International Symposium on Object Technologies for Advanced Software, pages 176-193, Kanazawa, Japan, November 1993. Springer Verlag.

Maintaining Behavioral Consistency during Schema Evolution

Paul L. Bergstein and Walter L. Hürsch*
College of Computer Science, Northeastern University
360 Huntington Avenue #CN237, Boston MA 02115
{ pberg | huersch }@ccs.neu.edu

Abstract

We examine the problem of how to ensure behavioral consistency of an object-oriented system after its schema has been updated. The problem is viewed from the perspective of both the strongly typed and the untyped language model. Solutions are compared in both models using C++ and CLOS as examples.

1 Introduction

Schema evolution and transformations have recently received increasing attention in the literature in both the area of object-oriented languages and especially in the area of object-oriented database systems: [Opd92, Ber92, Ber91, Cas91, CPLZ91, DZ91, Bar91, LH90, AH88, BKKK87, PS87, SZ86]. Most of this work has been done from the object-oriented database point of view where the focus is naturally on the structural, rather than behavioral, aspects of the evolving schema. Systems such as ORION [BKKK87], GemStone [PS87], and OTGen [LH90] guarantee the correctness of the performed schema changes and reflect the impact on the persistent instances in the database (structural consistency). However, none of them considers the impact of schema updates on existing programs (behavioral consistency).

In this paper we consider the problem of behavioral consistency for an important subset of possible schema transformations. The transformations in this subset are the schema extensions defined in [LHX93, Ber91]. We chose these transformations for three reasons. First, they have the desirable property that the transformed schema's consistency with the old objects either is maintained or can be easily restored. For object-oriented database design, this means that the database does

*Walter Hürsch's research has been generously supported by Mettler-Toledo AG

not need to be repopulated, or that the repopulation can be easily accomplished. In either case, no information from the old database is lost. Second, the extension transformations reflect a significant set of transformations that commonly occur in practice. Third, they can be decomposed into a sequence of primitive transformations.

The strategy we employ to solve the behavioral consistency problem relies heavily on the third property. Our approach is to divide a given extension into a sequence of primitives and then solve the problem for each of the primitives in turn. Behavioral consistency is investigated for two very different language models: strongly typed and untyped languages. We compare solutions in the two models using C++ and CLOS as examples. As one might expect, the problem is much more difficult for the strongly typed model.

The paper is organized as follows. Section 2 provides a brief description of the employed data and language models. The third section reviews the extension relation and its associated primitive transformations. In section 4 we propose a solution for untyped languages. We also present a partial solution for strongly typed languages and discuss some of the remaining problems. The last two sections present related work and conclusions.

2 The Demeter data model

2.1 Data Model

The data model used in this paper is the Demeter Kernel model which is formally defined in [LBSL91]. The Demeter Kernel model uses two kinds of classes: construction and alternation classes, and two kinds of relationships between classes: kind-of and part-of relationships.

Only the construction classes are instantiable, so every object must be an instance of some construction class. The alternation classes are used to model the union of object sets defined by the construction classes. This is often natural when modeling an application domain. For example (see Figure 1–Original, page 7), in an object-oriented drawing program the tool used to select and draw shapes on a canvas might be either a selection tool, a tool for drawing rectangles, or a tool for drawing ovals. So the objects that can be stored in the tool part of a Canvas object are either OvalTool, RectTool, or SelectTool objects. Alternation classes are used to define such unions. Each class which is an element of the union is called an alternative of the alternation class. One can think of an alternation class as an abstract superclass in a typical class-based object-oriented programming language, with the alternatives as immediate subclasses.

Any class may have various attributes represented by part classes. These “parts”

may be thought of as an abstraction of instance variables in a typical object-oriented programming language. Each alternation class must have at least one alternative or "kind". The "kinds" of an alternation class are represented by its subclasses.

If an alternation class has parts, they are implemented by inheritance in the subclasses. For example, each tool in our drawing application has a mouse interface. This common part is expressed as a single part in the alternation class which is shared (inherited) by all of its alternatives (subclasses). Thus, a kind-of relation in the Demeter Kernel model also implies an inheritance relation. Since alternation classes are not instantiable, it is only possible to inherit from abstract classes.

The classes and their relations are defined by a class dictionary graph. Construction classes are represented by rectangles and alternation classes are represented by hexagons in a class dictionary graph.

Part-of relations are expressed as directed edges called construction edges from a class to each of its part classes. The construction edges are drawn as thin arrows. Each part must have a name, and the construction edges are labeled with the part names in a class dictionary graph. In Figure 1–Original, for example, a DrawWindow is a construction class with two parts: a ShapeList called shapes, and a Screen called canvas where the shapes are to be displayed.

Kind-of/inheritance relations are expressed by directed edges called alternation edges from an alternation class to each of its alternatives. The alternation edges are drawn with thick arrows.

Legal class dictionary graphs must satisfy two independent conditions: (1) No class may inherit from itself; that is, there must not be any cyclic path consisting only of alternation edges. (2) There must not be any class which has two or more parts (including inherited parts) with the same name. The first condition implies that a class definition may not depend upon itself in a circular fashion. The second condition disallows "overriding" or "shadowing" of instance variables. It guarantees that the part names in each class are unambiguous.

A class dictionary graph may be used to easily generate a set of class definitions (minus method declarations) in any class-based object-oriented programming language. This may be done either by hand or automatically by using a tool like the Demeter System^{TM} . In the latter case, the declaration and implementation of many commonly useful "generic" methods may also be automatically generated.

2.2 Language Model

In this paper we consider (informally) two language models: untyped and strongly typed. As representative examples we consider CLOS (Common Lisp Object System) and C++, respectively. For simplicity, we consider the class definitions and the methods of a class separately, although some languages might require forward declarations of methods in the class definitions.

A class dictionary graph is essentially a language-independent set of class definitions, and the translation to a particular programming language is a straightforward process. The kind-of relations defined by the class dictionary graph are implemented by declaring a corresponding inheritance relation in the class definitions. In most languages, this means that if there is an alternation edge from A to B, then class B is declared to inherit from class A in the definition of class B. Part-of relations are implemented by instance variables. For each part of a class, an instance variable is declared whose name is the same as the part name. In the case of a typed language, the part's type is declared to be the corresponding class. For example, the class definition for ShapeList from the class dictionary graph in Figure 1–Original would be written in C++ or CLOS as:

```
      C++ Version                              CLOS Version
class ShapeList : public List {          (defclass ShapeList (List)
protected:                                  (firstShape restShapes))
  Shape* firstShape;
  List* restShapes;
};
```

Our two language models share several common features:

- The parts of an object are implemented as *references*.
- Any object can send another object any message for which the receiving object has a corresponding method. In C++ terminology, all methods are "public".
- Each method is attached to exactly one class. In CLOS terminology, each method has exactly one "specialized parameter", i.e. there are no "multi-methods".
- Any method available to an alternation class is also available to each of its alternatives through inheritance.
- Inherited methods may be overriden (specialized) in a subclass. In C++ terminology all methods are "virtual".
- Every object has access (through its methods) to all of its own parts, and to the parts of other objects of the same class. This level of encapsulation is equivalent to "protected" instance variables in C++.

3 Schema Extension Transformations

This section informally introduces two important kinds of schema transformations. One kind consists of the object-extending class transformations, presented formally in [LHX93]. The other kind consists of the object-preserving transformations, presented formally in [Ber91]. The latter kind is a special case of the former in that any object-preserving transformation can be regarded as an object-extending transformation. Thus, both kinds can be called schema extension tranformations. Schema extensions are defined as a relation on class dictionary graphs. This relation can be decomposed into a set of eight primitive relations that was shown to be correct, minimal and complete [Ber91, LHX93]. The completeness guarantees that for any two schemas in an object-extending relation there exists a sequence of primitive transformations that transforms the original into the extended schema. Since the completeness proofs are constructive, there also exists an algorithm to find the sequence. The primitive schema transformations will be used in the subsequent section to determine their impact on the behavioral consistency of a program.

3.1 The extension relation

For the following discussion it is important to remember that all alternation classes are abstract and only instances of construction classes can be assigned to a part. Thus, even if a construction edge points to an alternation class A, the only objects that can be assigned to the part are instances of construction classes that are subclasses of A.

Informally, two class dictionary graphs G_1 and G_2 are in an **object-equivalence relation** if they both define the same set of objects. Consequently, G_1 and G_2 must satisfy these conditions: (1) G_1 and G_2 have the same set of construction classes. (2) A construction class A of G_1 has a (inherited or direct) part b if and only if its corresponding class in G_2 has a (inherited or direct) part b . (3) An instance can be assigned to part b of class A in G_1 if and only if the instance can also be assigned to part b of class A in G_2.

As an example of two class dictionary graphs in an object-equivalence relation, consider Figures 1–Original and 1–Object-equivalent. Note that both class dictionary graphs contain the same construction classes. Furthermore, each construction class has the same parts and to each part one can assign the same instances. In particular, in both class dictionary graphs, instances of classes RectTool, OvalTool, and SelectTool can be assigned to part inputTool attached to class Screen.

Two class dictionary graphs G_1 and G_2 are in an **extension relation**, such that G_2 extends G_1, if they satisfy these conditions: (1) The set of construction classes

of G_2 is a superset of the set of construction classes of G_1. (2) If a construction class A of G_1 has a (inherited or direct) part b, then its corresponding class in G_2 has a (inherited or direct) part b . (3) If an instance can be assigned to part b of class A in G_1, then the instance can also be assigned to part b of class A in G_2. An example of two class dictionary graphs in an extension relation is given in Figures 1–Object-equivalent and 1–Extended.

As a consequence of the above definitions the following relationship holds between extension and object-equivalence. Class dictionary graph G_1 is object-equivalent to class dictionary graph G_2 if and only if G_1 is extended by G_2 and G_2 is extended by G_1.

The object-preserving transformation is composed of the five primitive operations: (1) Deletion of useless alternation, (2) Addition of useless alternation, (3) Abstraction of common parts, (4) Distribution of common parts, and (5) Part replacement [Ber91]. The extension relation is composed of the above five primitives and, in addition, the three primitives: (6) Part generalization, (7) Part addition, and (8) Class addition [LHX93].

We briefly summarize the semantics of the above primitives.

Deletion of useless alternation (DUA) An alternation class is "useless" if it has no incoming edges and no outgoing construction edges. In other words, an alternation class is useless if it is not a part of any class, and defines no parts for any class to inherit. If an alternation class is useless it may be deleted by the DUA primitive. An example of a DUA operation is the deletion of the alternation class Tool shown in the transition from the partially drawn class dictionary graph in Figure 2–PRP to the class dictionary graph in Figure 1–Extended.

Addition of useless alternation (AUA) This is the inverse operation of DUA. An alternation class can be added to a class dictionary graph along with outgoing alternation edges to any other classes. An example of an AUA operation is the addition of the two alternation classes DrawingTool and CanvasTool (Figure 1–Original to Figure 2–AUA).

Abstraction of common parts (ACP) If B_i $(1 \leq i \leq n)$ are all the alternatives of an alternation class A and each of them has a part c of class C, then ACP deletes all the construction edges $B_i \xrightarrow{c} C$ $(1 \leq i \leq n)$ and replaces them with a new construction edge $A \xrightarrow{c} C$. Intuitively, if all of the immediate subclasses of a class A have the same part, that part is moved up the inheritance hierarchy so that each of the subclasses will inherit it from A. An example of the ACP operation is the abstraction of the common part

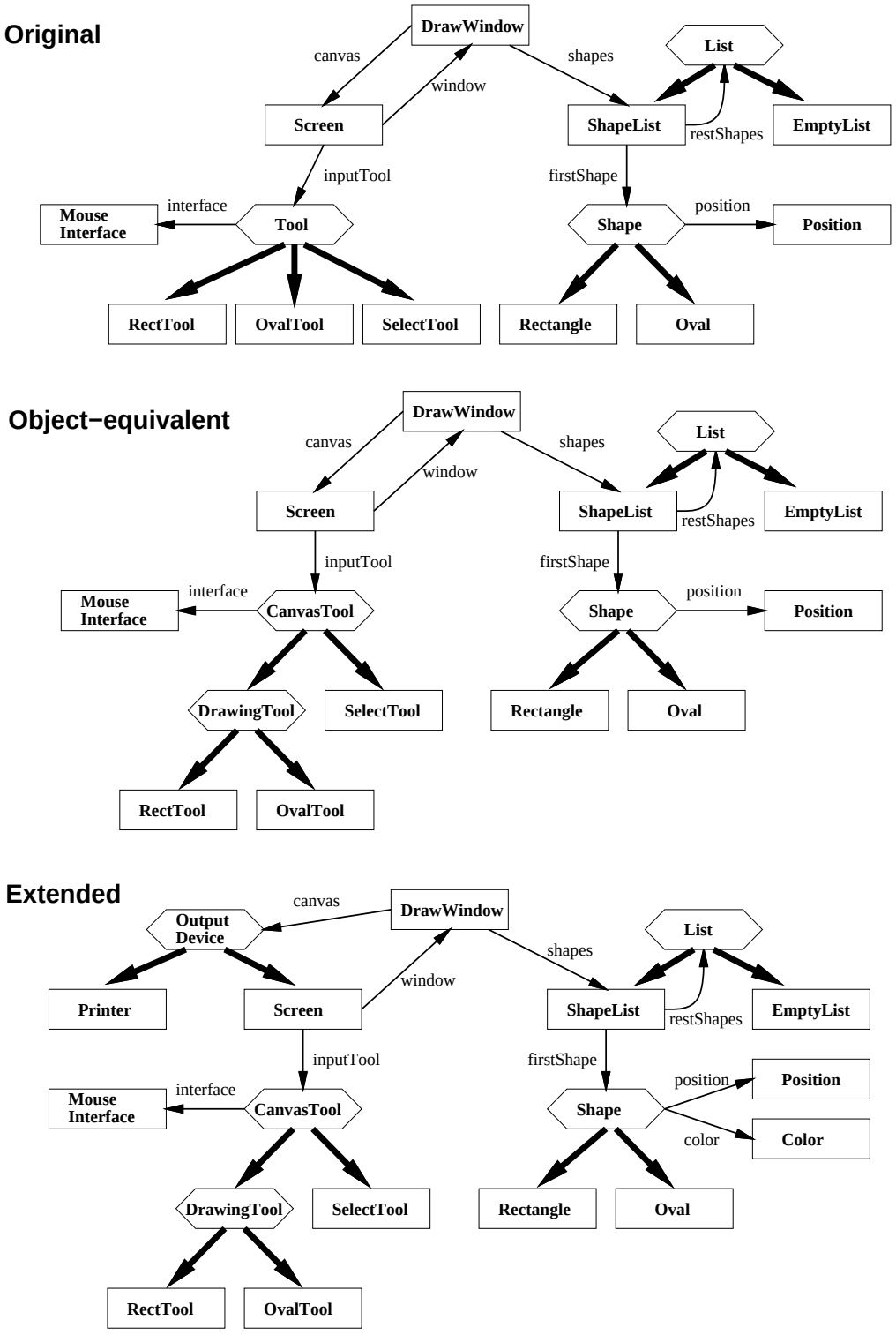

Figure 1: Extending a class dictionary graph

interface from the classes RectTool, OvalTool, SelectTool to their common superclass CanvasTool (Figure 2–DCP to Figure 2–ACP).

Distribution of common parts (DCP) This is the inverse of ACP. DCP deletes an outgoing construction edge $A \xrightarrow{c} C$ from an alternation class, A, and adds for each alternative B_i of A, a new construction edge $B_i \xrightarrow{c} C$. An example of DCP is the distribution of the part interface from class Tool to its subclasses RectTool, OvalTool, SelectTool (Figure 2–AUA to Figure 2–DCP).

Part replacement (PRP) If the set of construction classes that are subclasses of an alternation class A is the same as the set that are subclasses of another alternation class A', then PRP may delete any construction edge $X \xrightarrow{a} A$ and replace it with a new construction edge $X \xrightarrow{a} A'$. Intuitively, if two classes A and A' have the same set of instantiable (construction) subclasses then the definable objects do not change when A is replaced by A' in the definition of a part. An example of PRP is the rerouting of edge inputTool from class Tool to class CanvasTool (Figure 2–ACP to Figure 2–PRP).

Class addition (CAD) CAD adds to the existing class dictionary graph new classes and edges with the restriction that no old class may obtain new outgoing edges or new incoming alternation edges. An example of CAD is the addition of the classes Printer, `OutputDevice` and Color along with the outgoing alternation edges from `OutputDevice` (Figure 1–Object-equivalent to Figure 1–Extended).

Part addition (PAD) If the classes A and B already exist in a class dictionary graph, then PAD adds a new construction edge $A \xrightarrow{b} B$; that is, the class A obtains a new part b of class B. An example of PAD is the addition of the part color to the class Shape (Figure 1–Object-equivalent to Figure 1–Extended).

Part generalization (PGN) If a class C is a subclass of some alternation class B, then PGN reroutes a construction edge $A \xrightarrow{p} C$ to $A \xrightarrow{p} B$. In other words, PGN generalizes the domain of part p. An example of PGN is the generalization of part canvas from class Screen to the class OutputDevice (Figure 1–Object-equivalent to Figure 1–Extended).

3.2 Structural Consistency

Each of the primitive transformations, except part addition, maintains the structural consistency of the object base; that is, all the objects remain consistent with the transformed schema. When a part is added to a class A by a part addition, then structural consistency must be restored by adding an instance of that part's

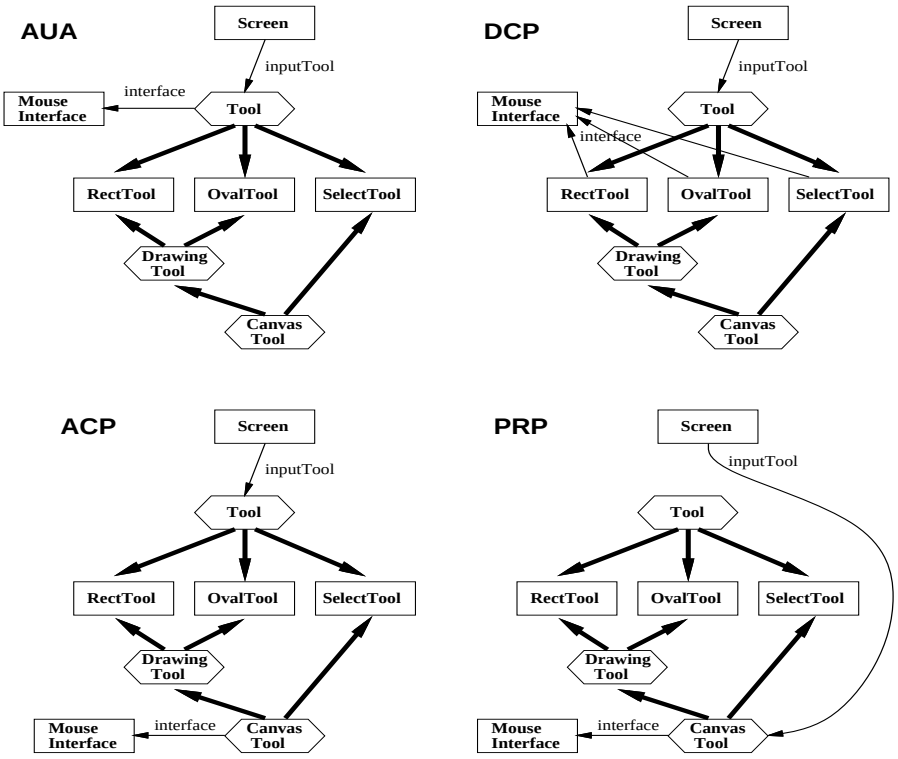

Figure 2: Steps in the object-preserving transformation

class to every instance of class A. The added object can either be some default object or specified by an object transformation function defined by the user.

4 Code Transformations

In this section we discuss how application code can be automatically updated after a class dictionary graph has been transformed or extended. The approach we take is to first reduce the transformation to a sequence of primitives. We then update the code *incrementally*, in steps that parallel the primitive transformations. Reduction to a sequence of primitives can be easily accomplished by following the constructions of the completeness proofs given in [LHX93] and [Ber91].

For each primitive transformation, we consider the rules that should be followed to update the application code so that it will meet all of the original requirements. Of course, if we wish to *extend*, rather than simply maintain the original functionality, it will be necessary to hand code some of the extension. Even so, a maintenance tool based on the primitive transformations could be used to do most of the work and generate hints for code that should be modified by hand.

4.1 Untyped Language Model

In the untyped language model the code transformations are very simple. Consider the example of the transformation of the schema in Figure 1–Original to the extended schema in Figure 1–Extended.

4.1.1 Addition of useless alternation classes

The first primitives in the sequence obtained by reducing the transformation are addition of the "useless" alternation classes DrawingTool and CanvasTool (Figure 2–AUA). The addition of these abstract classes does not require any modification of the code.

4.1.2 Distribution of common parts

In the next step (Figure 2–DCP), the interface part of the Tool class is distributed down the inheritance hierarchy to the classes RectTool, OvalTool, and SelectTool. Once again, there is no need to modify the code. Note that there may be methods attached to class Tool that refer to the interface part. In a strongly typed language such as C++, the method would no longer compile, since the part would be undefined within the scope of the method. In an untyped language such as CLOS, however, the symbol *interface* is bound at run time when the method is invoked in response to a message to a RectTool, OvalTool, or SelectTool object. Since Tool is abstract, the method can never be invoked in response to a message to a Tool object, and no run time errors occur.

4.1.3 Abstraction of common parts

When the part is moved up the new inheritance hierarchy to the CanvasTool class (Figure 2–ACP) by abstraction of common parts, there is still no need to modify the code. Every reference to interface in the RectTool, OvalTool, and SelectTool classes is still valid due to inheritance.

4.1.4 Part replacement

In the next step, the part class of Screen's inputTool is changed from Tool to CanvasTool by part replacement (Figure 2–PRP). Of course, every object that instantiates the inputTool part of a Screen must still be an instance of one of the three construction classes: RectTool, OvalTool, and SelectTool. Therefore any message that was sent to inputTool in the original code will still be understood after the class transformation and, once again, there is no need to modify existing code.

4.1.5 Deletion of useless alternations

Now that the Tool class has no incoming edges and no outgoing construction edges, it is considered "useless", and may be deleted. Note that the "useless" designation is only relevant from a data modelling point of view, since the class may have important methods attached. If the class is deleted to produce the

schema in Figure 1–Object-equivalent, the functionality of the methods attached to the class must be preserved. In the simplest case, we consider only primary methods and don't allow a method to explicitly call a method defined in a superclass (i.e. call-next-method in CLOS). In this case each method can be copied to each of the immediate subclasses that does not override it. Now every object will respond to messages in the same way after the "useless" class is deleted.

Suppose, for example, that the Tool class has a method called getPosition which is inherited in each of its subclasses:

```
(defmethod getPosition ((self Tool))
    (getPosition (slot-value self 'interface)))
```

In this case, the getPosition method is copied from the Tool class to the RectTool, OvalTool, and SelectTool classes:

```
(defmethod getPosition ((self RectTool))
    (getPosition (slot-value self 'interface)))
(defmethod getPosition ((self OvalTool))
    (getPosition (slot-value self 'interface)))
(defmethod getPosition ((self SelectTool))
    (getPosition (slot-value self 'interface)))
```

If there is another alternation class that covers the same set of construction classes as the "useless" alternation, the method could just be copied to that class instead. In the example, we could just copy the getPosition method from the Tool class to the CanvasTool class, so that the three methods above would be replaced with:

```
(defmethod getPosition ((self CanvasTool))
    (getPosition (slot-value self 'interface)))
```

If we wish to allow "before" and "after" methods, then any before method in the "useless" class can be prepended to the before method in each subclass or the primary method if the subclass has no before method. After methods are appended to the after methods in each subclass, or the primary method if there is no after method. If we allow "call-next-method", then in each subclass, every occurrance of call-next-method can be removed and the "next-method" defined in the "useless" class inlined in its place.

4.1.6 Class and part addition

Extension of a class dictionary graph by class addition or part addition does not require any modification of existing code. In the current example, addition of the classes OutputDevice, Printer, and Color (Figure 1–Extended) does not effect the application code. When the color part is added to the Shape class, existing code will continue to provide the same functionality. In this case, however, it is likely that methods attached to the Shape, Rectangle, and Oval classes would be extended to make use of the new color information. For example, if there are methods attached to these classes for drawing the shapes in black and white, they will still function properly, but the additional code required to produce color renderings would have to be added by hand.

4.1.7 Part generalization

Part generalization causes a problem similar to, but more serious than, part addition. When the part class of DrawWindow's canvas part is generalized from Screen to OutputDevice (Figure 1–Extended), the original code will continue to function properly as long as every DrawWindow continues to use a Screen as its output device. This is the case for all DrawWindow objects that were present in the old object store and possibly updated subsequently by an object transformation (see section 3.2) after the schema transformation. However, if new DrawWindow objects are introduced that use Printer output devices, messages to the canvas part will not be understood. Since it is not possible, in general, to automatically generate correct methods for the new part classes, warnings should be added to the code wherever a DrawWindow method accesses its canvas part to indicate that the part has been generalized.

4.2 Typed Language Model

For the discussion of code transformations in the typed language model, illustrated for the example of C++, we make the following simplifying assumptions on the mapping from the class dictionary graph schema to the C++ class definitions: (1) All parts are defined as protected data members. (2) In congruence with the Demeter data model, all alternation classes are mapped to abstract superclasses. (3) All member functions of alternation classes are defined as virtual member functions. (4) All data members are defined as pointers or references.

4.2.1 Addition of useless alternation classes

As for the untyped language model, the change in class definitions due to the addition of a useless alternation class requires no modification to the methods.

4.2.2 Distribution of common parts

We have seen above that, in the untyped language model, the distribution of a part from a superclass to its subclasses does not require any change in the methods. However, in the strongly typed model, the distributed part will no longer be defined within the scope of the superclass. Therefore any superclass method that accesses the part will no longer compile.

To restore behavioral consistency, every method in the superclass that accesses the part must be distributed along with the part to each of the subclasses that does not override the method. Since an object with the statically declared type of the superclass may be sent a message whose method has been distributed, we must replace the original method with a "pure virtual" method.

Constructor and destructor methods in C++ behave similarly to before and after methods in CLOS and may be treated much like the distribution of before and after methods in deletion of useless alternation classes. The body of a super-

class constructor accessing a distributed part is inlined at the beginning of each subclass constructor, and replaced with an empty body.

Consider, for example, what happens when the interface part of the Tool class is distributed down the inheritance hierarchy to the classes RectTool, OvalTool, and SelectTool (Figure 2–DCP). Suppose that the Tool class defines the method:

```
Position *Tool::getPosition() { return interface -> getPosition(); }
```

Then Tool::getPosition is replaced by a pure virtual function, and the following new methods are added:

```
Position *RectTool::getPosition() { return interface->getPosition(); }
Position *OvalTool::getPosition() { return interface->getPosition(); }
Position *SelectTool::getPosition() { return interface->getPosition(); }
```

4.2.3 Abstraction of common parts

As in the untyped model, no change is necessary for the implementation of member functions, since data members are defined to be protected and hence member functions of any subclass that accessed an abstracted part still have access through inheritance.

4.2.4 Part replacement

In the untyped language model, part replacement does not require any modification of the code since the objects that can be assigned to the replaced part are unchanged. However, in a typed language, the part replacement implies a change in the type declaration of the part. Two problems occur in this case. First, messages sent to the part might no longer be understood since there may be no such method known to the part's new class. Second, wherever the part is involved in an assignment statement, function call (as a passed parameter), or function return (as the returned value), the part's new type will no longer be compatible.

The first problem can be solved by supplying, for each method defined in the part's old class, a corresponding pure virtual function in the part's new class. Since each construction subclass now inherits methods from both the part's new and old class, it must provide its own method to resolve the ambiguity in favor of its original (possibly inherited) method.

The second problem requires that objects be converted to the appropriate type in assignment statements, function calls, and function returns. Note that simple casting will not work in C++ under multiple inheritance.

Consider what happens when the part class of Screen's inputTool is changed from Tool to CanvasTool by part replacement. Suppose that the following methods were originally defined:

```
void Tool::handleMouseClick(DrawWindow *win) = 0;
```

```
void Screen::handleMouseClick(DrawWindow *win)
            { inputTool -> handleMouseClick(win)}
void Screen::Screen(Tool *t) { inputTool = t; }
```

To solve the first problem, we define a pure virtual function in the CanvasTool class and a disambiguating method in each construction subclass:

```
void CanvasTool::handleMouseClick(DrawWindow *win) = 0;
void RectTool::handleMouseClick(DrawWindow *win)
              {Tool::handleMouseClick(win); }
void OvalTool::handleMouseClick(DrawWindow *win)
              {Tool::handleMouseClick(win); }
void SelectTool::handleMouseClick(DrawWindow *win)
                {Tool::handleMouseClick(win); }
```

To solve the second problem, we generate methods to transform the type of objects from Tool to CanvasTool and from CanvasTool to Tool. Wherever inputTool either occurs on the right hand side of an assignment, or is passed as a parameter to a function, or is returned from a function, it is first converted to its original type (Tool). Wherever inputTool occurs on the left hand side of an assignment statement, the expression on the right hand side is converted to its new type (CanvasTool).

```
Tool *CanvasTool::CT_to_T() = 0;
CanvasTool *Tool::T_to_CT() = 0;
Tool *RectTool::T_to_CT() { return this; }
CanvasTool *RectTool::CT_to_T()  { return this; }
Tool *OvalTool::T_to_CT() { return this; }
CanvasTool *OvalTool::CT_to_T()  { return this; }
Tool *SelectTool::T_to_CT() { return this; }
CanvasTool *SelectTool::CT_to_T()  { return this; }
void Screen::Screen(Tool *t) { inputTool = t -> T_to_CT(); }
```

4.2.5 Deletion of useless alternation classes

As in the case of the untyped language model, one problem with deleting a "useless" alternation class is that there may be methods attached to the class. There is the additional problem that the class name may be used in the static type declarations of objects.

If there are any methods attached to the useless alternation class, A, their implementations must be distributed to its immediate subclasses unless they are overridden there. If anywhere in the program an explicit call (i.e., through the scope resolution operator "::") to a method $A :: m$ is made, we create a new method with a unique name, say A_m, defined for each of A's immediate subclasses. The implementation for A_m is the same as for $A :: m$. Then, every occurrence of an explicit call to $A :: m$ is replaced with a call to A_m.

If there are any variables defined of static type A in the program, then we need to find an equivalent substitute type or else keep a class definition of A to be used

only to satisfy the type system. If there is an alternation class B with the same set of derived construction classes as for A, B can serve as a substitute type for A. In this case, all the member functions which were defined for A are now declared as pure virtual functions in class B, and class A is deleted. Wherever class A appeared in a type declaration, class B is substituted. Note that in conjunction with the part replacement transformation there is always such a corresponding class B.

If there is no such corresponding class B, then A can not be deleted since it must continue to be used in type declarations. In this case, class A is preserved, but contains only pure virtual functions. We regard A as a type rather than as a class.

As an example, consider what happens when the Tool class is deleted in the transformation from Figure 2–PRP to Figure 1–Object-equivalent. Suppose the methods declared in class Tool are these:

```
virtual void handleMouseClick( DrawWindow * ) = 0;
virtual Position getPosition() = 0;
virtual CanvasTool *T_to_CT() = 0;
```

All the methods happen to be pure virtual, so there are no implementations to be distributed. Furthermore, class CanvasTool qualifies as an equivalent substitute type for class Tool. For each method declared in class Tool a pure virtual method is declared in class CanvasTool. Everywhere that class Tool is used in a type declaration it is replaced with class CanvasTool. Finally class Tool can be deleted.

4.2.6 Class and part addition

As in the untyped language case, no changes are necessary for the method implementations.

4.2.7 Part generalization

The problem that occurs with part generalization is similar to one of the problems that occurs with part replacement. If the part class C of some part is generalized to a superclass of C, say B, then we must insure that for every method in class C there is a corresponding method defined in class B. This is done by defining empty virtual functions in B wherever necessary. Moreover, as for part replacement, wherever the part is involved in an assignment statement, function call (as a passed parameter), or function return (as the returned value), the part's new type will no longer be compatible. In this case, however, a simple cast will suffice since the new class is a superclass of the original.

Note that the PGN transformation indicates that a behavior extension is in order. Our goal in this work is simply to ensure behavior preservation. The above transformation achieves this goal, but the resulting code is not desirable from a software engineering point of view. The inserted cast operations are therefore

seen as a hint to the programmer as to where the behavior of the program should be extended.

4.3 Discussion

When comparing the update operations necessary in the two language models, the differences are striking. While in the untyped language model almost no updates to method implementations are necessary, the programmer working in the typed language model is faced with numerous problems. For the untyped language model, we have shown that a schema extension can always be propagated to the method implementations such that the behavior of the program is preserved. However, for the typed language model a behavior preserving update mechanism could only be outlined and is far from being satisfactory. The major reason for this is that the type system poses severe restrictions on how updates can be performed. Without semantic information on what the update's intentions are, it is not always possible to change the typing specifications in a reasonable way.

The above comparison underlines the popularity of untyped languages for prototyping purposes. Their ability to flexibly adapt themselves to different class structures gives them a major advantage over typed languages in environments where structural changes occur frequently. For typed languages, the propagation pattern approach [LXSL91, LHSLX92] achieves the same flexibility by decoupling the programs from the class structure. Consequently, any change in the class structure affects the propagation pattern only marginally.

5 Related Work

In the software refactory project at the University of Illinois, Opdyke and Johnson are investigating methods for *refactoring* object-oriented systems to support reuse [OJ90, Opd92]. Refactorings are defined as restructuring plans and are primarily used to aid the iterative design of an application framework. A feature of refactorings common to our approach is that they (1) also preserve behavior and (2) can be performed by applying a small set of basic refactorings.

Delcourt and Zicari designed a tool, called Integrity Consistency Checker (ICC), which ensures structural consistency while performing schema updates [DZ91]. The ICC guarantees that only those updates are performed that do not introduce any structural inconsistency. However, it allows behavioral inconsistencies that do not result in run-time errors. Nevertheless, the ICC is a very useful tool as a first component in the evolution process. In contrast, our tool guarantees behavioral consistency by automatically adapting programs to the new schema.

6 Conclusion

We have presented a code update mechanism for both CLOS and C++ that can be used to automate the propagation of changes from a modified schema to its affected programs. The mechanism guarantees not only that the schema and the programs stay structurally consistent, but also that the behavior of the old programs is consistently preserved after the schema update.

The ease with which CLOS programs can be adapted to extended schemas stands in striking contrast to the complexity involved in adapting C++ programs.

Acknowledgements: We would like to thank Karl Lieberherr, Ignacio Silva-Lepe and Cun Xiao for helpful discussions.

References

[AH88] Serge Abiteboul and Richard Hull. Restructuring hierarchical database objects. *Theoretical Computer Science*, 62:3–38, 1988.

[Bar91] Gilles Barbedette. Schema modifications in the *lispo*$_2$ persistent object-oriented language. In Pierre America, editor, *European Conference on Object-Oriented Programming*, pages 77–96, Geneva, Switzerland, July 1991. Springer Verlag, Lecture Notes in Computer Science.

[Ber91] Paul L. Bergstein. Object-preserving class transformations. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 299–313, Phoenix, Arizona, 1991. ACM Press.

[Ber92] Elisa Bertino. A view mechanism for object-oriented databases. In *International Conference on Extending Database Technology*, pages 136–151, Vienna, Austria, 1992.

[BKKK87] Jay Banerjee, Won Kim, Hyong-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 311–322. ACM, ACM Press, December 1987. SIGMOD Record, Vol.16, No.3.

[Cas91] Eduardo Casais. *Managing evolution in object-oriented environments: an algorithmic approach*. PhD thesis, University of Geneva, Geneva, Switzerland, May 1991. Thesis no. 369.

[CPLZ91] Alberto Coen-Porisini, Luigi Lavazza, and Roberto Zicari. Updating the schema of an object-oriented database. *Quarterly Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 14(2):33–37, June 1991. Special Issue on Foundations of object-Oriented Database Systems.

[DZ91] Christine Delcourt and Roberto Zicari. The design of an integrity consistency checker (icc) for an object oriented database system. In Pierre America, editor, *European Conference on Object-Oriented Programming*, pages 97–117,

Geneva, Switzerland, July 1991. Springer Verlag, Lecture Notes in Computer Science.

[LBSL91] Karl J. Lieberherr, Paul Bergstein, and Ignacio Silva-Lepe. From objects to classes: Algorithms for object-oriented design. *Journal of Software Engineering*, 6(4):205–228, July 1991.

[LH90] Barbara Staudt Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. In Norman Meyrowitz, editor, *Proceedings OOPSLA ECOOP '90*, pages 67–76, Ottawa, Canada, October 1990. ACM, ACM Press. Special Issue of SIGPLAN Notices, Vol.25, No.10.

[LHSLX92] Karl J. Lieberherr, Walter Hürsch, Ignacio Silva-Lepe, and Cun Xiao. Experience with a graph-based propagation pattern programming tool. In Gene Forte, Nazim H. Madhavji, and Hausi A. Müller, editors, *International Workshop on CASE*, pages 114–119, Montréal, Canada, July 1992. IEEE Computer Society Press.

[LHX93] Karl J. Lieberherr, Walter L. Hürsch, and Cun Xiao. Object-extending class transformations. *Formal Aspects of Computing, the International Journal of Formal Methods*, 1993. Accepted for publication.

[LXSL91] Karl J. Lieberherr, Cun Xiao, and Ignacio Silva-Lepe. Graph-based software engineering: Concise specifications of cooperative behavior. Technical Report NU-CCS-91-14, College of Computer Science, Northeastern University, Boston, MA, September 1991.

[OJ90] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of the Symposium on Object-Oriented Programming emphasizing Practical Applications (SOOPA)*, pages 145–160, Poughkeepsie, NY, September 1990. ACM.

[Opd92] William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing object-Oriented Application Frameworks*. PhD thesis, Computer Science Department, University of Illinois, May 1992.

[PS87] Jason D. Penney and Jacob Stein. Class modification in the GemStone object-oriented DBMS. In Norman Meyrowitz, editor, *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 111–117, Orlando, Florida, December 1987. ACM, ACM Press. Special Issue of SIGPLAN Notices, Vol.22, No.12.

[SZ86] Andrea H. Skarra and Stanley B. Zdonik. The management of changing types in an object-oriented database. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 483–495. ACM, ACM Press, September 1986.