SIGPLAN Notices Vol. 26, No. 11, pages 299-313, Phoenix, AZ, November 1991. ACM Press.

Object-Preserving Class Transformations

Paul L. Bergstein
Northeastern University, College of Computer Science
Cullinane Hall, 360 Huntington Ave., Boston MA 02115
pberg@corwin.CCS.northeastern.EDU

Abstract

Reorganization of classes for object-oriented programming and object-oriented database design has recently received considerable attention in the literature. In this paper a small set of primitive transformations is presented which forms an orthogonal basis for object-preserving class reorganizations. This set is proven to be correct, complete, and minimal. The primitive transformations help form a theoretical basis for class organization and are a powerful tool for reasoning about particular organizations.

Keywords: Object-oriented programming and design, object-oriented database design, class library organization.

1 Introduction

Reorganization of classes for object-oriented programming and object-oriented database design has recently received considerable attention in the literature: [BCG+87], [LBSL90], [LBSL91], [AH87], [BMW86], [Cas89], [Cas90], [LM91], [Pir89], [PW89]. A number of researchers have suggested algorithms and hueristics to produce "good" class organizations. A "good" class organization may be variously defined as one which promotes efficient reuse of code, one with a minimum of multiple-inheritance, a minimum of repeated-inheritance, or some other characteristics depending on the author's point of view.

In any case, it is usually desirable that reorganization of a class hierarchy should not change the set of objects which the classes define, that is the reorganization should be **object-preserving**. For object-oriented database design, this means that the database does not need to be repopulated. For object-oriented programming, this means that programs will still accept the same inputs and produce the same outputs. Furthermore, methods need not be rewritten (although they may need to be attached to different classes).

In this paper a small set of primitive transformations is presented which forms an orthogonal basis for object-preserving class organizations. This set is proven to be correct, complete, and minimal. The primitive transformations help form a theoretical basis for class organization and are useful in proving characteristics of particular organizations.

The concept of a primitive set of object-preserving class transformations was developed as part of the Demeter project to develop CASE tools for object-oriented design and programming. While the class model used in this paper is the simplified one of [LBSL91], the Demeter System^{TM} actually uses an expanded model which includes optional parts, collection (repetition) classes, and the ability to specify concrete syntax used for parsing and printing objects. Each notation has the advantage of being programming language independent and is therefore useful to programmers who use object-oriented languages such as C++ [Str86], Smalltalk [GR83], CLOS [BDG+88] or Eiffel [Mey88].

The C++ Demeter System incorporates a C++

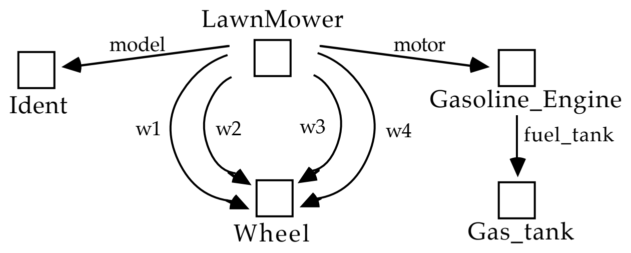


Figure 1: Construction class

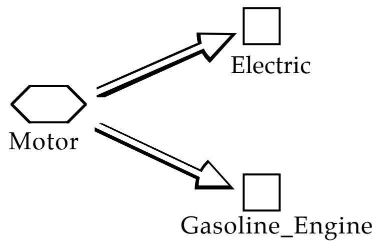


Figure 2: Alternation class

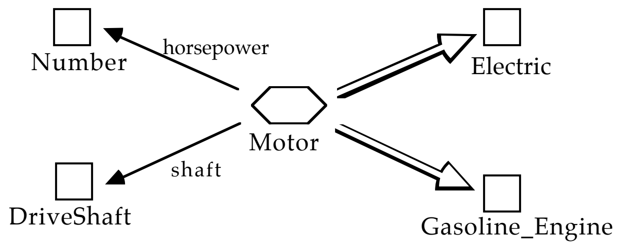


Figure 3: Common parts

code generation algorithm to translate the class definitions into C++ and generate methods for manipulating the application objects (e.g. parsing, printing, copying, comparing, traversing, etc.). The primitive transformations discussed in this paper were very helpful in developing and analyzing the latest additions to the Demeter System: tools for the abstraction of optimal class organizations from object examples, and for the optimization of existing class organizations [LBSL91] [BL91].

The second section provides a brief description of the class notation. In section 3 the primitive transformations are presented along with related proofs. In section 4 some practical rules for class hierarchy optimization, which can be built from the primitive transformations, are given.

2 Class notation

The class notation of [LBSL91] uses two kinds of classes: construction and alternation classes. A construction class definition is an abstraction of a class definition in a typical statically typed programming language (e.g., C++). A construction class does not reveal implementation information. Examples of construction classes are in Fig. 1 for: LawnMower, Wheel, etc.

Each construction class defines a set of objects which can be thought of as being elements of the direct product of the part classes. When modeling an application domain, it is natural to take the union of object sets defined by construction classes. For example, the motor of a lawn-mower can be either a gasoline engine or an electric motor. So the objects that can be stored in the motor part of a lawn-mower are either gasoline_engine or electric motor objects. Alternation classes are used to define such union classes. An example of an alternation class is in Fig. 2. `Gasoline_Engine` and `Electric` are called alternatives of the alternation class. Often the alternatives have some common parts. For example, each motor has a drive shaft. The notation in Fig. 3 is used to express such common parts. Alternation classes have their origin in the variant records of Pascal. Because of the delayed binding of function calls to code in object-oriented programming, alternation classes are easier to use than variant records.

Alternation classes which have common parts are implemented by inheritance. In Fig. 3, `Electric` and `Gasoline_Engine` inherit from `Motor`. Class `Motor` has methods and/or instance variables to implement the parts `horsepower` and `shaft`.

Construction and alternation classes correspond to the two basic data type constructions in denotational semantics: cartesian products and disjoint sums. They also correspond to the two basic mechanisms used in formal languages: concatenation and alternation.

The concept of a part class which is used throughout this paper needs further explanation. A part object does not have to be a physical part;

any attribute of an object is a part of it. Object o_2 is said to be a part of object o_1 if "o_1 knows about o_2". Therefore, the part-of relation is a generalization of the aggregation relation which only describes physical containment. For example, a car is part of a wheel if the wheel knows about the car.

Definition 1 . *A* **class dictionary graph** ϕ *is a directed graph* $\phi = (V, \Lambda;\ EC, EA)$ *with finitely many vertices* V*.* Λ *is a finite set of labels. There are two defining relations* EC, EA*.* EC *is a ternary relation on* $V \times V \times \Lambda$*, called the (labeled) construction edges:* $(v, w, l) \in EC$ *iff there is a construction edge with label* l *from* v *to* w*.* EA *is a binary relation on* $V \times V$*, called the alternation edges:* $(v, w) \in EA$ *iff there is an alternation edge from* v *to* w*.*

Next the set of vertices is partitioned into two subclasses, called the construction and alternation vertices.

Definition 2 .

- *The* **construction vertices** *are defined by* $VC = \{v \mid v \in V, \forall w \in V : (v, w) \notin EA\}$.
  In other words, the construction vertices have no outgoing alternation edges.
- *The* **alternation vertices** *are defined by* $VA = \{v \mid v \in V, \exists w \in V : (v, w) \in EA\}$.
  In other words, the alternation vertices have at least one outgoing alternation edge.

Sometimes, it is more convenient to describe a class dictionary graph as a tuple which contains explicit references to VC and VA: $\phi = (VC, VA, \Lambda; EC, EA)$.

The definition of a class dictionary graph is motivated by the interpretation in object-oriented design given in Figure 4. During the programming process, the alternation classes serve to define interfaces (i.e., they serve the role of types) and the construction classes serve to provide implementations for the interfaces.

The standard interpretation implies that the labels on construction vertices are significant. Consider two class dictionary graphs each with only a single construction vertex and no edges. From a graph theoretic point of view, the graphs are equal regardless of the labels on the vertices, but if the construction vertex of one graph is labeled *Integer* and the vertex of the other graph is labeled *String*, then the two class dictionary graphs define different sets of objects in the standard interpretation.

Graph	Object-oriented design
Vertex	Class
construction	instantiable
alternation	abstract
Edge	Class relationship
construction	part-of relationship "uses", "knows", labels are part names
alternation	inheritance relationship specialization, classification

Figure 4: Standard Interpretation

Since the mapping from construction vertices to labels is a bijection, its explicit inclusion in the definition of class dictionary graphs would only clutter the theory. When referring to an element of the construction vertices of a class dictionary graph, the reference is sometimes to a vertex and sometimes to the label of a vertex. The meaning should be clear from the context.

The following graphical notation, based on [TYF86], is used for drawing class dictionary graphs: squares for construction vertices, hexagons for alternation vertices, thin lines for construction edges and double lines for alternation edges.

Example 1 *Fig. 5 shows a class dictionary graph for telephones. Telephones can either be standard or cordless and they also can be either rotary dial or touch-tone. Cordless phones have an antenna while standard phones have a handset cord. The dialer on a touch-tone phone is a keypad, whereas a rotary dial phone has a dial. For further illustration the components of the formal definition are given, i.e.:*

```
V = { Telephone, Cordless, Standard,
      Antenna, Handset_Cord,
      Dialer, Rotary, Dial,
      TouchTone, Keypad }

VC = { Cordless, Standard, Antenna,
       Handset_Cord, Rotary, Dial,
       TouchTone, Keypad }

VA = { Telephone, Dialer }

EC = { (Telephone, Dialer, dialer),
       (Rotary, Dial, dial),
       (TouchTone, Keypad, dial),
       (Cordless, Antenna, ant),
       (Standard, Handset_Cord, cord) }

EA = { (Telephone, Cordless),
       (Telephone, Standard),
       (Dialer, Rotary),
       (Dialer, TouchTone) }
```

Λ = {dialer, ant, cord, dial }.

Definition 3 *In a class dictionary graph* $\phi = (V, \Lambda;\ EC, EA)$*, a vertex* $w \in V$ *is* **alternation-reachable** *from vertex* $v \in V$ *(we write* $v \overset{*}{\Rightarrow} w$*):*

- *via a path of length 0, if* $v = w$
- *via a path of length* $n+1$*, if* $\exists u \in V$ *such that* $(v,u) \in EA$ *and* $u \overset{*}{\Rightarrow} w$ *via a path of length* n*.*

A legal class dictionary graph is a structure which satisfies 2 independent axioms.

Definition 4 *A class dictionary graph* $\phi = (V, \Lambda;\ EC, EA)$ *is* **legal** *if it satisfies the following two axioms:*

1. *Cycle-free alternation axiom:*

   There are no cyclic alternation paths, i.e., $\{(v,w) \mid v,w \in V, v \neq w, \text{ and } v \overset{*}{\Rightarrow} w \overset{*}{\Rightarrow} v\} = \emptyset$*.*

2. *Unique labels axiom:*

   $\forall u, v, v', w, w' \in V,\ l \in \Lambda$ *such that*
   $v \overset{*}{\Rightarrow} u,\ v' \overset{*}{\Rightarrow} u,$ *and* $(v,w) \neq (v',w')$ *:*
   $\{(v,w,l),\ (v',w',l)\} \not\subseteq EC$

The cycle-free alternation axiom is natural and has been proposed by other researchers, e.g., [PBF+89, page 396], [Sno89, page 109: Class names may not depend on themselves in a circular fashion involving only (alternation) class productions]. The axiom says that a class may not inherit from itself.

The unique labels axiom guarantees that "inherited" construction edges are uniquely labeled. Other mechanisms for uniquely naming the construction edges could be used, e.g., the renaming mechanism of Eiffel [Mey88].

Throughout the rest of this paper, the term class dictionary graph refers to a legal class dictionary graph.

3 Primitive Object-Preserving Transformations

An informal definition of **object-preserving** has already been given in the introduction. For a formal definition we first need a definition of **object-equivalence.** [1]

Definition 5 *Given a class dictionary graph* $\phi = (VC, VA, \Lambda;\ EC, EA)$*, for* $v \in V$ *let*

$$PartClusters_\phi(v) = \{(l, \mathcal{A}(w)) \mid \exists v' : v' \overset{*}{\Rightarrow} v \text{ and } (v', w, l) \in EC\}$$

where $\mathcal{A}(w) = \{w' \mid w \overset{*}{\Rightarrow} w' \text{ and } w' \in VC\}$*.*

Then, class dictionary graphs ϕ_1 *and* ϕ_2 *are* **object-equivalent** *if:*

- $VC_{\phi_1} = VC_{\phi_2}$
- $\forall v \in VC :$
  $PartClusters_{\phi_1}(v) = PartClusters_{\phi_2}(v)$*.*

Intuitively, two class dictionary graphs are object-equivalent if they define sets of corresponding construction classes with the same names, and for each construction class defined by one class dictionary graph the parts are the same as those defined for the corresponding class in the other class dictionary graph.

[1]The most straight-forward definition would be: Two class dictionary graphs, ϕ_1 and ϕ_2, are object-equivalent if $Objects(\phi_1) = Objects(\phi_2)$, where $Objects(\phi)$ is formally defined in [LBSL91]. The equivalent definition of object-equivalence given here is more appropriate for this paper.

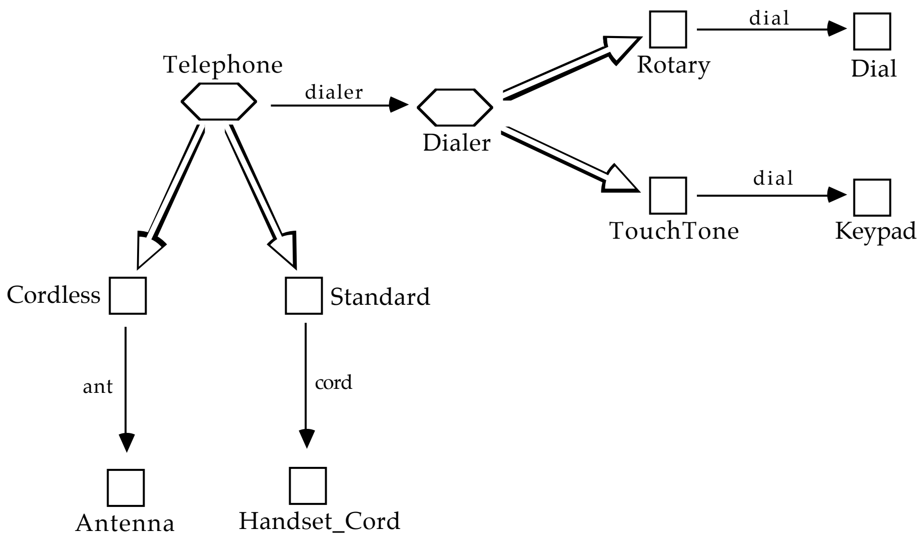


Figure 5: Telephones

Example 2 *The two class dictionary graphs in Fig. 6, ϕ_1 and ϕ_2, are object-equivalent since:*

$$VC_{\phi_1} = VC_{\phi_2} = \{\texttt{Undergrad, Grad, Prof, TA, Admin\_asst, Coach, Num, Real\_Num}\}$$

$$PartClusters_{\phi_1}(\texttt{Undergrad}) = PartClusters_{\phi_2}(\texttt{Undergrad}) = \{(\texttt{ssn}, \{\texttt{Num}\}), (\texttt{gpa}, \{\texttt{Real\_Num}\})\}$$

$$PartClusters_{\phi_1}(\texttt{Grad}) = PartClusters_{\phi_2}(\texttt{Grad}) = \{(\texttt{ssn}, \{\texttt{Num}\}), (\texttt{gpa}, \{\texttt{Real\_Num}\})\}$$

$$PartClusters_{\phi_1}(\texttt{TA}) = PartClusters_{\phi_2}(\texttt{TA}) = \{(\texttt{ssn}, \{\texttt{Num}\}), (\texttt{salary}, \{\texttt{Real\_Num}\}), (\texttt{assigned}, \{\texttt{Course, Committee}\})\}$$

$$PartClusters_{\phi_1}(\texttt{Prof}) = PartClusters_{\phi_2}(\texttt{Prof}) = \{(\texttt{ssn}, \{\texttt{Num}\}), (\texttt{salary}, \{\texttt{Real\_Num}\}), (\texttt{assigned}, \{\texttt{Course, Committee}\})\}$$

$$PartClusters_{\phi_1}(\texttt{Admin\_asst}) = PartClusters_{\phi_2}(\texttt{Admin\_asst}) = \{(\texttt{ssn}, \{\texttt{Num}\}), (\texttt{salary}, \{\texttt{Real\_Num}\})\}$$

$$PartClusters_{\phi_1}(\texttt{Coach}) = PartClusters_{\phi_2}(\texttt{Coach}) = \{(\texttt{ssn}, \{\texttt{Num}\}), (\texttt{salary}, \{\texttt{Real\_Num}\})\}$$

$$PartClusters_{\phi_1}(\texttt{Course}) = PartClusters_{\phi_2}(\texttt{Course}) = \emptyset$$

$$PartClusters_{\phi_1}(\texttt{Committee}) = PartClusters_{\phi_2}(\texttt{Committee}) = \emptyset$$

$$PartClusters_{\phi_1}(\texttt{Real\_Num}) = PartClusters_{\phi_2}(\texttt{Real\_Num}) = \emptyset$$

$$PartClusters_{\phi_1}(\texttt{Num}) = PartClusters_{\phi_2}(\texttt{Num}) = \emptyset$$

Definition 6 *A class dictionary graph* **transformation**, *T, is a rule which defines an allowable modification of class dictionary graphs. Let*

$$R_T = \{(\phi_1, \phi_2) \mid \phi_2 \textit{ can be obtained from } \phi_1 \textit{ by a single application of } T\}.$$

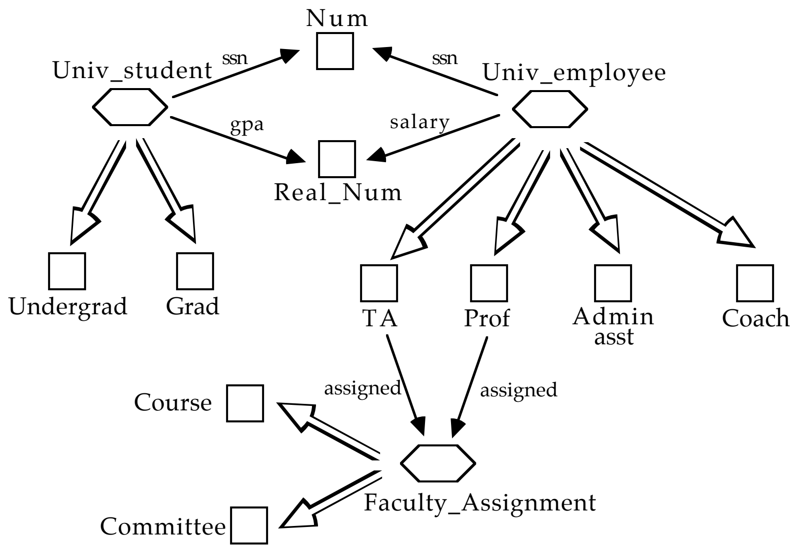
Then T is called **object-preserving** *if ϕ_1 is object-equivalent to ϕ_2 for all $(\phi_1, \phi_2) \in R_T$.*

3.1 Primitive Transformations

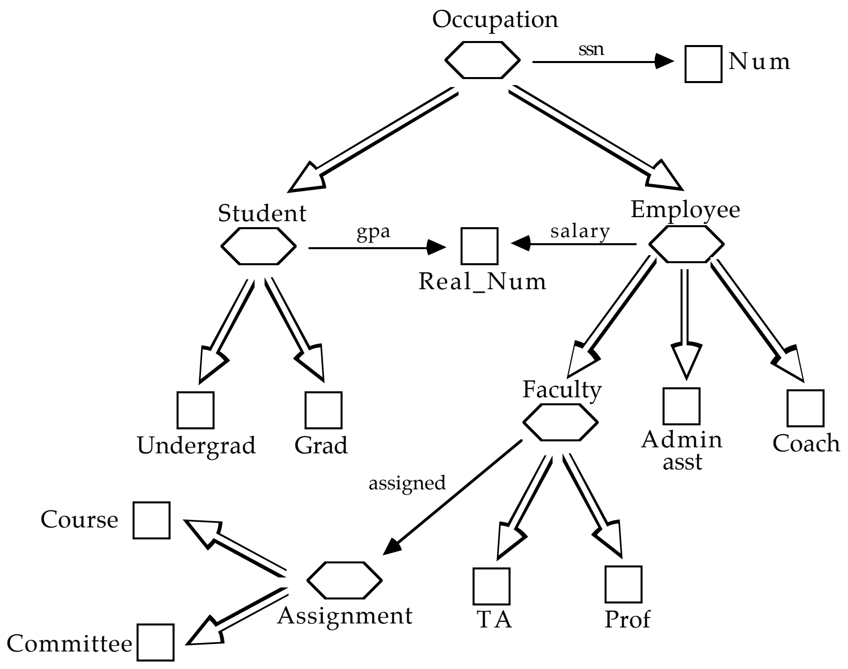
The following five primitive transformations form an orthogonal basis for object-preserving transformations:

1. **Deletion of "useless" alternation**. An alternation vertex is "useless" if it has no incoming edges and no outgoing construction edges. If an alternation vertex is useless it may be deleted along with it's outgoing alternation edges.

   Intuitively, an alternation vertex is useless if it is not a part of any construction class, and



(a) Class Dictionary Graph ϕ_1



(b) Class Dictionary Graph ϕ_2

Figure 6: Object-equivalent Class Dictionary Graphs

it has no parts for any construction class to inherit.

2. **Addition of "useless" alternation.** An alternation vertex, v, can be added along with outgoing alternation edges to any set of vertices already in the class dictionary graph. This is the inverse of transformation 1.

3. **Abstraction of common parts.** If $\exists v, w, l$ such that $\forall v'$, where $(v, v') \in EA : (v', w, l) \in EC$, then all of the edges, (v', w, l), can be deleted and replaced with a new construction edge, (v, w, l).

   Intuitively, if all of the immediate subclasses of class C have the same part, that part can be moved up the inheritance hierarchy so that each of the subclasses will inherit the part from C, rather than duplicating the part in each subclass.

4. **Distribution of common parts.** An outgoing construction edge, (v, w, l) can be deleted from an alternation vertex, v, if for each $(v, v') \in EA$ a new construction edge, (v', w, l) is added.

   This is the inverse of transformation 3.

5. **Part replacement.** If the set of construction vertices which are alternation-reachable from some vertex, $v \in V$, is equal to the set of construction vertices alternation-reachable from another vertex, $v' \in V$, then any construction edge $(w, v, l) \in EC$ can be deleted and replaced with a new construction edge, (w, v', l).

   Intuitively, if two class C1 and C2 have the same set of instantiable (construction) subclasses then the defined objects do not change when C1 is replaced by C2 in a part definition.

The set of primitive object-preserving transformation given in this section is *correct*, i.e. any sequence of primitive transformations preserves object-equivalence; *complete*, i.e. for any two object-equivalent class dictionary graphs, ϕ_1, ϕ_2, there is a sequence of primitive operations which transforms ϕ_1 to ϕ_2; and *minimal*, i.e. none of the primitive transformations can be derived from any set of the others.

3.2 Proofs

3.2.1 Correctness

Each primitive operation preserves object-equivalence.

3.2.2 Completeness

Given two object-equivalent class dictionary graphs, ϕ_1 and ϕ_2, ϕ_1 can be transformed to ϕ_2 using only primitive operations as follows:

1. Use primitive operation 2 (addition of useless alternation) to "superimpose" the alternation subgraph of ϕ_2 onto ϕ_1.

   Since there are no alternation cycles in ϕ_2, there must be some $v \in VA_{\phi_2}$ with outgoing alternation edges only to construction vertices (if there are any alternation vertices at all). For each such alternation vertex, add a new alternation vertex to ϕ_1 with alternation edges to the corresponding construction vertices.

   Now continue adding new alternation vertices corresponding to alternation vertices in ϕ_2 that have outgoing alternation edges only to construction vertices and alternation vertices which have already been added in ϕ_1 until all the alternation vertices in ϕ_2 are duplicated in ϕ_1.

2. Use primitive operation 4 (distribution of common parts) to remove the outgoing construction edges from all of the original alternation vertices in ϕ_1.

   Distribution of common parts is applied repeatedly until all of the parts are attached directly to construction vertices.

3. Use primitive operation 3 (abstraction of common parts) to move construction edges up the "new" inheritance hierarchy in ϕ_1 until they are all attached to vertices corresponding to

the vertices where they are attached in ϕ_2. This must be possible since ϕ_1 and ϕ_2 are object-equivalent.

At this point ϕ_1 and ϕ_2 have the same number of construction edges and the construction edges have the same labels and the same sources, but may have different targets.

4. Use primitive number 5 (part replacement) to move any construction edge with an "old" alternation vertex or construction vertex as its target so that its target corresponds to the proper vertex in ϕ_2.

5. Use primitive transformation 1 (deletion of useless alternation) to delete the "old" alternation subgraph from ϕ_1. At this point there are no construction edges (either incoming or outgoing) attached to any of the "old" alternation vertices. Also, since there are no cycles in the old alternation subgraph, and since we have not added any edges from "new" alternation vertices to "old" alternation vertices or vice versa, at least one of the "old" alternation vertices must be "useless" (if there are any at all). After deleting that useless alternation vertex the condition still holds, so we can continue deleting the "old" alternation vertices until there are none left.

   Now $\phi_1 = \phi_2$.

3.2.3 Minimality

No primitive transformation can be derived from any set of the others since:

- No sequence of primitive operations can reduce the number of alternation vertices without deletion of useless alternations.
- No sequence of primitive operations can increase the number of alternation vertices without addition of useless alternations.
- No sequence of primitive operations can reduce the number of construction edges without abstraction of common parts.
- No sequence of primitive operations can increase the number of construction edges without distribution of common parts.
- No sequence of primitive operations can change the construction edge in-degree of a vertex from 0 to 1 or from 1 to 0 without part replacement.

Example 3 *This example illustrates the construction of the completeness proof with the class dictionary graphs of Figure 6. Note that although the labels on construction vertices are significant, the labels on the alternation vertices are only provided as a means of referring to particular vertices in the following discussion.*

Addition of Useless Alternations. In ϕ_2 there are three alternation vertices which have outgoing alternation edges only to construction vertices: `Faculty, Assignment, and Student`. These are added to ϕ_1 along with their outgoing alternation edges. Next, the `Employee` vertex is added with its outgoing alternation edges, including an edge to `Faculty`. Finally, the `Occupation` vertex is added along with its edges to `Student` and `Employee`. At this point ϕ_1 has been transformed to the class dictionary graph shown in Figure 7.

Distribution of Common Parts. The `ssn` and `gpa` parts are distributed from class `Univ_student` to classes `Undergrad` and `Grad` where they are inherited. Similarly, parts `ssn` and `salary` are distributed from `Univ_employee to TA, Prof, Admin_asst,` and `Coach`. The result is the class dictionary graph shown in Figure 8. In a deeper inheritance hierarchy some parts might need to be distributed repeatedly until they are attached directly to construction classes.

Abstraction of Common Parts. Parts `ssn` and `gpa` are abstracted from `Undergrad` and `Grad` to `Student`. Next, parts `ssn, salary,` and `assigned` are abstracted from `TA` and `Prof` to `Faculty`. Parts `ssn` and `salary` are then abstracted from `Faculty, Admin_asst,` and `Coach` to `Employee`. Finally, part `ssn` is abstracted from `Employee` and `Student` to `Occupation`. The result is shown in Figure 9.

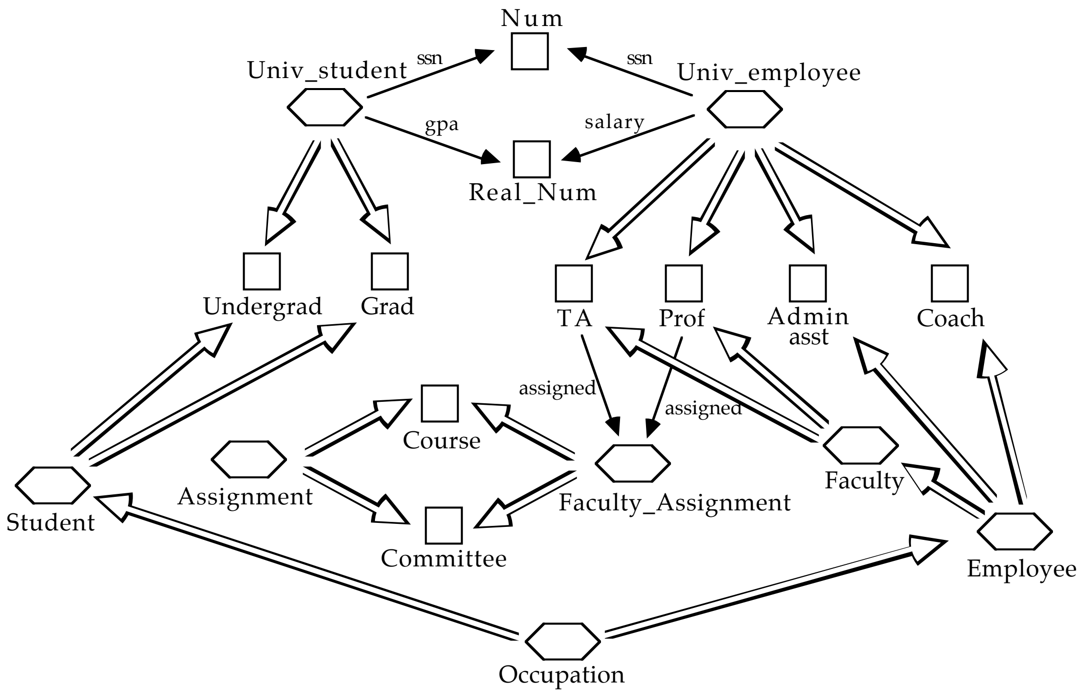


Figure 7: Addition of Useless Alternations

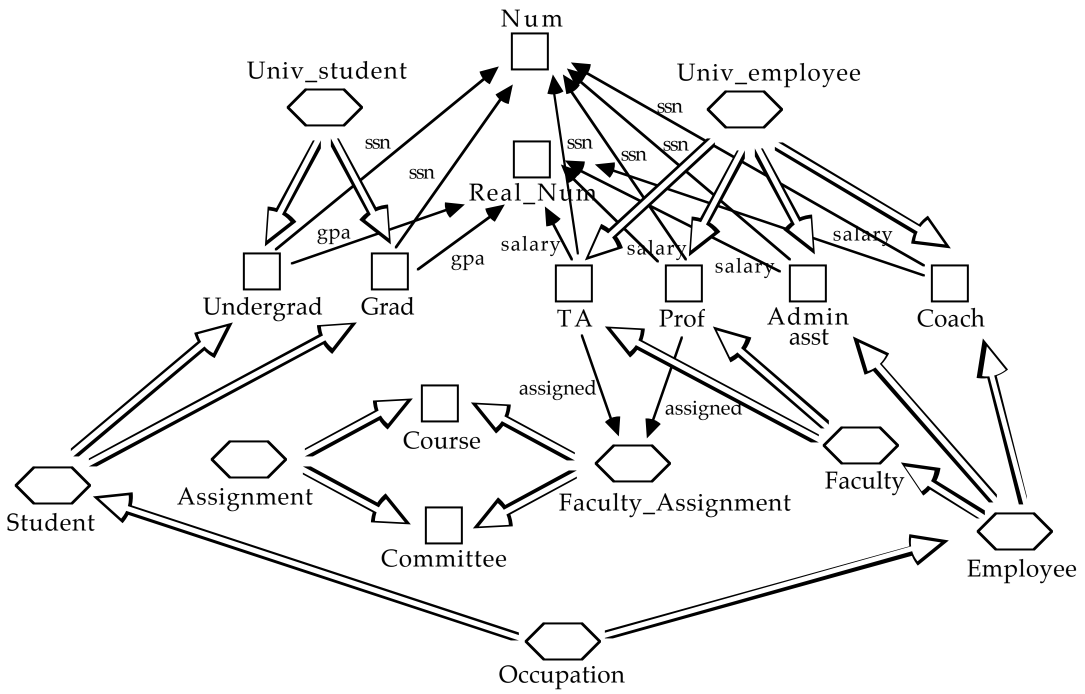


Figure 8: Distribution of Common Parts

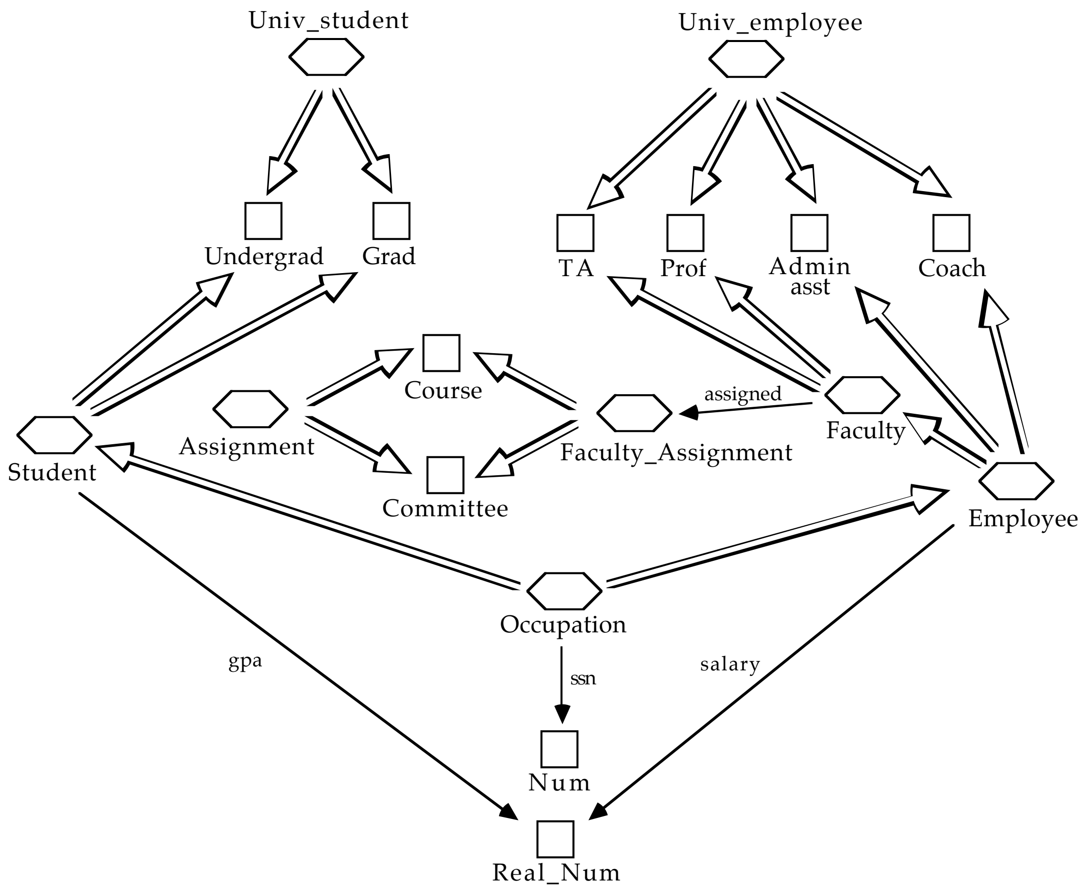


Figure 9: Abstraction of Common Parts

Part Replacement. The "old" alternation vertex `Faculty_Assignment` still has an incoming construction edge from the new vertex `Faculty`. In ϕ_2 the corresponding edge is to vertex `Assignment`, so the edge is moved accordingly in ϕ_1. This is allowed since the set of construction vertices alternation reachable from `Assignment` is equal to the set alternation reachable from `Faculty_Assignment`. Such a part replacement must always be possible since ϕ_1 is object-equivalent to ϕ_2. The result is shown in Figure 10.

Deletion of Useless Alternations. The alternation vertices `Faculty_Assignment`, `Univ_student`, and `Univ_employee` are now "useless" since they have no incoming edges and no outgoing construction edges. These vertices and their outgoing alternation edges are deleted, and the transformation from ϕ_1 to ϕ_2 is complete.

4 Practical Applications

There are many useful rules which can be derived from the primitive transformations and are therefore guaranteed object-preserving. The following examples show how object-preserving transformations can be used to improve class organization by reducing the number of construction edges, the number of alternation edges, or the degree of multiple inheritance in a class dictionary graph.

1. **Elimination of redundant parts.**
   If a vertex, v, has two incoming construction edges with the same label, (u, v, l) and (u', v, l), then those edges should be replaced by a single edge (w, v, l) where w is an alternation vertex with exactly u and u' as alternation successors, by *abstraction of common parts.* If necessary, w is first introduced by *addition of useless alternation.* (See Fig. 11.)

2. **Removal of singleton alternation vertices.**
   If an alternation vertex, v, has only one outgoing alternation edge, (v, w), then that vertex should be removed. Incoming construction edges (u, v, l), and alternation edges, (u, v), are replaced by edges (u, w, l) and (u, w) respectively. Outgoing construction edges, (v, x, l), are replaced by edges (w, x, l). The incoming construction edges can be moved by *part replacement* and the outgoing construction edges by *distribution of common parts.* Moving the incoming alternation edges can be accomplished by *alternation replacement* which is analogous to *part replacement* but is not primitive. It is easy to see how *alternation replacement* can be accomplished using only primitive transformations. Finally, the vertex v is deleted by *deletion of useless alternation.* (See Fig. 12.)

3. **Complete Cover**
   If a subset, S, of the outgoing alternation edges from a vertex, u, completely cover the alternatives of another alternation vertex, v, then replace the edges in S with a single alternation edge to v. We say the alternatives of an alternation vertex, v, are completely covered by a set of edges, S, if every vertex which is the target of an outgoing alternation edge from v is also the target of an edge in S. This rule can be derived from the primitive transformations using a construction similar to that given in section 3.2.2. (See Fig. 13.)

4. **Partial Cover**
   This rule applies if two alternation vertices, u and v, cover a common set of alternatives, but neither contains a subset of outgoing alternation edges that completely covers the alternatives of the other. In this case, a new alternation vertex, w, is created with an outgoing alternation edge to each of the vertices that is a target of outgoing alternation edges from both u and v, and incoming alternation edges (u, w) and (v, w). For each edge (w, x) which is added, the corresponding edges (u, x) and (v, x) are deleted. (See Fig. 14.)

5. **MI Minimization**
   If there are alternation edges, (u, w) and (v, w) such that for all other alternation edges from v, (v, w'), w' is alternation reachable from u,

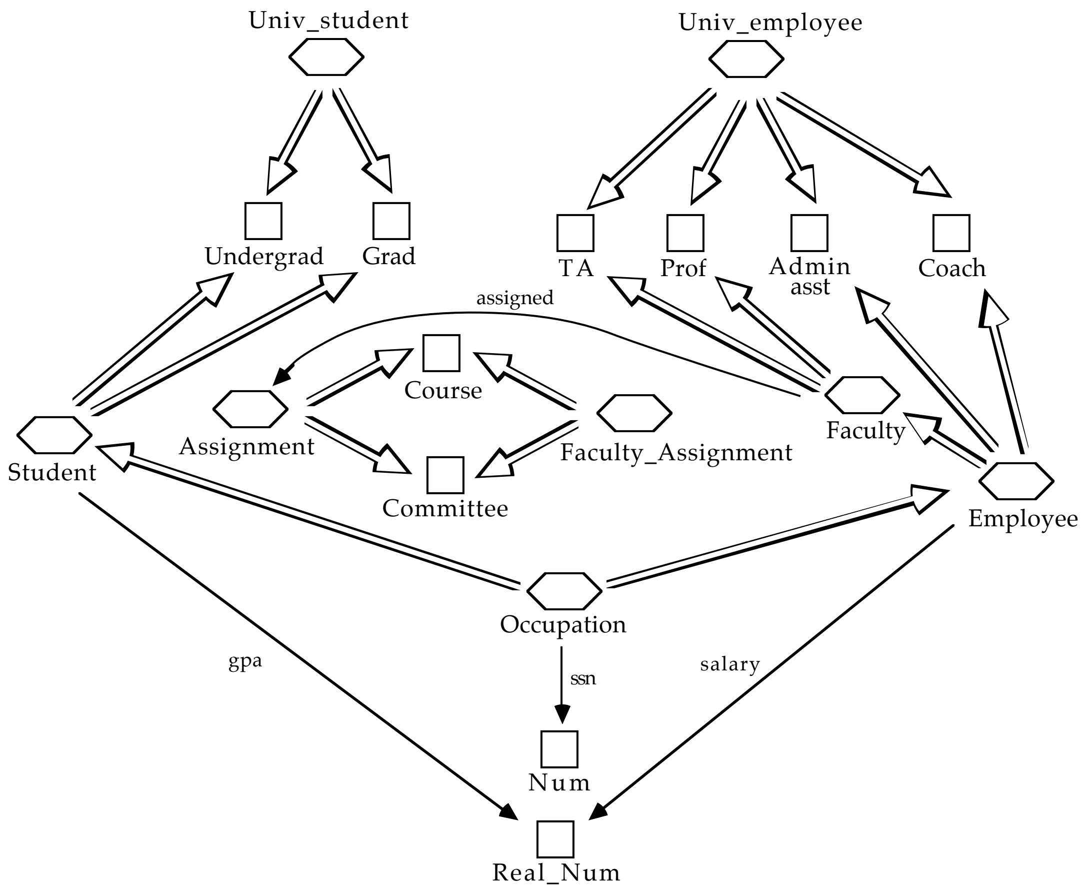


Figure 10: Part Replacement

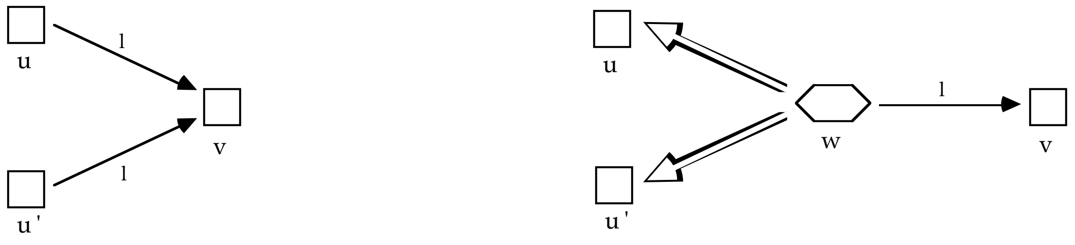


Figure 11: Elimination of redundant parts

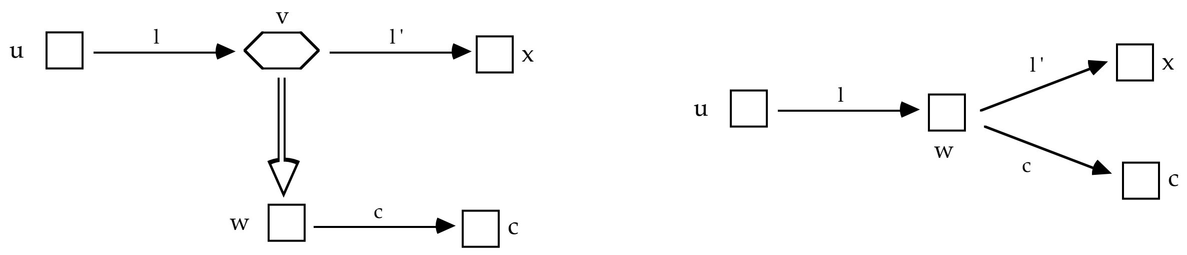


Figure 12: Removal of singleton alternation vertex

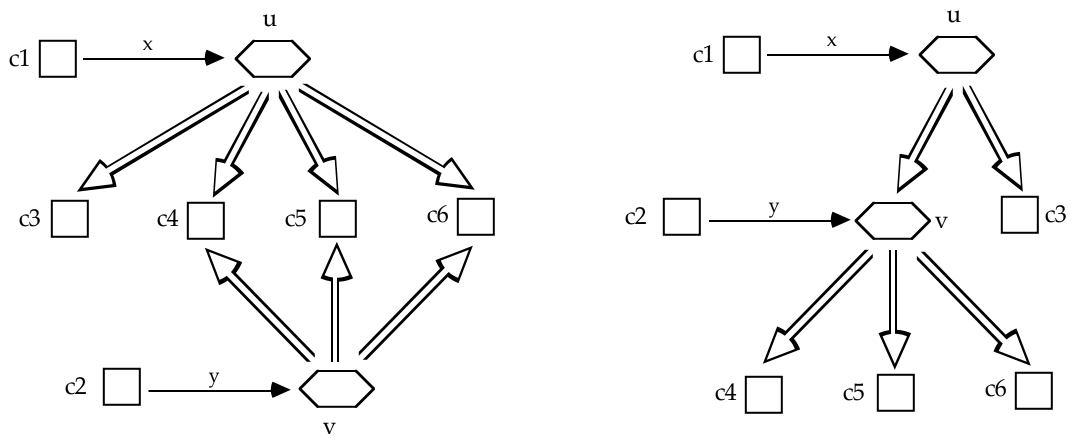


Figure 13: Complete Cover

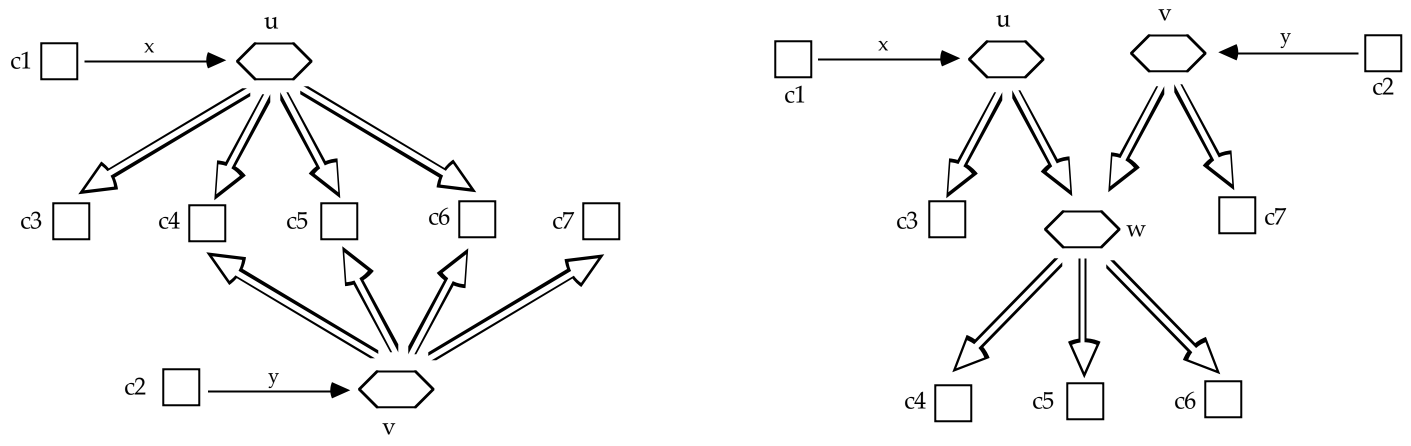


Figure 14: Partial cover

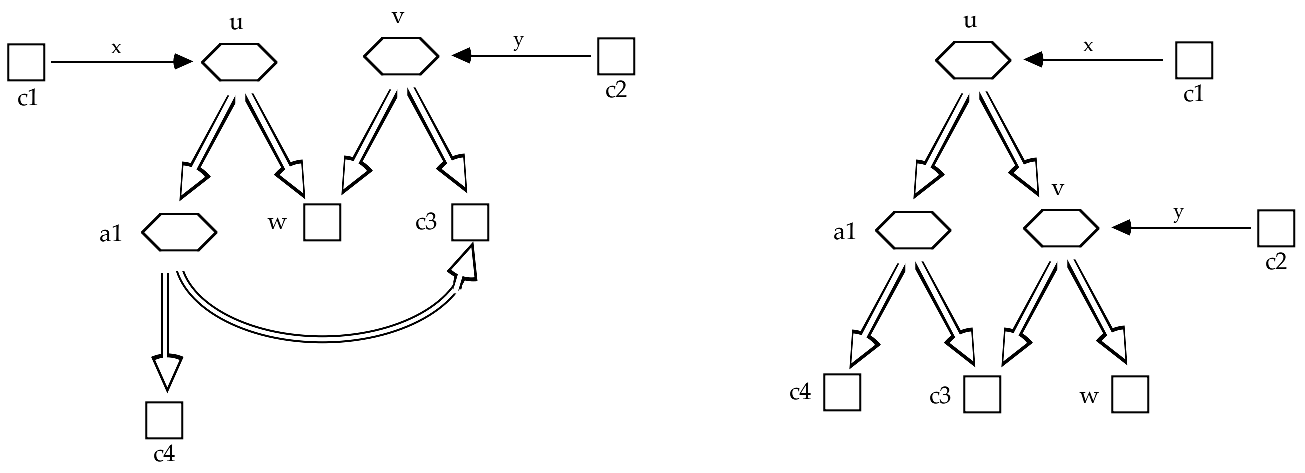


Figure 15: MI minimization

then replace the edge (u, w) with the edge (u, v). This rule reduces the amount of multiple inheritance without changing the edge size. However, it introduces repeated inheritance. (See Fig. 15.)

5 Conclusion

The primitive object-preserving class transformations presented in this paper are a powerful tool for reasoning about object-preserving transformations and optimizations. In order to determine whether a transformation is guaranteed to be object-preserving it is only necessary to show whether it can be derived from the primitive transformations.

To prove that a particular class organization is in some sense *optimal* (see, for example, [LBSL91]), it is only necessary to consider improvements that might be possible through the primitive transformations.

An area for further research is the study of *object-extending* class reorganizations [LHX91]. An object-extending transformation is one which adds to the set of defined objects or adds part classes to previously defined objects. For object-oriented data base design this means that the objects can be updated automatically. For object-oriented programming it means that the programs will still accept similar inputs and produce similar outputs.

Acknowledgments: I would like to thank Karl Lieberherr for his generous support and feedback. Additional thanks go to Cun Xiao for his help in polishing some of the definitions.

References

[AH87] S. Abiteboul and R. Hull. A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, Dec. 1987.

[BCG+87] Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, Won Kim, Darrell Woelk, and Nat Ballou. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems*, 5(1):3 – 26, January, 1987.

[BDG+88] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, and D.A. Moon. Common Lisp Object System Specification. *SIGPLAN Notices*, 23, September 1988.

[BL91] Paul Bergstein and Karl Lieberherr. Incremental class dictionary learning and optimization. In *European Conference on Object-Oriented Programming*, pages 377–396, Geneva, Switzerland, 1991. Springer Verlag.

[BMW86] Alexander Borgida, Tom Mitchell, and Keith Williamson. Learning improved integrity constraints and schemas from exceptions in data and knowledge bases. In Michael L. Brodie and John Mylopoulos, editors, *On Knowledge Base Management Systems*, pages 259–286. Springer Verlag, 1986.

[Cas89] Eduardo Casais. Reorganizing an object system. In Dennis Tsichritzis, editor, *Object Oriented Development*, pages 161–189. Centre Universitaire D'Informatique, Genève, 1989.

[Cas90] Eduardo Casais. Managing class evolution in object-oriented systems. In Dennis Tsichritzis, editor, *Object Management*, pages 133–195. Centre Universitaire D'Informatique, Genève, 1990.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison Wesley, 1983.

[LBSL90] Karl J. Lieberherr, Paul Bergstein, and Ignacio Silva-Lepe. Abstraction of object-oriented data models. In Hannu Kangassalo, editor, *Proceedings of International Conference on Entity-Relationship*, pages 81–94, Lausanne, Switzerland, 1990. Elsevier.

[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.

[LHX91] Karl J. Lieberherr, Walter L. Hürsch, and Cun Xiao. Object-extending class transformations. Technical Report NU-CCS-91-8, Northeastern University, July 1991.

[LM91] Qing Li and Dennis McLeod. Conceptual database evolution through learning. In Rajiv Gupta and Ellis Horowitz, editors, *Object-oriented Databases with applications to CASE, networks and VLSI CAD*, pages 62–74. Prentice Hall Series in Data and Knowledge Base Systems, 1991.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Series in Computer Science. Prentice Hall International, 1988.

[PBF+89] B. Pernici, F. Barbic, M.G. Fugini, R. Maiocchi, J.R. Rames, and C. Rolland. C-TODOS: An automatic tool for office system conceptual design. *ACM Transactions on Office Information Systems*, 7(4):378–419, October 1989.

[Pir89] Fiora Pirri. Modelling a multiple inheritance lattice with exceptions. In *Proceedings of the Workshop on Inheritance and Hierarchies in Knowledge Representation and Programming Languages*, pages 91–104, Viareggio, February 1989.

[PW89] Winnie W. Y. Pun and Russel L. Winder. Automating class hierarchy graph construction. Technical report, University College London, 1989.

[Sno89] Richard Snodgrass. *The interface description language.* Computer Science Press, 1989.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison Wesley, 1986.

[TYF86] T.J. Teorey, D. Yang, and J.P. Fry. A logical design methodology for relational data bases. *ACM Computing Surveys*, 18(2):197–222, June 1986.

Contents