definition 1, if applied to a consistent input schema, generates a consistent output schema.*

The nature of OQL queries implies that they perform a read operation on either the schema or the application data within the database. All write operations to the schema are performed via schema evolution operations, while all write operations to the application data is done via the system_function. Thus, the proof of this theorem can be derived based on Lemmas 1, 2 and 3 above.

6 Case Study of Different Schema Transformations

We have presented the SERF framework as a system that gives users:

- Extensibility: Allows users to define new transformations and does not limit them to the fixed taxonomy of schema evolution primitives provided by the underlying OODB system;
- Flexibility: Allows users to define semantics of their choice for a transformation and does not limit them to the pre-defined semantics set forth by the underlying OODB system.
- Re-usability: Allows the users to move away from writing ad-hoc programs for complex transformations and instead offers the users re-usability of their OQL code via the templates as well as by building templates composed of other templates.

Several researchers have looked at simple and complex schema transformation in the context of schema evolution [Ler96, Bré96], schema integration [BLN86, MIR93], and also for re-structuring [BB99, DKE94, DK97] in general. A large subset of these transformations involves changing more than one class, such as *inline* which inlines two classes dependent on the reference attribute, or *merge* which combines two classes based on a variety of criteria. We have been successful at modeling all transformations proposed in prior works using SERF templates. The goal of the case study presented here is to show that SERF transformations are capable of not only handling the transformations presented in literature (*inline*, *encapsulate*, *split*, *merge-union*, *merge-difference*) but are also capable of capturing different semantics as and when specified by the user. In this section we present the *merge* transformation and show how a user can specify different templates each with different semantics for creating the *merge* class. Appendix A gives additional sample templates based on the ones that we have found in the literature to further illustrate the variety of transformations achievable by SERF.

6.1 The Merge Transformation

Lerner [Ler96] defines *merge* as a compound type change involving type deletion. *Merging* deletes two or more object types and creates a new type that represents the integration of the deleted types [Ler96]. The semantics of the merge, such as the attributes based on which the objects can be combined, are built into the merge compound change. However, we note that there are several different possibilities for the structural definition of the new type as well as for its object creation. For example, the structure of the new class can be defined by doing a union of the properties of the two source classes or by an intersection of the attributes of the two classes. Similarly, object creation can be specified by a Cartesian product, by a simple join of one single property or by possibly a more complex join. By previous approaches in the literature focusing on complex transformations [Bré96, Ler96], the merge has generally been defined as *one fixed transformation.* Below in our case study we first show how using SERF templates we can achieve the semantics as defined by them and then go on to show how we can apply new semantics at both the structural level as well as the object transformation level flexibly within our SERF framework. In the merge transformation case study below we show three different semantics for achieving the structural definition of the new class (merge, difference, intersection) and then we will show two possibilities for doing the object transformations for the merge transformation.

6.2 The Merge-Union Template

Figure 9 represents the semantics of the merge-union process using an example. Here the structure of the new class `Person` is defined by a union of the properties of the two source classes `PersonInfo` and the `EmployeeInfo`. Figure 11 shows the template that is used for creating the merge-union class `JournalPaper`. This template takes five parameters: the name of the new class (merge-union) class; the two source classes; and the join pair i.e., `merge-union`(className1, className2, newClassName, className1.attr1, className2.attr2, join_op). In our example, creation of the new objects for the `JournalPaper` class is specified by the join of the `Author` objects and the `Paper` objects such that `Author.AuthorName = Paper.AuthorName`.

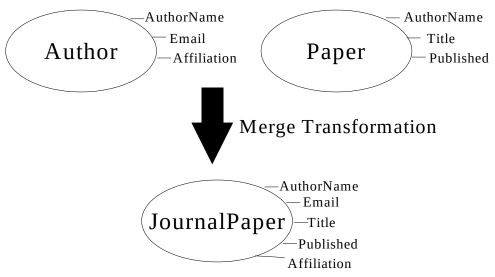


Figure 9: Merge-Union: The Structure of the New Class given by Union of the Properties of the Two Source Classes

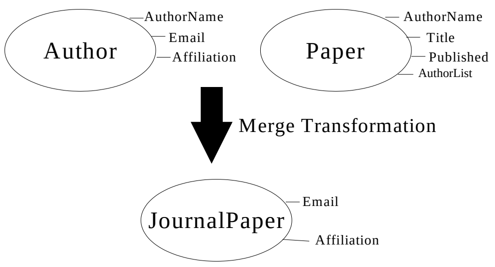


Figure 10: Merge-Difference: The Structure of the New Class given by Difference of the Properties of the Two Source Classes (Author - Paper)

6.3 The Merge-Difference Template

The Merge-Union template shows one possible semantic definition for the structure of the new class. The power of the SERF framework lies in the ease with which the framework allows the user to write new templates to express any different desired semantics other than those provided by the system. Figure 10 represents the merge-difference process using an example [11]. Here the structure of the new class `JournalPaper` is defined by the difference of the properties of the two source classes `Author` and `Paper`. Figure 12 shows the template that is used for creating the merge-difference class `JournalPaper`. The object transformation for the template is based on the same join pair as above.

It can be observed by comparing the templates that the only a few lines of oql code have been modified as marked in Figure 12. Thus the SERF framework not only offers users *extensibility* by allowing them to describe different semantics for the same transformation, but it also offers it with very little pain to the users.

[11]The structure of the source classes has been slightly altered from that shown in Figure 9 so as to make the example more meaningful.

6.4 A More Complex Merge Template

One of the key features of the SERF framework is the *re-usability* of the SERF templates. As noted above, the Merge-Union and the Merge-Difference templates share a common set of OQL statements, i.e., the creation of the actual new class, i.e., class `JournalPaper` in our running example and the object transformations are common for both the templates.

The only difference between the two templates is the set of oql statements that acquire the attribute set that must be added to the new class, `JournalPaper`. In one case (Merge-Union) it is the union of all the attributes of the two source classes; in the other case (Merge-Difference) only the attributes that are different are added to the new class. In order to capitalize on the common set of OQL statements, we create a new template for the common OQL statements as shown in Figure 13. Figures 14 and 15 show templates for Merge-Union and Merge-Difference that *re-use* the template Merge-Common to accomplish the same behavior as shown in Figures 11 and 12 respectively.

```
begin template merge_union(Class className1,
                           Class className2,
                           String newCName,
                           Attribute attr1,
                           Attribute attr2,
                           String join_op)
{
    // Retrieve all attributes of class1 and class2
    // from the Schema Repository
    define unionAttributes(cName1, cName2) as
        select distinct c.localAttrList
        from MetaClass c
        where c.className = cName1 or
            c.className = cName2;

    // Create the new merge class
    create_class ($newCName);

    // Add all the attributes to it
    for all attr in unionAttributes($className1,
                                    $className2):
        add_atomic_attribute($newCName,
                             attr.attrName,
                             attr.attrType,
                             attr.attrDefault);

    // Get the extent of a class
    define extent(cName) as
        select c.*
        from c in cName;

    // Do the object transformation
    define new_extent () as
        select obj1.*, obj2.*
        from extent($className1) obj1,
            extent($className2) obj2
        where obj1.$attr1 $join_op obj2.$attr2;

    for all obj in new_extent ():
        $newCName(obj);

}
end template
```

Figure 11: The Merge-Union Template

```
begin template merge_diff(Class className1,
                          Class className2,
                          String newCName,
                          Attribute attr1,
                          Attribute attr2,
                          String join_op)
{
    // Basic statement to get attributes of a class
    define attributes(cName) as
        select c.localAttrList
        from MetaClass c
        where c.className = cName;

    // Retrieve the difference attributes of class1
    // and class 2 from the Schema Repository
    define diffAttributes(cName1, cName2) as
        select attrs.*
        from attributes(cName1) attrs
        where not
        (attrs exists in attributes(cName2));

    // Create the new merge class
    create_class ($newCName);

    // Add all the attributes to it
    for all attr in diffAttributes($className1,
                                   $className2):
        add_atomic_attribute($newCName,
                             attr.attrName,
                             attr.attrType,
                             attr.attrDefault);

    // Get the extent of a class
    define extent(cName) as
        select c.*
        from c in cName;

    // Do the object transformation
    define new_extent () as
        select obj1.*, obj2.*
        from extent($className1) obj1,
            extent($className2) obj2
        where obj1.$attr1 $join_op obj2.$attr2;

    for all obj in new_extent ():
        $newCName(obj);
}
end template
```

Figure 12: The Merge-Difference Template

```
**begin template merge_common(Class className1,
                              Class className2,
                              String newClassName,
                              Attribute attr1,
                              Attribute attr2,
                              String join_op)**
{
    // Get the extent of a class
    define extent(**cName**) as
        select c.*
        from c in **cName**;

    // Do the object Transformation
    define new_extent () as
        select obj1.*, obj2.*
        from extent($className1) obj1,
            extent($className2) obj2
        where obj1.$attr1 $join_op obj2.$attr2;

    for all obj in new_extent ():
        $newClassName(obj);
}
**end template**
```

Figure 13: The Merge-Common Template - Shows the Common Code between the Merge-Union and the Merge-Difference templates

```
begin template merge_union(
                              Class className1,
                              Class className2,
                              String nCName,
                              Attribute attr1,
                              Attribute attr2,
                              String join_op)
{
    // Retrieve all attributes of class1 and class2
    // from the Schema Repository
    define unionAttributes(cName1, cName2) as
        select distinct c.localAttrList
        from MetaClass c
        where c.className = cName1 or
            c.className = cName2;

    // Create the new merge class
    create_class ($nCName);

    // Add all the attributes to it
    for all attr in unionAttributes($className1,
                                    $className2):
        add_atomic_attribute($nCName,
                             attr.attrName,
                             attr.attrType,
                             attr.attrDefault);

    // Do object transformation using
    // Common template
    merge_common($className1, $className2,
        $attr1, $attr2, $join_op);
}
end template
```

Figure 14: The Merge-Union Template with Embedded Merge-Common Template

```
begin template merge_diff(Class className1,
                          Class className2,
                          String nCName,
                          Attribute attr1,
                          Attribute attr2,
                          String join_op)
{
    // Basic statement to get attributes of a class
    define attributes(cName) as
        select c.localAttrList
        from MetaClass c
        where c.className = cName;

    // Retrieve the difference attributes of class1
    // and class 2 from the Schema Repository
    define diffAttributes(cName1, cName2) as
        select attrs.*
        from attributes(cName1) attrs
        where not
        (attrs exists in attributes(cName2));

    // Create the new merge class
    create_class ($nCName);

    // Add all the attributes to it
    for all attr in unionAttributes($className1,
                                    $className2):
        add_atomic_attribute($nCName,
                             attr.attrName,
                             attr.attrType,
                             attr.attrDefault);

    // Do object transformation using
    // Common template
    merge_common($className1, $className2,
        $attr1, $attr2, $join_op);
}
end template
```

Figure 15: The Merge-Difference Template with Embedded Merge-Common Template

7 Design and Implementation of the OQL-SERF System

In this section we present a brief overview of our implementation of the SERF Framework - OQL-SERF, while a more detailed description can be found in a technical report [CJR98a]. OQL-SERF, a Java based system is built using Object Design Inc.'s Persistent Storage Engine Pro 2.0 (PSE Pro 2.0) as the underlying OODB system. It uses the JDK 1.2 and has been implemented and tested on the Windows NT as well as the Linux platforms. It is based on the ODMG standard. 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 [Cat97]. A prototype of this system was demonstrated at SIGMOD 1999 [RCL+99].

7.1 System Architecture

Figure 8 presents the system design for OQL-SERF using PSE Pro 2.0 as the underlying OODB system [O'B97]. PSE, a lightweight OODB system written in Java [O'B97], has been chosen 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.
- An OQL Query Engine for querying the objects in PSE.

While we have implemented these system modules, we expect an underlying OODB system realizing the SERF framework to provide these basic capabilities. In this section we hence do not go into the details of the system modules which can be found in a separate report [CJR98a] but only focus on describing the framework modules that are the core of the SERF system.

7.2 SERF Framework Modules

The SERF Framework Modules are the core components that need to be provided by any system realizing the SERF Framework. Sections 7.2.1 and 7.2.3 describes the functionality and support needed for SERF templates and transformations.

7.2.1 Template Module

The Template Module provides all of the functionality for storing, retrieving and executing templates. Figure 16 shows the architecture for the Template Module in OQL-SERF.

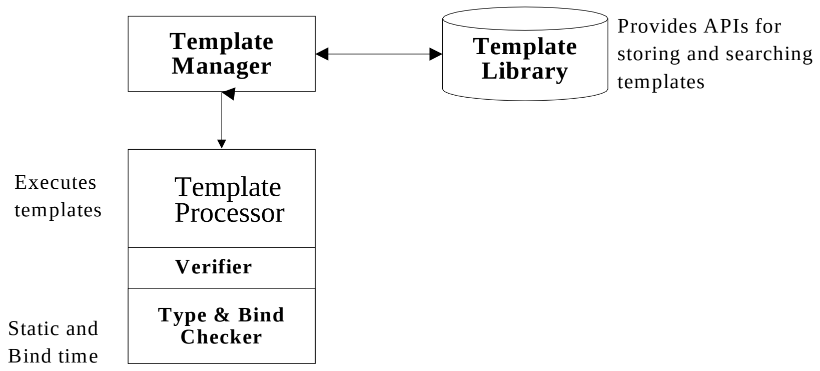


Figure 16: The Template Module.

The Template Manager. The Template Manager is the public interface of the template module. Through the template manager the user can retrieve, edit and execute already existing SERF templates as well as create and store new SERF templates. It invokes the Template Library or the Template Processor dependent on the functionality required by the user.

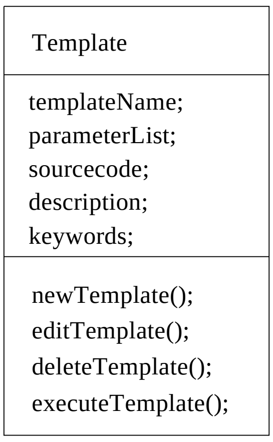


Figure 17: The Template Class.

The Template Processor. The Template Processor provides the functionality for the execution of a template. Figure 7 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 and they exist in the system as well as the correct number of parameters are supplied by the user. This is followed by a *bind-check* that 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.

7.2.2 Template Library

The SERF templates for a particular domain are collected in a Template Library. A Template Manager at any given time can manage multiple Template Libraries. For this reason `libraryName.templateName` gives the complete path for storing and retrieving a SERF template. Within a template library each SERF template is represented by a corresponding template object [12], an instance of the TemplateClass (see Figure 17). Each template object hence contains the *name*, *description*, a set of *input parameters*, a list of *keywords* and the *oql source* for its SERF template.

The Template Library currently provides the users (via the Template Manager) the capability to search for a template in the library via simple key-word search on all stored parameters of the template such as the input and output parameters, the name, and the description. We are currently in the process of defining *inheritance* and other semantic relationships between templates to facilitate a multi-level organization of the template library.

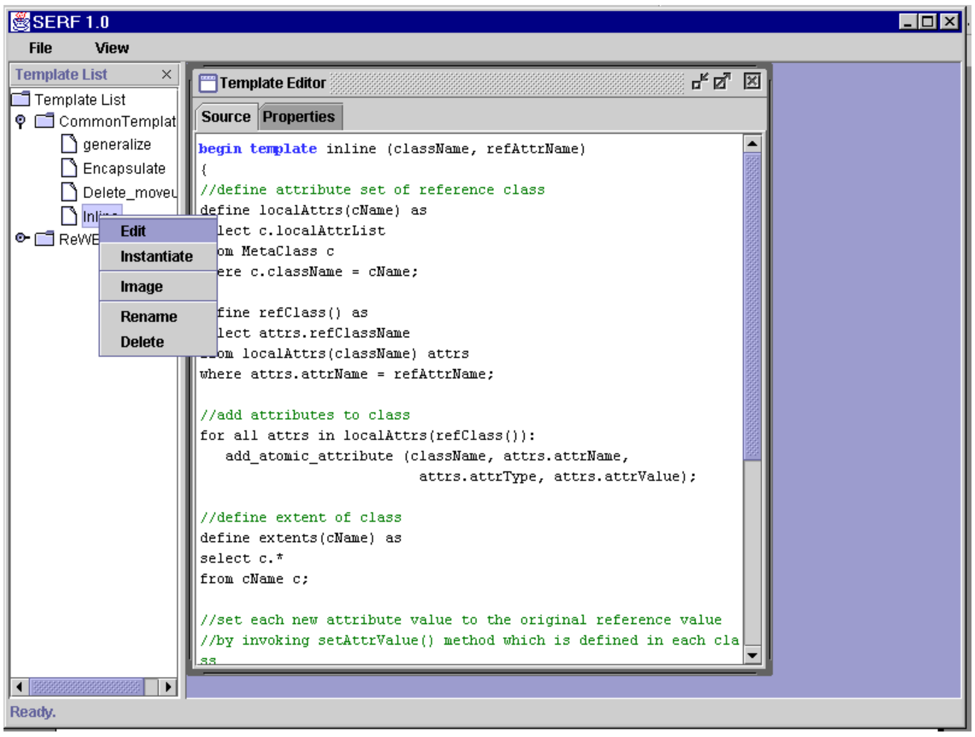
7.2.3 User Interface

For OQL-SERF version 1.0, we have designed a graphical user interface (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 as shown in Figure 18. 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. It also provides a graphical interface for viewing, searching and retrieving the templates stored in the Template Library. The GUI allows the user to utilize a schema viewer to select the parameters for instantiating a template (see Figure 19). The user can then 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

Summary. In this paper we have presented an extensible schema evolution framework for OODBs that addresses the limitations of current OODB systems that just provide a fixed set of schema evolution

[12]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.



```
begin template inline (className, refAttrName)
{
//define attribute set of reference class
define localAttrs(cName) as
lect c.localAttrList
om MetaClass c
ere c.className = cName;

fine refClass() as
lect attrs.refClassName
om localAttrs(className) attrs
where attrs.attrName = refAttrName;

//add attributes to class
for all attrs in localAttrs(refClass()):
   add_atomic_attribute (className, attrs.attrName,
                              attrs.attrType, attrs.attrValue);

//define extent of class
define extents(cName) as
select c.*
from cName c;

//set each new attribute value to the original reference value
//by invoking setAttrValue() method which is defined in each cla
ss
```

Figure 18: The Template Editor.

operations. We offer re-use of these transformations through the key concept of this framework that is a *template*, a generic, named and parameterized transformation. The transformation templates are written using a query language, like OQL, and use the schema evolution primitives and the meta-knowledge provided by the underlying OODB system. We believe the notion of transformation templates is a very powerful idea and to our knowledge this is the first time it has been introduced in the schema evolution domain. Users of any specific OODB will no longer be limited to the schema evolution primitives that are offered by the OODB. Instead, a user will be able to shop around for the best schema evolution template library for their own use, or if not available, they would be fairly easy for a user to construct. A prototype of this system was demoed at SIGMOD99 [RCL$^+$99].

Contributions In summary, the contributions of this work are:

- Identification of problem - we have identified the limitation of the current OODBs in terms of schema evolution support.
- Solution - as a solution we have presented the novel idea of templates. The concept of templates exists in the programming language domain but to the best of our knowledge we are the first to introduce and adapt this in the context of databases and in particular schema evolution.
- Consistency - in addition we have shown that templates preserve the structural consistency of the schema, if the underlying schema evolution primitives are invariant preserving.

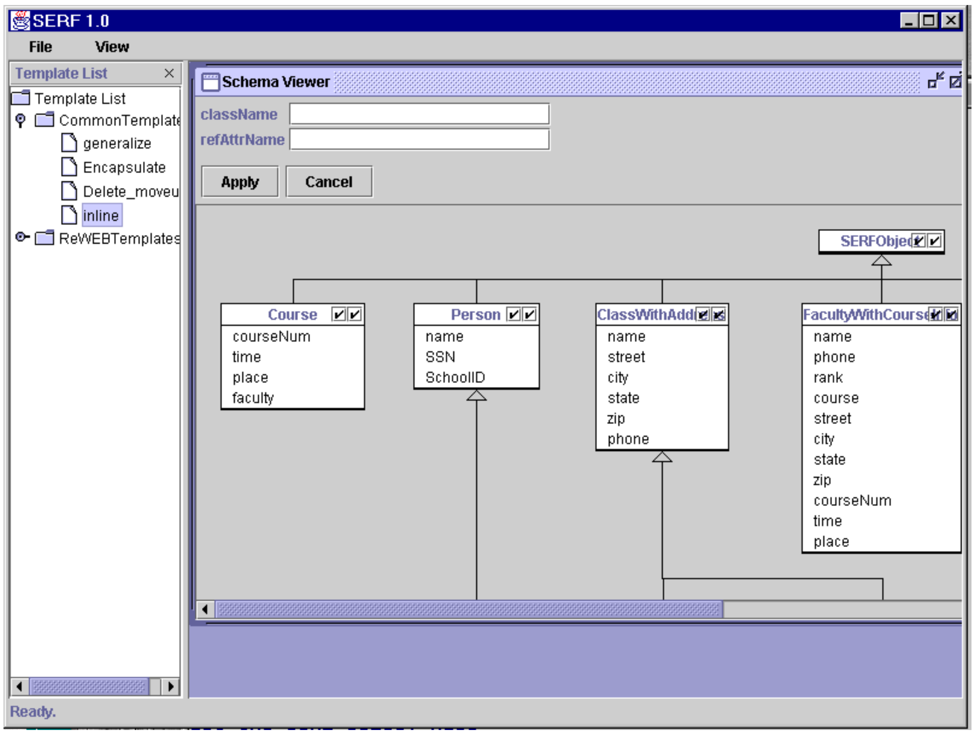


Figure 19: The Schema Viewer.

- Implementation - we have designed and implemented a prototype to show the feasibility of our approach and in the process have come up with our own minimal and complete taxonomy of schema evolution primitives [RCL+99].

- Evaluation - we have verified that our proposed notion of transformation templates is powerful via a case study.

Future Work. As part of our continuing work we would like to offer a more complete schema evolution framework. For this, we have begun to and plan to continue looking at the issues of:

- Consistency management and the SERF framework [CRH99],

- Template optimization [CNR99], and

- Application of the SERF framework to other domains as a re-structuring framework [CRCK98].

Acknowledgments. The authors would like to thank students at the Database Systems Research Group at WPI for their interactions and feedback on this research. In particular we would like to thank Keiji Oenoki for his continued work on the implementation of the SERF User Interface. We are also grateful to Anuja Gokhale, Parag Mahalley, Swathi Subramanian, Jayesh Govindrajan, Stacia De Lima, Stacia Weiner, Chandrakant Natarajan, Jin Jing, Xin Zhang and Ming Li for their help with the implementation of OQL-SERF.

References

[BB99] P.A. Bernstein and T. Bergstraesser. Meta-Data Support for Data Transformations Using Microsoft Repository. In *IEEE Bulletin - Special Issue on Database Tranformation Technology*, pages 9–14, 1999.

[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. *SIGMOD*, pages 311–322, 1987.

[BLN86] C. Batini, M. Lenzerini, and S. Navathe. A Comparative Analysis of Methedologies for Database Schema Integration. *ACM Computing Surveys*, 18(4), December 1986.

[BMO+89] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Management System. In *Object-Oriented Concepts, Databases and Applications*, pages 283–308. ACM Press, 1989.

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

[Cat97] Cattell, R.G.G and et al. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, Inc., 1997.

[CJR98a] K.T. Claypool, J. Jin, and E.A. Rundensteiner. OQL_SERF: An ODMG implementation of the template-based schema evolution framework. Technical Report WPI-CS-TR-98-14, Worcester Polytechnic Institute, July 1998.

[CJR98b] K.T. Claypool, J. Jin, and E.A. Rundensteiner. SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework. In *Int. Conf. on Information and Knowledge Management*, pages 314–321, November 1998.

[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. *Journal of Information Systems*, 23(5):279–305, 1998.

[CNR99] K.T. Claypool, C. Natarajan, and E.A. Rundensteiner. Optimizing the Performance of Schema Evolution Sequences. Technical Report WPI-CS-TR-99-06, Worcester Polytechnic Institute, February 1999.

[CRCK98] K.T. Claypool, E.A. Rundensteiner, L. Chen, and B. Kothari. Re-usable ODMG-based Templates for Web View Generation and Restructuring. In *CIKM'98 Workshop on Web Information and Data Management (WIDM'98), Washington, D.C., Nov.6*, 1998.

[CRH99] K.T. Claypool, E.A. Rundensteiner, and G.T Heineman. Extending Schema Evolution to Handle Object Models with Relationships. Technical Report WPI-CS-TR-99-15, Worcester Polytechnic Institute, March 1999.

[DK97] S.B. Davidson and A.S. Kosky. WOL: A Language for Database Transformations and Constraints. In *IEEE Int. Conf. on Data Engineering*, pages 55–65, 1997.

[DKE94] S.B. Davidson, A.S. Kosky, and B. Eckman. Facilitating Transformations in a Human Genome Project Database. In *Int. Conf. on Information and Knowledge Management*, pages 423–432, 1994.

[DZ91] C. Delcourt and R. Zicari. The Design of an Integrity Consistency Checker (ICC) for an Object Oriented-Database System. In P. America, editor, *ECOOP*, pages 97–117, 1991.

[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 *Int. 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.

[Inc93] Itasca Systems Inc. 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.

[KGBW90] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[Lau97] S.-E. Lautemann. Schema Versions in Object-Oriented Database Systems. In *Int. 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.

[LSS96] L.V.S. Lakshmanan, F. Sadri, and I.N Subramanian. SchemaSQL - A Language for Interoperability in Relational Multi-database Systems. In *vldb*, 1996.

[MIR93] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The Use of Information Capacity in Schema Integration and Translation. In *Int. Conference on Very Large Data Bases*, pages 120–133, 1993.

[MNJ94] M.A Morsi, S. Navathe, and Shilling J. On Behavioral Schema Evolution in Object-Oriented-Database System. In *Int. Conference on Extending Database Technology (EDBT)*, pages 173–186, 1994.

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

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

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

[RCL$^+$99] E.A. Rundensteiner, K.T. Claypool, M. Li, L. Chen, X. Zhang, C. Natarajan, J. Jin, S. De Lima, and S. Weiner. SERF: ODMG-Based Generic Re-structuring Facility. In *Demo Session Proceedings of SIGMOD'99*, pages 568–570, 1999.

[RLR98] E.A. Rundensteiner, A. Lee, and Y.-G. Ra. Capacity-Augmenting Schema Changes on Object-Oriented Databases: Towards Increased Interoperability. In *Object-Oriented Information Systems*, 1998.

[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*, 9(4):600–624, September 1997.

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

[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.

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

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

A SERF Template Examples

In this section we provide some more examples of SERF templates from commonly found transformations [Ler96, Bré96]. Lerner and Breche have developed a set of transformations that deals with multiple types. Here we describe SERF templates for all the transformations that can be found in [Bré96, Ler96], along with the *rename* template which is a primitive in most systems.

- **Inline -** Replace a type reference with its type definition as in Figure 5.
- **Encapsulate -** Create a new type by encapsulating parts of one or more old types as in Figure 20.
- **Merge -** Replace two or more type definitions with a new type that merges the old type definitions as in Figure 11.
- **Generalize -** Move part of a type definition from two types to a common super type as in Figure 21.
- **Specialize -** Specialize the behavior of a type by creating a new sub-type as in Figure 22.
- **Rename -** Rename an existing attribute in a class [13] as in Figure 23.

[13]This a primitive in most systems but as we show here it is not essential to provide a primitive for this as a template can accomplish it.

```
begin template encapsulate(
                    Class className,
                    List attrList,
                    String refAttrName,
                    Class refClass)
{
    // This template encapsulates a set of
     // attributes into a new class.

    // Create the encapsulated class
    create_class($refClass);

    // Add the attributes in the attrList
    // to the new class refClass.
    for all attr in $attrList:
        add_atomic_attribute($refClass,
                             attr.attrName,
                             attr.attrType,
                             attr.attrDefault);

    // Fix the old class so it now has a
    // new reference attribute
    add_atomic_attribute($className,
                         refAttrName,
                         refClass,
                         Default);

    // Get the extent of a class
    define extent(cName) as
        select c
        from c in cName;

    // Do the Object Transformation.
    for all obj in extent($className):
        obj1 = $refClass and
        obj.set($ refAttrName, obj1) and
        for all attr in $attrList:
        obj.set(obj1.attr, valueOf(obj.attr));

    // Remove the attrList from the old class
    for all attr in $attrList:
        delete_attribute($className,
                         attr.attrName);
}
end template
```

Figure 20: The Encapsulate Template

```
begin template generalize(Class subClass1,
                          Class subClass2,
                          Class superClass)
{
    // This template finds common attributes
    // between two given classes and builds
    // new super class

    // Create the super class
    create_class($superClass);

    // Find the common set of attributes
    define attributes(cName) as
        select c.localAttrList
        from MetaClass c
        where c.className = cName;

    define commonAttributes
        (cName1,cName2) as
        select attrs.*
        from attributes(cName1) attrs
        where
            (attrs exists in attributes(cName2));

    // Add all the attributes to it
    for all attr in commonAttributes($subClass1,
                                     $subClass2):
        add_atomic_attribute($superClass,
                             attr.attrName,
                             attr.attrType,
                             attr.attrDefault);

    // Add the Inheritance Relationship
    add_ISA_Edge ($superClass, $subClass1);
    add_ISA_Edge ($superClass, $subClass2);

    // Delete the local attributes
    for all attr in commonAttributes($subClass1,
                                     $subClass2):
        delete_attribute($subClass1, attr)
and
        delete_attribute($subClass2, attr);
}
end template
```

Figure 21: The Generalize Template

```
begin template specialize(Class class1,
                          Class subClass,
                          List attrList)
{
    // This template specializes into a new subclass

    // Create the sub class
    create_class($subClass);

    // Add all the attributes
    for all attr in $attrList:
        add_atomic_attribute($subClass,
                             attr.attrName,
                             attr.attrType,
                             attr.attrDefault);

    // Get the list of subclasses for class1
    define subClasses (cName) as
        select c.subClassList
        from c in MetaClass
        where c.className = cName;

    // Make these subclasses for subClass
    for all s in subClasses ($class1 ):
        add_ISA_Edge($subClass, s);

    // Make this new class a subclass of class1
    add_ISA_Edge($class1, $subClass);

    // Delete tyhe subclasses from class1
    for all s in subClasses ($class1 ):
        delete_ISA_Edge($class1, s);
}
end template
```

Figure 22: The Specialize Template

```
begin template rename(Class class1,
                      String oldAttr,
                      String newAttr)
{
    // This template renames an attribute in a class

    // Find the attribute
    define getAttr (aName, cName) as
        select attr
        from Attribute attr
        where attr.attrName = aName and
            attr.classDefinedIn = cName;

    oldA = getAttr($oldAttr, $class1);

    // Create a new attribute
    add_atomic_attribute($class1,
                         $newAttr,
                         oldA.attrType,
                         oldA.Default);

    // Get the extent of a class
    define extent(cName) as
        select c
        from c in cName;

    // Do the object transformation.
    for all obj in extent($class1):
        obj.set(obj.$newAttr,
                valueOf(obj.$oldAttr));

        delete_attribute($class1, $oldAttr);
}
end template
```

Figure 23: The Rename Template