WPI-CS-TR-99-21 August 1999

An Ounce of Prevention is Worth A Pound of Cure: Formal Verification for Consistent Database Evolution

by

Kajal T. Claypool and Elke A. Rundensteiner

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

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

An Ounce of Prevention is Worth A Pound of Cure: Formal Verification for Consistent Database Evolution*

Kajal T. Claypool and Elke A. Rundensteiner

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

Abstract

Consistency of a database is as an important property that must be preserved at all times. In most OODB systems today, application code can directly access and alter both the data as well as the structure of the database. As a consequence application code can potentially violate the integrity of the database, in terms of the invariants of the data model, the user-specified application constraints, and even the referential integrity of the objects themselves. A common form of consistency management in most databases today is to encode constraints at the system level (e.g., foreign keys), or at the trigger based level (e.g., user constraints) and to perform transaction rollback on discovery of any violation of these constraints. However, for programs that alter the structure as well as the objects in a database, such as an extensible schema evolution program, roll-backs are expensive and add to the already astronomical cost of doing schema evolution. In this paper, pre-execution formal verification of schema evolution programs is proposed as an alternative solution to the traditional rollback solution for consistency management. As part of this work we introduce the notion of *contracts*, i.e., pre- and post-conditions for an extensible schema evolution program, and demonstrate that they can be specified using a familiar language, OQL. We also demonstrate the ease and practicality of using a theorem prover for the formal verification of schema evolution programs. The theorem prover tool can be set up initially with all the information about the environment, i.e., the axioms of the database, the invariants and the basic schema evolution primitives. A writer then of an extensible schema evolution program need only supply the contracts and the program written in OQL to guarantee the correctness of their program. We highlight the main features of the verification process using a complete walk-through example. The end result of our approach is a more efficient consistency management framework that has limited overhead to the users and yet provides flexibility to safely add new schema evolution transformations to the system while assuming complete correctness.

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

*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.

1 Introduction

Consistency Specification and Management Approaches. The advances in both persistent languages as well as in OODBtechnology have aided in making the distinction between applications and databases transparent [Obj93, Tec92, Obj94]. In most OODB systems today, application code can directly access and alter both the data as well as the structure of the database. As a consequence application code can potentially violate the integrity of the database, in terms of the invariants of the data model and the user-specified application constraints, as well as the referential integrity of the objects themselves. Thus, it is required that the application programmer be aware of the invariants as well as the constraints of the database and ensure that their code does not violate them. The application programmer is hence saddled with either the manual exercise of verifying the application code, a tedious and un-reliable course mostly handled by some amount of testing of the code [Pre97]; or with writing time consuming test programs to verify that the database is in fact in a consistent state after the execution of the application code.

Support from existing databases is mostly in the form of a built-in pre-defined set of consistency definitions, such as referential integrity [MS90], programming language kinds of consistency definitions [VD91, AH90] such as assertions and exception handling mechanism in languages like C++, Java, Ada, or domain constraints over only a subset of types such as constraints on collection types [SHO95]. Relational database systems (RDBs) offer some control over constraint specification by users via the mechanism of *triggers*. RDBs then apply roll-back semantics, i.e., if a constraint is not satisfied at the end of a transaction, then the entire transaction is rolled back [EN96]. Active database systems [BCVG86, LLPS91, BK90] perhaps provide the most powerful capabilities for constraint specification in the form of event-condition-action (ECA) rules and then roll-back semantics for constraint management enforcement. While some researchers have used ECA rules to implement consistency management capabilities, the semantics of consistency management are fairly different from reactive control [SHO95, KBS, BK92].

Verification as Consistency Management Technique. However, while this roll-back based support for consistency management might be adequate for some types of application programs, it is a very expensive management strategy for schema evolution programs. Consider a large schema evolution program which may take over 24 hours [FMZ94]. A roll-back due to the detection of an erroneous condition in the 23^{rd} hour is not an attractive or a viable option. It would be much preferable to have a guarantee of success *before* executing the evolution program. To address this, in this paper we propose verification of schema evolution programs prior to their execution.

Verification of any program relies on the knowledge that given a start state, the program code will take the system to a desired final state. Thus, before verification can be applied, it becomes essential to describe these states, the start and the final states. However, most programs only describe them implicitly as part of the actual program code itself. For example, a statement such as `if subclasses == 0 then deleteClass()` indicates implicitly a start state of the system by the conditional statement. The actual code of the system-supplied schema evolution operation `deleteClass()` gives its behavior and thus implicitly the final state of the system. However, such *hard-coded* states would require inference from the program code. This in general is very difficult and inefficient and possibly intractable depending on the programming language. Moreover, any change in the domain, such as a change in the underlying object model, may force a code change in order to properly add new or adjust existing constraints. While these hard-coded constraints are still the most common approach for this problem, this re-engineering effort is a tedious, expensive and error-prone process and occurs usually as an after-thought. Moreover, it is not extensible. For instance, if a user were to write a new complex schema evolution program [CJR98b], such an approach would not scale to ensure that these programs now in turn guarantee the consistency of the database. The application programmer would now need to also hard-code their constraints within their new evolution programs.

Applying Verification to Extensible Schema Evolution Facilities. To address this problem in this paper we present a mechanism, called *contracts*, that allows the user to declaratively describe the constraints under which an evolution program can be applied as well as the expected outcomes of the program. We have adopted the notion of contracts from the programming language Eiffel [Mey92] and now apply it to assure the consistency for an extensible schema evolution framework, SERF [CJR98b]. SERF [CJR98b] allows its users to compose *arbitrarily complex* schema evolution transformations using the basic schema evolution primitives (provided by the underlying OODB system) and OQL. As the OQL query language is capable of invoking programs or methods in other languages such as Smalltalk, Java, C++ and OQL, we can bind the notion of *contracts* to these languages. We have been able to show that OQL is sufficient for expressing the contracts necessary to assure consistency of OODBs. This offers the added benefit that OQL is a standard language familiar to database developers and thus offers ease of use to writers (compared to a formal specification language).

The contracts are a key feature for the practicality of our overall verification approach. In this paper we present a formal verification mechanism using theorem proving [BHJ+96] that allows us to verify the correctness of schema evolution programs in terms of the consistency constraints (system constraints and the contracts) prior to their execution. This is a complimentary approach to current techniques of consistency management such as transaction roll-backs as described above [1]. For highly expensive programs such as schema evolution, applying a theorem prover to verify the program is efficient compared to the potential cost of transaction roll-backs. In this paper we demonstrate the practicality of our approach and present a proof of concept by showing the verification of schema evolution transformations written in OQL. To reduce the start-up of learning such a complex tool with a formal language and to make this approach more feasible and practical, we propose developing a user-friendly theorem prover tool. The user would only need to input the schema transformation with OQL contracts in order to use it.

Overview of the Paper. The rest of the paper is organized as follows: Section 2 reviews the SERF framework. Section 3 presents an example that we use through-out this paper. Section 4 presents our overall approach. Section 5 describes contracts as integrated with SERF and their advantages. Section 6 describes how to realize the key features of theorem provers for supporting the verification of schema evolution programs. Section 7 demonstrates the verification process using an example. Section 8 presents related work, and in Section 9 we give a summary of our work and future direction.

2 Review of the SERF Framework

In this section we present a brief overview of the extensible schema evolution framework, SERF [CJR98b] which motivates this work. Schema evolution support today provide a fixed set of simple schema evolution operations and thus are not able to cover all changes that a user might want to make to an object schema. In SERF we address this limitation and allow users to customize semantics of transformations in a *flexible* and *re-usable* manner [CJR98b]. Our approach is based on the hypothesis that complex schema evolution transformations can be decomposed into a sequence of basic evolution primitives, where each basic primitive is an invariant-preserving atomic operation with fixed semantics provided by the underlying OODB system. To effectively combine these primitives and perform arbitrary transformations on objects within a complex transformation, we rely on a standard query language, namely OQL [Cea97]. We have demonstrated that a language such as OQL, which can be applied to both data and meta-data, is sufficient for accomplishing schema evolution, thereby re-using existing technology. The SERF system is proposed as a value-added re-structuring layer on top of existing database systems.

A SERF transformation *flexibly* allows a user to define different semantics for any type of schema transformation (see Figure 1). However, these transformations are *not re-usable* across different classes or different schemas. To address this, we have introduced the notion of templates in the SERF framework [CJR98b]. A template uses the fact that meta knowledge can be accessed in the OODB's system dictionary (as per the ODMG standard). In addition a template is encapsulated via a name and a set of parameters to make transformations *generic* and *re-usable* (Figure 2). Thus, when the example SERF template in Figure 2 is instantiated with actual schema elements it results in the SERF transformation shown in Figure 1.

An implementation of the SERF framework, called OQL-SERF, has been developed at Worcester Polytechnic Institute [CJR98a]. It is based on the ODMG standard and uses the ODMG object model, the ODMG Schema Repository definition, and OQL. The system is being implemented entirely in Java and uses

[1] The transaction roll-backs here provide a safety net for any violations that may go undetected during the verification process.

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

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

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

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

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

```
begin template inline (className, refAttrName)
{

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

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

   // get all attributes in refAttrName and add to className
   for all attrs in localAttrs(refClass)
       add-attribute($className, attrs, attrs.type, attrs.default)


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

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

   delete-attribute ($className, $refAttrName);
}

end template

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

Figure 2: The Inline Template.

Object Design's Persistent Storage Engine (PSE) for Java as its back-end database [O'B97]. The system was demonstrated at SIGMOD'99 [RCL+99] in May 1999 and will be released to public domain shortly.

3 Running Example: The Delete-Class Evolution Program

Consistency management requires the specification of comprehensive constraints that must be checked to ensure that they are not violated. For example, the basic schema evolution primitive *delete-class(C_i)* [PS87] can only be applied when the class $\mathtt{C}_i$ is a *leaf* class, i.e:

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

However, while this is a necessary and sufficient constraint for the delete of the `HomeAddress` class specified in the schema depicted in Figure 3, it is no longer a sufficient stipulation for a schema that contains references to other classes as in Figure 4.

For example, the delete of the *leaf* class `Address` in the schema in Figure 4 is a valid evolution operation as per the constraints specified Equation 1. This however causes dangling references and hence compromises the consistency of the system by violating both the structural integrity (schema-level) and the referential integrity (object-level) of the system. Equation 2 provides a comprehensive list of constraints that should perhaps be checked prior to the deletion of a class.

$$\left.\begin{array}{rcl} sub(C_i) & = & \emptyset \\ in-degree(C_i) & = & 0 \\ \forall o_i \in \mathbf{extent}(t): obj-in-degree(o_i) & = & 0 \quad for \quad t = type(C_i) \end{array}\right\} \tag{2}$$

Today, while most state-of the art OODB systems allow the use of reference attributes, the *delete-class()* primitive in these systems only needs to satisfy *one* constraint, i.e., the class being considered for deletion must be a leaf class [Obj94, Obj93]. From the example shown here it can be seen that this can cause inconsistencies.

4 Overall Approach: Contract-Based Pre-Execution Verification

In this section we introduce a two-step approach that helps address the problems of hard-coded constraints and transaction roll-backs in the context of schema evolution programs. We propose an approach:

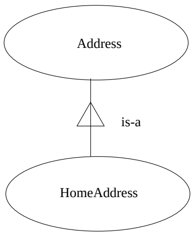

Figure 3: Example Schema With No References

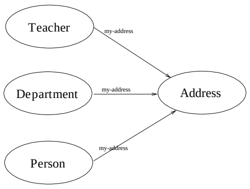

Figure 4: Example Schema Showing References

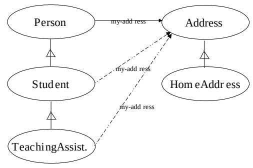

Figure 5: Sample Schema Containing References via Inheritance

- that allows the user to declaratively describe both the constraints under which an evolution program can be applied and the behavior of the program, and
- that allows the user to use a formal verification mechanism using theorem proving to verify the correctness of their schema evolution programs.

4.1 Explicit Declarative Constraint Specification

From Equation 2 we can observe that when considering and compensating for the effects of reference attributes as in the `delete-class` schema evolution primitive program, the actual functionality of the program itself is not being changed. For example, for the `delete-class` example in Section 3, the program is still simply deleting a class. However, the conditions or the constraints under which a class can be deleted are changed. Namely to compensate for the consistency violations that could occur with reference attributes, the `delete-class` program must now also consider all the classes that are referring to a particular class before the class can be deleted. Thus, we now propose elevating the constraints which must hold true for the successful execution of a program outside the actual program code a la software contracts in Eiffel [Mey92]. The software contracts are classified into two categories **pre-conditions** and **post-conditions**. Pre-conditions are the constraints that must be satisfied prior to the execution of the program code, and post-conditions describe the state of the system after the execution of the program code.

We thus, as part of our approach, integrate the notion of *contracts* into SERF templates. Hence, any schema evolution program, irrespective of the language it was written in, would be encapsulated and packaged as a SERF template in our system which captures the pre-and post-conditions for the schema evolution program. For example, the program for the schema evolution primitive `delete-class` would have a SERF template that explicitly details its pre- and post-conditions as presented in Equation 2 irrespective of any hard-coded constraints that may be part of the system-level implementation code of the primitive. Using this approach, a user can now also describe the pre- and post-conditions for any new complex schema evolution program that they may write, such as an *inline* SERF template (see Figure 2).

Advantages of Contracts. Contracts in general make the program easier to understand by others as they abstract away from the actual code and are explicit declarations. Programs with contracts thus achieve (potentially) higher level of re-use. In our specific case, SERF templates with contracts provide faster updates to the OODB system when the underlying object model is updated, for example with the modeling support of relationships. Namely, we would now simply add additional declarative constraints to existing schema evolution primitive templates rather than having to update the system code. Moreover, a mechanism such as formal verification can now be put to work to detect erroneous conditions prior to the execution of the schema evolution programs. This would help avoid the cost of roll-backs in cases of failure. Finally, the *post-conditions* allow the user to verify that from the initial state of the system the final expected state is reached, and thus determine the correct functioning of the program.

4.2 Consistency Management Via Verification

To fully exploit the potential benefits of *contracts*, the second step of our approach utilizes them to verify the correctness of a SERF template. When a template is applied to a system that satisfies the template's pre-conditions, then the correctness of the template is verified if after execution it produces the state of the system as specified by its post-conditions. This aids in ensuring the consistency of the OODB systems in the presence of user-specified schema changes not only at the user-level (the state of the system is as per user's specification via the post-conditions) but also at the system-level (the new state of the system does not violate the invariants of the system). One approach to verification is via extensive testing. However, while widely used in industry [Pre97], this approach is error-prone as it is not feasible to test all possible cases. Instead we take a more formal approach to verification.

We base our formal verification on theorem proving techniques [BHJ+96] and demonstrate the practicality of this technique for our application space. The theorem prover verification process is applied to SERF templates at runtime as a pre-cursor to execution. As per the theorem prover, the set of *pre-conditions* indicates a state of the system that must be satisfied prior to execution and the set of *post-conditions* the final state of the system that the execution of the template program must achieve.

Advantages of Formal Verification as a Consistency Management Tool. The key advantage of formal verification is to prevent a problem (a consistency violation) from occurring rather than either trying to patch up after the fact or trying to recover from it via roll-backs. In the case of schema evolution programs, there is a potential performance win over the strategy of doing roll-backs. One of the big advantages of SERF are the SERF Template Libraries, a resource that can be re-used by a large software community. Verification allows us to guarantee the correctness of the templates in this library.

5 Contracts in SERF Templates

Contracts bring forth many advantages as a mechanism that allows for easier reasoning and as a tool that aids the formal verification of a program. As stated earlier, we have extended the SERF template to include the notion of contracts. To allow for a seamless integration with SERF as well as for ease of usability, we investigated the use of OQL as a user-friendly constraint specification language and found it to be very adequate for our purposes. This offers the advantage of staying within the same language for SERF templates, namely, the standard query language that should be familiar to anyone working with database management systems (DBMS). Thus, the users don't have to learn a new language. For programs that are not written in OQL, this is still advantageous given the simplicity of OQL compared to formal specification languages. Contracts are divided into two categories **pre-conditions** and **post-conditions** where the pre-conditions are placed prior to any code and the post-conditions representing the state of the database after the execution of the code are placed after the code.

Pre-Conditions for a Contract SERF Template. The constraints, termed *pre-conditions*, are placed prior to any code (OQL statement) in a template. The *pre-conditions* are a logical expression separated from the actual OQL statements representing the template body by means of the keyword **requires** and are expressed in OQL [2]. Each statement of Figure 6 shows the constraints for the *delete-class* primitive as expressed in Equation 2 as a pre-condition.

The pre-conditions must all hold true before the actual schema evolution program (`delete-class` in this example) can be executed.

Post-Conditions for a Contract SERF Template. The behavior of the primitives is declared by *post-conditions*, a set of contracts that appear after the body of the actual schema change code at the end of the SERF template. These post-conditions are preceded by the keyword **ensures:** and describe the exact changes that are made to the schema by the evolution program. We extend the pre-condition verification process to also do the post-condition verification, which then is responsible for validating that the schema change specified actually accomplished what it set out to do.

Figure 7 shows the post-conditions of the *delete-class* primitive. Here the class C_i is deleted and the system must ensure that this has in fact occurred by checking the conditions shown in Figure 7. For example, after deletion we do not expect the schema to have any class C_i nor have any class referring via a

[2] The functions used here those shown in Table 3.

delete-class (C_i)
{
    requires:
        exists x in $\mathcal{C}$:
            x = C_i and
        exists y in **types**($\mathcal{C}$):
            y = $\sigma(C_i)$ and
        sub(C_i) = 0 and
        in-degree(C_i) = 0 and
        for all z in **extent**(C_i):
            obj-in-degree(z) = 0

    `schema evolution program`
    `(delete-class) here`
}

Figure 6: Pre-Conditions for Delete-Class Evolution Program

delete-class (C_i)
{   `schema evolution program`
    `(delete-class) here`

    ensures:
        for all a in *out-paths*(C_i):
            not (exists b in *in-paths*(a.class):
                b.class = C_i) and
        for all c in *super*(C_i):
            not(exists d in *sub*(c):
                d = C_i) and
        not(exists e in $\mathcal{C}$:
            e = C_i) and
        not(exists e in **types**($\mathcal{C}$):
            e = $\sigma(C_i)$)
}

Figure 7: Post-Conditions for the Delete-Class Evolution Program

relationship to the class C_i. Thus, together the *pre-conditions* and the *post-conditions* declaratively define both the constraints that must be satisfied prior to execution of the specified schema change as well as the behavior of the change itself.

A declarative approach, such as offered by contracts, combined with SERF templates brings another dimension to the extensibility of SERF. Using this mechanism we are able to describe contracts for any evolution program using SERF as a wrapper. Figure 8 shows the *Inline* template of Figure 2, with its contracts, the conditions under which it can be applied and the resultant schema after its application. The pre-conditions here also describe the semantics under which the user thinks that the template would be valid. For example, in Figure 8, the last pre-condition states that the inline transformation can only be done for reference attributes that are not self-referential.

begin template inline (className, refAttrName)
{
    requires:
        exists x in C:
            x = className and
        exists y in N(className):
            y = refAttrName and
        exists z in C:
            z = domain(refAttrName) and
        not (className = domain(refAttrName))

        Body of Inline Template

    ensures:
        exists x in C:
            x = className and
        not (exists y in N(className):
            y = refAttrName) and
        for all a in N(domain(refAttrName)):
            exists z in N(className):
                z = a
}
end template

Figure 8: Inline Template with Contracts.

6 Theorem Prover for SERF Templates

6.1 Introduction

As noted in Section 4.1, contracts provide a mechanism for declaratively describing constraints and behavior. In order for contracts to be an effective basis for constraint management, they need a mechanism that performs their checking. Eiffel [Mey92] provides support for contract checking as part of its language compiler itself. However, such an approach is limited to a particular language and also does not take into account the outside invariants such as the invariants of the object model that may need to be maintained. In our work we instead apply theorem provers [GSW95, BHJ+96] as a mechanism for validating the correctness of schema evolution programs via the use of contract SERF templates.

Proving correctness of programs requires knowledge about the initial (starting) state of a system, the final states that need to be reached and a set of functions that are applied to reach the targeted states. Additional constraints such as information and axioms about the given environment (such as the object model) form a tangential part of the functions and must be maintained at all times. When repetitive applications of functions are required, the final state of the system after the application of a previous function is regarded as the initial state. If the final state is reached, the program is verified to be correct.

Theorem proving approaches verification by formalizing (a) a model of computation, (b) the specification and (c) the rules of inference [BHJ+96]. The axioms and other knowledge about the environment comprise the model of computation, the pre- and post- conditions are the specification, i.e., the initial state of the system as well as the final targeted state that must be reached. The rules of inference are the functions that help reason about the validity of the path from the initial to the final states of the system. Table 1 shows what these components correspond to in the context of our SERF database environment. In the following subsections, we sketch out these three components for our problem domain to show the viability of theorem provers for SERF templates (see also Table 1).

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

Table 1: Theorem Prover Components for the SERF Environment.

6.2 Model of Computation for SERF Templates

The model of computation formally describes the environment in which the theorem prover is being applied. The SERF framework is based on the ODMG object model [Cea97]. Hence, the theorem prover must be provided with a formal definition of the ODMG object model, its invariants and the functionality of each of the system dictionary functions as described in Table 3. This model of computation is part of the setup of the theorem prover system and thus would be generated once a-priori for the contract SERF system. It would only need to be modified if and when there is a change in the environment itself, for example if the object model changes. While different theorem provers use different languages [GM93, ORS92], for the purpose of this paper, we assume the language of the theorem prover to be set-theoretic.

6.2.1 The Object Model

Table 2 gives a brief description of the components of the object model. In general, a schema that ties all this together is as defined below.

Definition 1 *A* schema *is a 7-tuple* $\mathcal{S} = (\mathbf{C}, \sigma, \prec, \mathbf{M}, \mathbf{G}, \mathcal{R}, \alpha)$ *where*

- $\mathbf{G}$ *is a set of* names *disjoint from* $\mathbf{C}$*,*

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

Table 2: Components of the Object Model

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

A more thorough treatment of the formal description of the object model can be found in [AHV95].

System Functions. Table 3 describes some helper functions which are a part of the system definition. For the theorem prover, the behavior of each of these functions is precisely defined in a set-theoretic language.

Term	Description
super(t)	The set of all direct supertypes of type t
sub(t)	The set of all direct subtypes of type t
super (t)*	The set of all direct and indirect supertypes of type t
sub (t)*	The set of all direct and indirect subtypes of type t
in-paths(t)	The set of all paths <c,r> referring to type t
in-degree(t)	The count of all paths referring to type t
out-paths(t)	The set of all paths <t,r> going out of type t
out-degree(t)	The count of all paths going out of type$t$
obj-in-degree(o_i)	The number of objects referring to the object o_i
obj-out-degree(o_i)	The number of objects being referred to by the object o_i

Table 3: Notation for Axiomatization of Schema Changes

Invariants of the Object Model. Table 4 presents the invariants for the ODMG object model.

6.3 Specification of SERF Templates

A theorem prover requires the specification of the initial state of the system as well as the final state that needs to be verified. The contracts, i.e., the pre- and post- conditions as defined in Section 5, fulfill these requirements by providing an initial state (the pre-conditions) that must be valid and an expected final state (the post-conditions) that must be met. However, the theorem prover expects its inputs to be expressed in a formal language. Hence these contracts need to be converted to the language of the theorem prover, i.e., in

Axioms	Description
Rootedness	$T =$ **root** $\mid \forall\, t \in$ **types**$(\mathcal{C})$, $t \in sub^*(T)$
Closure	$\forall\, t \in$ **types**$(\mathcal{C})$, $super^*(t) \in$ **types**$(\mathcal{C}) \mid t =$ root)
Pointedness	$\perp =$ **leaf** $\mid sub(\perp) = \emptyset$
Nativeness	$N(t) =$ The set of native (local) properties of type t
Inheritance	$H(t) =$ The set of inherited properties of type t
Distinction	$c \in \mathcal{C} \mid c$ is unique
Degree	total in-degree (T-IN) / total out-degree (T-OUT) is an invariant

Table 4: Invariants of the Model

our case to a set-theoretic language. We have found based on an extensive study of templates [CR99] that our OQL contracts can easily be expressed in set-theoretic notation. Figure 9 represents the specification of the `delete-class` evolution program from Figures 6 and 7 and Figure 10 the `inline` template from Figure 8 in set-theoretic notation. As part of our work, we are developing a *translator* tool that automatically translates the OQL contracts to this set-theoretic language. This elevates the burden of writing contracts in a formal, mathematical language from the user.

begin template delete-class (C_i)
{
requires:

$C_i \in \mathcal{C} \wedge$
$\sigma(C_i) \in \mathbf{types}(\mathcal{C}) \wedge$
$sub(C_i) = 0 \wedge$
$in\text{-}degree(C_i) = 0 \wedge$
$\forall\, o_i \in \mathbf{extent}(t)$
$(obj\text{-}in\text{-}degree(o_i) = 0;)$

delete-class-program (C_i);

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

Figure 9: Delete-Class Primitive Template with Set-Theoretic Contracts

begin template inline (C_s, r_s)
{
requires:

$C_s \in \mathcal{C} \wedge$
$\sigma(C_s) \in \mathbf{types}(\mathcal{C}) \wedge$
$r_s \in N(C_s) \wedge$
$\mathrm{domain}(r_s) \in \mathcal{C} \wedge$
$C_s \neq \mathrm{domain}(r_s);$

Body of inline template

ensures:
$C_s \in \mathcal{C} \wedge$
$\sigma(C_s) \in \mathbf{types}(\mathcal{C}) \wedge$
$r_s \notin N(C_s) \wedge$
$\mathrm{domain}(r_s) \notin \mathcal{C} \wedge$
$\forall\, x \in N(\mathrm{domain}(r_s))$
$(x \in C_s);$
}

Figure 10: Inline Template with Set-Theoretic Contracts

6.4 Rules of Inference

The rules of inference are operations that move the system from one given state to another state, i.e., code segments that take the system from an initial specification to a final specification. The body of a SERF template, i.e., the actual schema evolution functions and OQL code, are hence the rules of inference in our system.

For example each step (statement, OQL, or schema evolution function) of the inline template as shown in Figure 1 is a rule of inference[3]. Each of these rules is applied one at a time to a given state of the system. For

[3]As a first step, we consider only a restrictive set of OQL for verification purposes as discussed in Section 6.3.

example, the pre-conditions given in Figure 10 represent the initial state of the system that must be satisfied prior to any code checking. Here, the class `className` and the reference attribute `refAttrName` must exist otherwise it is meaningless to proceed with the verification. Given this initial state, we proceed to apply the first evolution change in the template, the `add-attribute` (statement 1). As part of our work, we have wrapped each individual change primitive program in a SERF template with their specific contracts (see Figures 9 and 10for example of this.). Hence forth we consider the contract SERF template for each evolution primitive program. Thus, in this example the initial state of the system must meet the pre-conditions for the `add-attribute`. The post-conditions specified by the `add-attribute` represent the final state S_n and are indicative of the behavior of the `add-attribute` change primitive. Figure 17 in Appendix A gives the specification for the `add-attribute` function.

A subsequent evolution change, `add-attribute`, uses this state S_n as its initial state against which its preconditions must match. Table 5 represents the schema evolution operations that we consider as rules of inference in our system. Appendix A gives the specification for each of the evolution primitive programs.

Evolution Primitive	Description
add-class$(c, \mathcal{C})$	Adds new class c to $\mathcal{C}$ in the schema **S**
delete-class(c)	Deletes class c from $\mathcal{C}$ in the schema **S**
add-ISA-edge(c_x, c_y)	Adds an inheritance edge from c_x to c_y
delete-ISA-edge(c_x, c_y)	Deletes the inheritance edge from c_x to c_y
add-attribute(c_x, a_x, t, d)	Add attribute a_x of type t and default value d to class c_x
delete-attribute(c_x, a_x)	Deletes the attribute a_x from the class c_x
add-reference-attribute(c_x, r_x, c_y, d)	Add unary relationship from class c_x to class c_y named r_x with the default value d
delete-reference-attribute(c_x, r_x)	Delete unary relationship in class c_x named r_x
form-relationship(c_x, r_x, c_y, r_y)	Promotes the specified two unary relationships to a binary relationship
drop-relationship(c_x, r_x, c_y, r_y)	Demotes the specified binary relationship to two unary relationships

Table 5: Taxonomy of Basic Schema Evolution Primitives for Classes, Attributes and Inheritance Hierarchy.

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

7 Formal Verification Process: Application to Inline Template

In this section, we illustrate the working of the theorem prover by a step by step verification of a template (namely the `inline` template from Figure 10), thereby showing how theorem provers can be applied to our domain for verification of schema evolution transformations. This is an automated process where the computation model (Section 6.2) and the rules of inference (Section 6.4 and Appendix A) have already been setup as part of the tool. The user only needs to input the specification contracts and the template code in OQL.

The theorem prover proves the correctness of the inline transformation shown in Figure 2 by first proving three theorems and then tying them together to prove the correctness of the inline transformation itself (the fourth theorem). Each of the theorems specifies the properties of one of the evolution programs in the inline transformation.

Schema Evolution Primitive: add-attribute. The preconditions from the contract specification in Figure 10 that must hold for `add_attribute(`C_S`, `a_x`, type, default)` are given in Equation 3 [4]:

$$\left.\begin{array}{rcl} C_s & \in & \mathcal{C}\wedge \\ \sigma(C_s) & \in & \mathbf{types}(\mathcal{C})\wedge \\ a_x & \notin & N(C_s) \end{array}\right\} \tag{3}$$

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

$$\left.\begin{array}{rcl} C_s & \in & \mathcal{C}\wedge \\ \sigma(C_s) & \in & \mathbf{types}(\mathcal{C})\wedge \\ a_x & \in & N(C_s)\wedge \\ (\forall \mathbf{x} & \in & sub^*(C_s) \\ a_x & \in & H(\mathbf{x})) \end{array}\right\} \tag{4}$$

Theorem 1 *If the add-attribute program is applied to arguments satisfying preconditions given in Equation 3, then the program results satisfy the postconditions given in Equation 4.*

Proof: Assume that the preconditions in Equation 3 hold. The `add-attribute` function adds the attribute a_x to the class C_s, i.e.:

$$\left. a_x \quad \cup \quad N(C_s) \right\} \tag{5}$$

The `add-attribute` function also adds a_x to all the subclasses *sub* (C_s)* of C_s. Assume that the preconditions in Equation 3 also hold for all *sub* (C_s)*. Hence we have:

$$\left.\begin{array}{rcl} \forall \mathbf{x} & \in & sub^*(C_s) \\ (a_x & \cup & H(\mathbf{x})) \end{array}\right\} \tag{6}$$

Here Equations 5 and 6 show the altered states of the system after the execution of each `add-attribute` function. From Equations 5 and 6, we have the desired postconditions as specified in Equation 4. □

In Figure 2, **line 1** shows a `for-all` loop used to copy all the attributes of the class C_d to the class C_s [5]. At the end of the `for-all` loop, with repetitive application of the `add-attribute`, the desired state is given by the postconditions in Equation 7.

$$\left.\begin{array}{rcl} C_s & \in & \mathcal{C}\wedge \\ \sigma(C_s) & \in & \mathbf{types}(\mathcal{C})\wedge \\ \forall \mathbf{x} & \in & N(C_d) \\ (x & \in & N(C_s))\wedge \\ \forall \mathbf{y} & \in & sub^*(C_s) \\ (x & \in & H(\mathbf{y})) \end{array}\right\} \tag{7}$$

Theorem 2 *If the* `add-attribute` *function is correct as per Theorem 1, then repetitive execution of the* `add-attribute` *function with different arguments result in a cumulative effect such that postconditions given in Equation 7 are satisfied.*

Proof: (Proof By Induction)
Base Case: Assume that the class C_d [6] has only one attribute *a*. This reduces the `for all` statement (line 1) to a simple `add-attribute(`C_s`, `*a*`, `*a.attrType*`, `*a.defaultValue*`)`. We know by Theorem 1 that if the pre-conditions given in Equation 3 hold for these arguments, then post-conditions as given in Equation 4 will also hold.
Induction Hypothesis: Assume that the theorem holds true when class C_d has `k-1` attributes and they are added to class C_s, i.e., the post-conditions given in Equation 7 are satisfied for `k-1 add-attribute` applications.

[4] These are repeated from Figure 10 for convenience.
[5] These are the `Person` and the `Address` classes in the example.
[6] C_d is the class that is being referred to by the reference attribute r_x in class C_s.

Induction: Prove that the post-condition in Equation 7 holds when the class C_d has `k` attributes.

We know that the postconditions (7) hold when class C_d has `k-1` attributes and they are added to the class C_s. To add the k^{th} attribute to class C_s we do: `add-attribute(`C_s`,` a_k`,` $a_k.attrType$`,` $a_k.defaultValue$`)`. We know by Theorem 1 that if this satisfies the pre-conditions given in Equation 3, then the post-conditions in Equation 4 hold true (Base Case). Combining the post-conditions for the addition of `k-1` attributes (Induction Hypothesis) with the post-conditions of the Base Case, we get the post-conditions as given in Equation 7. □

Schema Evolution Function delete-attribute. First we give the initial state of the system, i.e., the preconditions in Equation 8 that must be satisfied for the function `delete-attribute(`C_S`,` a_x`)`:

$$\left.\begin{array}{rcl} C_s & \in & \mathcal{C} \wedge \\ \sigma(C_s) & \in & \mathbf{types}(\mathcal{C}) \wedge \\ a_x & \in & N(C_s) \end{array}\right\} \tag{8}$$

After the execution of the `delete-attribute`, the final state of the system is given in the post-conditions in Equation 9.

$$\left.\begin{array}{c} a_x \notin N(C_s) \wedge \\ \forall \mathbf{x} \in sub^*(C_s) \\ (a_x \notin H(\mathbf{x})) \end{array}\right\} \tag{9}$$

Theorem 3 *If the delete-attribute function is applied to arguments satisfying precondition (8), then the function results satisfy the postcondition (9).*

Proof: The proof of this can be given in a manner similar to Theorem 1.

Schema Evolution Function delete-class. The necessary preconditions that must hold for the function `delete-class(`C_s`)` are given in Equation 10:

$$\left.\begin{array}{rcl} C_s & \in & \mathcal{C} \wedge \\ \sigma(C_s) & \in & \mathbf{types}(\mathcal{C}) \wedge \\ sub(C_s) & = & 0 \wedge \\ in-degree(C_s) & = & 0 \wedge \\ \forall o_i \in \mathbf{extent}(t) : (obj-in-degree(o_i) & = & 0) \end{array}\right\} \tag{10}$$

The desired postconditions (11) after the application of the function `delete-class` are as given in Equation 11.

$$\left.\begin{array}{rcl} \forall < C_x, r_x > & \in & out-paths(C_i) \\ (< C_i > & \notin & in-paths(C_x)) \wedge \\ \forall C_x & \in & super(C_i) \\ (C_i & \notin & sub(C_x)) \wedge \\ C_i & \notin & \mathcal{C} \wedge \\ \sigma(C_i) & \notin & \mathbf{types}(\mathcal{C}) \end{array}\right\} \tag{11}$$

Theorem 4 *If delete-class function is applied to arguments satisfying preconditions in Equation 10, then the function results satisfy the postconditions in Equation 11.*

Proof: The proof of this can be given in a similar manner to the previous one.

The Inline Transformation Thus, to verify the correctness of the inline transformation, we chain the results of Theorems 1, 2, 3 and 4. The overall pre-conditions for this are as given by Equation 12 [7]. The execution of the inline transformation must result in the final state as specified by Equation 13 [8].

[7] These are the overall pre-conditions for the `inline` template in Figure 10.

[8] These are the post-conditions for the `inline` template given in Figure 10.

$$\left.\begin{array}{lcl} C_s & \in & \mathcal{C}\wedge \\ a_x & \in & N(C_s)\wedge \\ C_d & = & domain(a_x)\wedge \\ C_d & \in & \mathcal{C}\wedge \\ C_s & \neq & C_d \end{array}\right\} \quad (12)$$

$$\left.\begin{array}{lcl} C_s & \in & \mathcal{C}\wedge \\ C_d & \notin & \mathcal{C}\wedge \\ \forall \mathbf{x} & \in & N(C_d) \\ (x & \in & N(C_s))\wedge \\ \forall \mathbf{y} & \in & sub^*(C_s) \\ (x & \in & H(\mathbf{y})\wedge \\ a_x & \notin & C_s) \end{array}\right\} \quad (13)$$

Theorem 5 *If Theorems 1, 2, 3, and 4 are satisfied in the order specified, then the inline transformation satisfies the postconditions given in Equation 13.*

Proof: The proof for this can be given by a combination of the postconditions in Equations 7, 9 and 11. □

Using theorem proving techniques as shown for the template here it is possible to verify the correctness of any given template. If at any point one of the sub-theorems is not satisfied, i.e., if Theorems 1, 2, 3, or 4 are not satisfied, the verification process is aborted and the template is not permitted to be executed.

8 Related Work

Basic Schema Evolution. The goal of schema evolution research is to allow schema evolution mechanisms to change not only the schema but also the underlying objects to have them conform to the modified schema. The first taxonomy of primitive schema evolution operations was defined by Banerjee et al. [BKKK87]. They defined consistency and correctness of these primitives in the context of the Orion system. Until now, current commercial OODBs such as Itasca [Inc93], GemStone [BMO+89], ObjectStore [Obj93], and O_2 [Tec94] all essentially handle a similar set of fixed evolution primitives; though based on their own respective object models. In recent years, the advent of more advanced applications has led to the need for support of complex schema evolution operations. Both Breche and Lerner [Bré96, Ler96] have investigated the issue of complex operations. Lerner [Ler96] has introduced compound type changes in a software environment, i.e., focusing on type and not on object instance changes. The SERF framework [CJR98b] presents a flexible way of doing transformations by means of a re-usable, parameterized SERF template. However, none of these systems provide any form of constraint specification or management other than hard-coded invariants.

Formal Verification Mechanisms. Formal verification is a powerful mechanism for providing proofs of correctness for programs. There is a large body of work [GSW95, ORS92, Bla98, GM93] that has looked into making such mechanisms somewhat semi-automated, called *theorem provers*. Work has focused on formalizing software semantics and proving theorems about code and algorithms [GSW95, Bla98]. However, no work has yet been done in applying formal verification as a technique for maintaining the consistency of the database during schema evolution as done by our work.

Consistency Management. Support from ODBMS mostly follows the support that is already provided by programming languages in terms of consistency definitions [VD91, AH90] such as assertions and exception handling mechanisms in languages like C++, Java and Ada. Relational database systems (RDBMS) offer some additional support in the form of *triggers* but only support roll-back semantics, i.e., if a constraint is not satisfied at the end of a transaction, then the entire transaction is rolled back [EN96].

Active database systems [BCVG86, LLPS91, BK90] provide event-condition-action (ECA) rules that are a mechanism for detecting the occurrence of some event and responding to it by some action. While some researchers have used ECA rules to implement consistency management capabilities, the semantics of consistency management are fairly different from reactive control [SHO95, KBS, BK92]. Consistency management activities are a required part of any computation in which constraints are enforced and failure to satisfy the constraints may invalidate the activity. The failure to complete the activity associated with an ECA rule, however, may not necessarily invalidate the activity associated with it. While it may be possible to implement such semantics manually in some systems, it is not desirable to do so.

Much research has also been done in consistency management for software process languages. Tarr et al. [TC98] have developed a consistency management system which allows for the specification of consistency conditions and the degree of inconsistency tolerable by the user.

However, all above consistency management systems accept the fact that inconsistencies happen and try to rectify the problem after the fact. We approach the problem by attempting to prevent the problem from occurring by means of a pre-execution verification mechanism. Our approach could be combined with any of these above approaches where these would provide a safety net.

9 Conclusions

In this paper, we describe a verification process for an extensible schema evolution system (SERF) and its transformation programs (SERF templates) as a measure for assuring the consistency of the OODB system. The practicality and the feasibility of the approach has been shown. While verification of programs has an up front cost for the verification, consistency management in situations such as for schema evolution programs which are extremely expensive to execute via prevention is a more effective and efficient mechanism compared to transaction roll-backs.

We have demonstrated how consistency management via verification can be done when the programs are schema transformation programs. The verification technique however can be applied for different types of programs and can thus be utilized for consistency assurance in general by any database system.

References

[AH90] T. Andrews and C. Harris. Combining Language and Database Advances in an Object-Oriented Development Environment. In *Readings in Object-Oriented Database Systems*, pages 186–196, 1990.

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

[BCVG86] A. Buchmann, R. Carrera, and M. Vazquez-Galindo. A Generalized Constraint and Exception Handler for Object-Oriented CAD-DBMS. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 38–49, 1986.

[BHJ+96] P.J. Black, K.M. Hall, M.D. Jones, T.N. Larson, and P.J. Windley. A Brief Introduction to Formal Methods. In *Proceedings of CICC*, pages 377–380, 1996.

[BK92] N.S. Barghouti and G.E. Kaiser. Scaling Up Rule-Based Development Environments. *International Journal on Software Engineering and Knowledge Engineering 2(1)*, pages 59–78,, March, 1992.

[BK90] F. Bancilhon and W. Kim. Object-Oriented Database Systems: In Transition. In *SIGMOD RECORD*, volume 19, December 90.

[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.

[Bla98] P.E. Black. *Axiomatic Semantic Verification of a Secure Web Server*. PhD thesis, Brigham Young University, February 1998.

[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. In *Centre for Advanced Studies Conference*, pages 108–122, November 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.

[CR99] K.T. Claypool and E.A. Rundensteiner. An Ounce of Prevention is Worth A Pound of Cure: Formal Verification for Consistent Database Evolution. Technical Report WPI-CS-TR-99-21, Worcester Polytechnic Institute, July 1999.

[EN96] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, Inc., 1996.

[FMZ94] 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.

[GM93] M.J.C Gordon and T.F. Melham. Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, 1993.

[GSW95] Y. Gurevich, N. Soparkar, and C. Wallace. Formalizing database recovery. In *COMAD*, 1995.

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

[KBS] G.E. Kaiser and I.Z. Ben-Shaul. Process Evolution in the MARVEL Environment. In Schafer [14].

[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.

[LLPS91] G. Lohman, B. Lindsay, H. Pirahesh, and K.B. Schiefer. Extensions to StarBurst: Objects, Types and Rules. *Communications of the ACM*, 34(10):95–109, 1991.

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

[MS90] D. Maier and J. Stein. Development and Implementation of an Object-Oriented Database System. In *Readings in Object-Oriented Database Systems*, pages 167–185, 1990.

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

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

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

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *11th CADE, Lecture Notes in Artificial Inteliigence*, pages 748–752, 1992.

[Pre97] R.S Pressman. *Software Engineering: A Practitioner's Approach.* McGraw Hill Company, 1997.

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

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

[SHO95] S.M. Sutton, D. Heimbigner, and L.J. Osterweil. APPL/A: A Language for Software Process Programming. *ACM Transactions on Software Engineering*, 3(4):221–286, 1995.

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

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

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

[VD91] S.L. Vandenberg and D.J. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity and Inheritance. In *SIGMOD*, pages 158–167, 1991.

A Taxonomy of Schema Evolution Operations

We also present here the contract-serf templates for each of the primitives specified in Table 5.

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

$C_i \notin \mathcal{C} \wedge$
$\sigma(C_i) \notin \textbf{types}(\mathcal{C})$

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

ensures:
$C_i \in \mathcal{C} \wedge$
$\sigma(C_i) \in \textbf{types}(\mathcal{C})$
$C_i \in sub(\text{root}) \wedge$
}

Figure 11: Add-Class Primitive Template with Contracts

delete-class (C_i)
{
requires:

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

delete-class-primitive (C_i)

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

Figure 12: Delete-Class Primitive Template with Contracts

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

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

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

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

Figure 13: Add-ISA-Edge Primitive Template with Contracts

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

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

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

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

Figure 14: Delete-ISA-Edge Primitive Template with Contracts

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

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

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

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

Figure 15: Form-Relationship Primitive Template with Contracts

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

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

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

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

Figure 16: Drop-Relationship Primitive Template with Contracts

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

requires:

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

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

ensures:

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

}

Figure 17: Add-Attribute Primitive Template with Contracts

delete-attribute (C_s, a_x)
{

requires:

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

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

ensures:

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

}

Figure 18: Add-Attribute Primitive Template with Contracts

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

requires:

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

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

ensures:

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

}

Figure 19: Add-Reference-Attribute Primitive Template with Contracts

delete-reference-attribute (C_s, r)
{

requires:

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

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

ensures:

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

}

Figure 20: Delete-Reference-Attribute Primitive Template with Contracts