MANAGING THE EVOLUTION OF OBJECT-ORIENTED SYSTEMS

A Thesis
Presented to the Faculty of the Graduate School
of the College of Computer Science
of Northeastern University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Paul L. Bergstein
June 1994

Abstract

Class organizations (schemas) evolve over the life cycle of object-oriented systems for a variety of reasons. This issue has recently been a subject of increasing attention in the literature of both object-oriented languages and object-oriented database systems.

One of the most common forms of evolution involves the extension of an existing system by addition of new classes of objects or the addition of attributes to the original objects. Sometimes class structures are reorganized even when the set of objects is unchanged. In this case the reorganization might represent an optimization of the system, or just a change in the users' perspective. At the other extreme, a class reorganization might reflect not only the extension and reclassification of existing objects, but also structural changes (other than addition of attributes) in the original objects.

This work provides a mathematical treatment of a calculus of class transformations. Three kinds of transformations that commonly occur in the evolution of class structures are considered: object-extending, object-preserving, and language-preserving. For each kind of transformation, methods for automating the maintenance of systems based on the evolving class structure are discussed.

The language-preserving transformations are a special case of transformations that change the structure of existing objects. If an object schema is decorated with concrete syntax, it defines not only a class structure, but also a language for describing the objects. When two schemas define the same language but different classes, the language may be used to guide the discovery of analogies between the classes. The resulting analogies may then be used to transport functionality between domains.

Acknowledgments

I would like to thank my advisor, Karl Lieberherr, for his generous support, guidance, and feedback. I would also like to thank my wife, Vickie, for her constant encouragement and understanding without which this work would not have been possible.

Contents

Abstract iii

Acknowledgments iv

1 Introduction 1

1.1 Schema extension 1

1.2 Class reorganization 2

1.3 Object restructuring 3

2 Data Model 4

2.1 Motivation for object example notation 4

2.2 Motivation for class notation 5

2.3 Class dictionary graphs 7

2.3.1 Legality conditions 14

2.3.2 Programming with class dictionary graphs 14

2.4 Object graphs 17

2.5 Related work 21

3 Extending a class organization 22

3.1 A simple example of incremental class dictionary graph learning 22

3.2 Basic Learning 25

3.3 Correctness of Basic Learning 28

3.4 Incremental Learning 29

3.5 Optimum class dictionary graph learning 31

3.6 Extending a class dictionary graph based on object examples 34

3.7 Training set 36

3.8 Related work . . . 37

4 Object preserving transformations 40

4.1 Primitive Object-Preserving Transformations . . . 40
4.2 Primitive Transformations . . . 44
4.3 Proofs . . . 46
4.3.1 Correctness . . . 46
4.3.2 Completeness . . . 47
4.3.3 Minimality . . . 49
4.4 Related work . . . 54

5 Class dictionary graph optimization 55

5.1 Practical applications of the object-preserving transformations . . . 55
5.1.1 Elimination of redundant parts . . . 55
5.1.2 Removal of singleton alternation vertices . . . 55
5.1.3 Complete Cover . . . 56
5.1.4 Partial Cover . . . 58
5.1.5 MI Minimization . . . 58
5.2 Metrics for class organizations . . . 58
5.3 Minimizing construction edges . . . 61
5.3.1 CNF Rule . . . 61
5.4 Minimizing alternation edges . . . 64
5.5 Fast algorithms for optimization . . . 67
5.5.1 Single-inheritance hierarchies . . . 67
5.5.2 Common normal form . . . 68
5.5.3 Minimizing alternation edges . . . 71
5.6 Related work . . . 78

6 Maintaining Behavioral Consistency 80

6.1 Features of language models . . . 81
6.2 The extension relation . . . 82
6.3 Class dictionary graph Extension Transformations . . . 85
6.4 Structural Consistency . . . 89
6.5 Code Transformations . . . 90

6.5.1 Untyped Language Model 90
6.5.2 Typed Language Model 93
6.6 Discussion 98
6.7 Related work 99

7 Object grammars **100**
7.1 Extended data model 100
7.1.1 Extended class dictionary graphs 100
7.1.2 Extended object graphs 101
7.2 Working with extended graphs 103
7.2.1 Learning extended class dictionary graphs 103
7.2.2 Object-preserving transformations 104
7.3 A Simple Object-oriented Programming Language 105
7.3.1 Overview 105
7.3.2 Methods, Messages, and Expressions 106
7.3.3 Built-ins 107

8 Program Evolution by Analogy **109**
8.1 Problem Description 110
8.2 Transforming a CDG Program 114
8.3 Primitive Language-Preserving Transformations 116
8.3.1 Object-preserving transformations 116
8.3.2 Renaming of vertices and edges 117
8.3.3 Nesting of parts 117
8.3.4 Unnesting of parts 118
8.3.5 Addition of lambda parts 119
8.3.6 Deletion of lambda parts 119
8.3.7 Addition of lambda alternative 119
8.3.8 Deletion of lambda alternative 119
8.3.9 Insertion of singleton construction 120
8.3.10 Deletion of singleton construction 120
8.3.11 Attribute to subclass 120
8.3.12 Subclass to attribute 121
8.4 Justification for the primitive transformations 121

8.4.1 Regular class dictionary graphs 122
8.4.2 Axiom systems for regular expressions 123
8.4.3 Completeness proof 124
8.5 CDG Program Transformations 126
8.5.1 The object-preserving transformations 131
8.5.2 Renaming of vertices and edges 131
8.5.3 Nesting of parts 131
8.5.4 Unnesting of parts 132
8.5.5 Addition of lambda parts 132
8.5.6 Deletion of lambda part 132
8.5.7 Addition of lambda alternative 132
8.5.8 Deletion of lambda alternative 133
8.5.9 Insertion of singleton construction 133
8.5.10 Deletion of singleton construction 133
8.5.11 Attribute to subclass 133
8.5.12 Subclass to attribute 133
8.6 Examples of CDG program transformations 134
8.7 Search algorithms 134
8.7.1 Regular languages 134
8.7.2 Context free languages 140
8.8 Related Work 142
8.8.1 Structure Mapping Theory 142
8.8.2 Graph Transformations in Analogical Reasoning 143
8.8.3 Analogical Program Synthesis Guided by Correctness Proofs 143

Bibliography **145**

List of Tables

1 Standard interpretation of class dictionary graphs 8
2 Standard interpretation for object graphs 17
3 Extended Interpretation 101

List of Figures

1 Lawnmower object 5
2 Construction class 5
3 Alternation class 6
4 Common parts 6
5 Satellites 10
6 Child 11
7 List 13
8 Forbidden subgraphs 15
9 Prefix expression class dictionary graph 15
10 Pocket calculator C++ implementation 16
11 Legality Rule 18
12 Class dictionary graph, ϕ, and legal object graph, ψ. 18
13 Fruit class dictionary graph 19
14 Fruit object graphs 19
15 Fruit basket objects 23
16 Fruit basket class dictionary graphs 24
17 Class dictionary graph and object graph for incremental evolution 36
18 Class dictionary graph after incremental evolution 36
19 A class dictionary graph defining 27 different `A` objects 37
20 The training set 37
21 Class Dictionary Graph ϕ_1 44
22 Class Dictionary Graph ϕ_2 45
23 Addition of Useless Alternations 50
24 Distribution of Common Parts 51
25 Part Replacement 52

26 Abstraction of Common Parts 53
27 Elimination of redundant parts 56
28 Removal of singleton alternation vertex 56
29 Complete Cover 57
30 Partial cover 59
31 MI minimization 60
32 Before transformation to CNF 63
33 After transformation to CNF 63
34 After change of weight composition 63
35 Matrix representation of alternation subgraph 65
36 Original class dictionary graph 70
37 After factoring weight and position 71
38 After factoring width, height, and length 72
39 Inventing new alternations 73
40 Covering existing alternations 75
41 Original class dictionary graph 83
42 Object-equivalent class dictionary graph 84
43 Extended class dictionary graph 85
44 Steps in the object-preserving transformation 87
45 Steps in the object-extending transformation 88
46 Program A 108
47 Language-equivalent Class Dictionary Graphs 111
48 Program B 115
49 Nesting of parts 118
50 Addition of lambda alternative 119
51 Insertion of singleton construction 120
52 Attribute to subclass 121
53 Closure operator 122
54 Solution of equations 125
55 $(a + (b + c)) = ((a + b) + c)$ 126
56 $(a \cdot (b \cdot c)) = ((a \cdot b) \cdot c)$ 126
57 $(a + b) = (b + a)$ 126
58 $(a \cdot (b + c)) = ((a \cdot b) + (a \cdot c))$ 127

59 $((a+b)\cdot c) = ((a\cdot c)+(b\cdot c))$ 128
60 $(a+a) = a$ 128
61 $(a\cdot\lambda) = a$ 128
62 $(a*) = (\lambda+(a\cdot(a*)))$ 129
63 $(a*) = ((\lambda+a)*)$ 130
64 Program C 135
65 Program D 136
66 Program E 137
67 Program to calculate weight of brick pile 138
68 After adding balloons to the pile 139

Chapter 1

Introduction

Class organizations (schemas) evolve over the life cycle of object-oriented systems for a variety of reasons. This issue has recently been a subject of increasing attention in the literature of both object-oriented languages and especially object-oriented database systems: [Opd92, Ber92, Ber91, Cas91, CPLZ91, DZ91, Bar91, LH90, AH88, BKKK87, PS87, SZ86].

One of the most common forms of evolution involves the extension of an existing schema by addition of new classes of objects or the addition of attributes to the original objects. Sometimes class structures are reorganized even when the set of objects is unchanged. In this case the reorganization might represent an optimization of the system, or just a change in the users' perspective. At the other extreme, a class reorganization might reflect not only the extension and reclassification of existing objects, but also structural changes (other than addition of attributes) in the original objects.

1.1 Schema extension

Chapter 3 addresses the structural aspects of schema extension. In particular, we consider how a schema can be automatically updated to accommodate new objects based on examples. The problem of updating the original objects (e.g. persistent instances in a database) so that they remain consistent with the new schema has been addressed by others, but a simple solution is presented in Chapter 3.

In Chapter 6, we return to the problem of schema extension to address the issue of behavioral consistency. It is shown that once a schema has been extended, it is possible to automate the modifications of methods so that programs based on the schema will

exhibit behavior identical to that which was exhibited before the extension. This problem is examined using both strongly typed (C++) and untyped (CLOS) language models, and it is shown how the type system of strongly typed languages can complicate the maintenance of evolving object-oriented systems.

The algorithms developed in Chapter 3 are also useful in contexts other than evolution. In class-based object-oriented languages, the user has to define classes before objects can be created. For the novice as well as for the experienced user, the class definitions are a non-trivial abstraction of the objects. The evolution algorithm can be used to automate the abstraction simply by starting with an "empty" schema and extending it by presenting the algorithm with a set of example objects.

1.2 Class reorganization

In another common form of schema evolution, the class structures are reorganized but the set of objects is unchanged. Consider, for example, the set of objects: {`motorboat`, `sailboat`, `automobile`, `bicycle`}. In one context it might be desirable to classify these objects as either `water-vehicles` or `land-vehicles`. In another context, however, it might make more sense to classify the objects as either `motorized` or `non-motorized`. A small set of primitive transformations that can be used to achieve any reorganization of classes that preserves the set of objects is presented in Chapter 4. A constructive proof of the completeness of the set of primitives is presented. Since the proof is constructive, there is an algorithm to determine whether two arbitrary class organizations define the same set of objects, and if they do, to find a sequence of primitives to transform one organization to the other.

Chapter 5 discusses some metrics for schema design, and presents an algorithm for optimizing a class organization based on the the primitive object-preserving transformations. In Chapter 6 the problem of class reorganization is reexamined with the focus on behavior, in the same manner as for the schema extension case.

1.3 Object restructuring

While the schema extension and class reorganization are mainly concerned with the organization of objects into classes, object restructuring is concerned primarily with the organization of attributes, or "parts", into objects. Here the concern is how to modify the code of an object-oriented program if the class definitions are changed so that the same data is organized into a different object structure. If the new objects hold the same data as the original objects, the class structures can be considered in some way analogous. The problem is to find a mapping of the code (methods) from the old class structure to the new one.

In Chapter 7 the data model presented in Chapter 2 is extended by decorating the schema with concrete syntax. In the extended model, the schema defines both a set of objects and a language for representing the objects textually. The concept of a part is extended to mean either another object or a text string. An interesting class of transformations investigated in Chapter 8 are those that change the structure of the objects, but preserve the language defined by the schema. If the concrete syntax specified in a schema is meaningful and a transformation preserves the defined language, it is reasonable to hypothesize that the objects are intended to represent the same data in the transformed schema as in the original. Therefore, we can expect to find a mapping of the methods from the old class structure to the new one which will preserve the behavior of the system.

Chapter 2

Data Model

2.1 Motivation for object example notation

The importance of objects extends beyond the programmer concerns of data and control abstraction and data hiding. Rather, objects are important because they allow the program to model some application domain in a natural way. In [MMP88], the execution of an object-oriented program is viewed as a physical model consisting of objects, each object characterized by parts and a sequence of actions. It is the modeling that is significant, rather than the expression of the model in any particular programming language. We have devised a programming language independent object example notation to describe objects in any application domain.

The objects in the application domain are naturally grouped into classes of objects with similar subobjects. For our object example notation it is important that the designer names those classes consistently. Each object in the application domain has either explicitly named or numbered subobjects. It is again important for our object example notation that the explicitly named parts are named consistently. This consistency in naming classes and subparts is not difficult since it is naturally implied by the application domain.

An object is described by giving its class name, followed by the named parts. The parts are either physical parts of the object (e.g., wheels of a lawnmower) or attributes or properties (e.g. model number). An object example is in Figure 1 which defines a lawnmower object with 6 parts: 5 physical parts (legs and motor) and one attribute: model. The object example also indicates that the four wheels have no parts and that the motor is a gasoline engine object with one part called fuelTank.

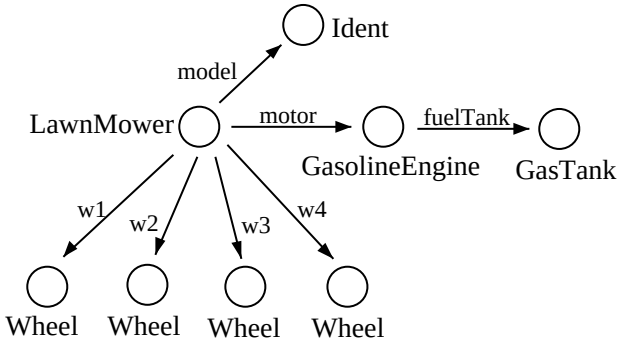


Figure 1: Lawnmower object

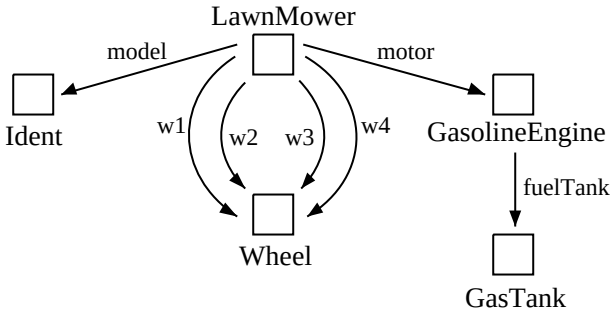


Figure 2: Construction class

2.2 Motivation for class notation

We use a class notation which uses two kinds of classes: construction and alternation classes.[1] 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. We view a part as a high-level concept which might be implemented as a method, not necessarily as an instance variable. An example of a construction class corresponding to the object in Figure 1 is in Figure 2.

Each construction class inductively defines a set of objects which can be thought of 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 lawnmower can be either a gasoline engine or an electric motor. So the objects we want to store in the motor part of the lawnmower are either gasoline

[1] In practice we use a third kind, called repetition classes, which can be expressed in terms of construction and alternation [Lie88].

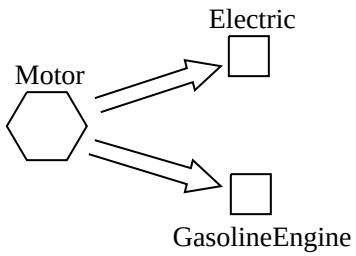


Figure 3: Alternation class

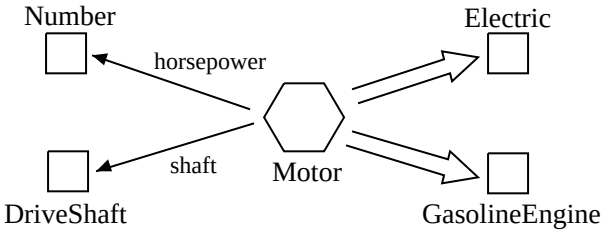


Figure 4: Common parts

engine or electric motor objects. We use alternation classes to define such union classes. An example of an alternation class is in Figure 3. GasolineEngine and Electric are called alternatives of the alternation class. Often the alternatives have some common parts. For example, each motor has a drive shaft. We use the notation in Figure 4 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.

Parts which are common to more than one class may be implemented by inheritance from an alternation class. In Figure 4, GasolineEngine and Electric inherit from Motor. Class Motor has methods and/or instance variables to implement the parts shaft and horsepower.

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.

2.3 Class dictionary graphs

To describe multiple inheritance class libraries with part-of and inheritance relationships we use graphs with construction and alternation vertices and edges. The information stored in class dictionary graphs is considered to be essential for object-oriented design, as Booch writes [Boo91]: "We have found it essential to view a system from both perspectives, seeing its "kind-of" hierarchy as well as its "part-of" hierarchy."

The concept of a part class and a part object which is used throughout the text needs further explanation. A part object does not have to be a physical part; any attribute of an object is a part of it. We say that object o_2 is a part of object o_1, if "o_1 knows about o_2". Therefore, our 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. The concept of a part-class is a high-level concept which does not reveal implementation detail; the parts might be implemented by operations.

Class dictionary graphs focus only on part-of and inheritance relations between classes. One notably absent relation is the "uses" relation between class operations (see e.g., [LG86]). The call relationships between classes describe important design information, e.g., for checking the Law of Demeter [LHR88]. However, we find that class dictionary graphs as presented here are a useful design abstraction which can be debugged independently. Only in later design stages do we augment class dictionary graphs with other information such as operations.

We call a class S a supplier class to a class C, if in C we use the functions of class S. The part classes of a class C are one important kind of supplier classes of C. If a design follows the Law of Demeter, then there are only two other kinds of supplier classes (which are not considered in a class dictionary graph): argument classes of functions of C and classes of objects which are created in functions of C. It is an important insight of our approach that it is very worthwhile for a first design step to consider only a limited set of supplier classes (the part classes) and inheritance.

In database terminology, a class dictionary graph is an object base schema with only a minimal set of integrity constraints. Class dictionary graphs can be viewed as an adaptation of extended entity-relationship diagrams for object-oriented design [TYF86]. More recently, graphs have been used to model object-oriented data bases in [LRV90, GPG90].

Graph	Object-oriented Design
Vertex	**Class**
construction	instantiable class with members defined by construction edges (including "inherited" edges)
alternation	abstract class with subclasses defined by alternation edges
Edge	**Class Relationship**
construction	part-of relationship, "uses", "knows", — labels are part names
alternation	inheritance relationship, specialization, classification

Table 1: Standard interpretation of class dictionary graphs

The definition of a class dictionary graph is motivated by the interpretation in object-oriented design given in Table 1. 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.

Definition 2.1. *A* **class dictionary graph**[2]*, 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: $(v \stackrel{l}{\longrightarrow} w) \in EC$ 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: $(v \Longrightarrow w) \in EA$ 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 2.2.

[2]The class dictionary graphs described here are a specialization of the class dictionaries described in [Lie88], [LR88]. The class dictionary graphs contain all the information necessary for many applications; however they omit: terminal classes, concrete syntax, ordering of parts. For presenting design algorithms, e.g., we are not concerned with the grammar aspects of class dictionaries since they would only clutter the presentation of the algorithms. We also omit optional and repeated part-of relationships since they can be easily expressed in terms of the primitives given here.

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

- *The* **alternation vertices** *are defined by:*
  $VA = \{v \mid v \in V, \exists w \in V : (v \Longrightarrow w) \in EA\}$.
  In other words, the alternation vertices have at least one outgoing alternation edge.

Sometimes, when we want to talk about the construction and alternation vertices of a class dictionary, it is more convenient to describe a class dictionary graph as a tuple which contains explicit references to VC and VA: $\phi = (VC, VA, \Lambda; EC, EA)$.

In standard object-oriented terminology we describe here the accepted programming rule: "Inherit only from abstract classes" [JF88]. This rule can be exploited to derive an analogy between class dictionary graphs and grammars.

We use the following graphical notation, based on [TYF86], for drawing class dictionary graphs: squares for construction vertices, hexagons for alternation vertices, thin arrows for construction edges and wide arrows for alternation edges.

Example 2.1. *Figure 5 shows a class dictionary graph for satellites. Satellites can either be military or civilian and they also can be either low orbit or geosynchronous. Military satellites belong to a country and have a contract number assigned. Civilian satellites are described by a manufacturer. For geosynchronous satellites we store their position while for orbiting satellites we represent their path. For further illustration we give the components of the formal definition, i.e.,*

```
V  = { Satellite, Orbit, Low_orbit, Geosynchronous,
       Military, Civilian, Country, Position, Manufacturer, Path},
VC = { Low_orbit, Geosynchronous, Military, Civilian, Country,
       Contract, Position, Manufacturer, Path,
VA = { Satellite, Orbit },
EC = { (Satellite, Orbit, orbit), (Low_orbit, Path, p),
       (Geosynchronous, Position, p), (Military, Contract, c),
       (Military, Country, country), (Civilian, Manufacturer, m) },
EA = { (Satellite, Military), (Satellite, Civilian),
       (Orbit, Low_orbit), (Orbit, Geosynchronous) },
```

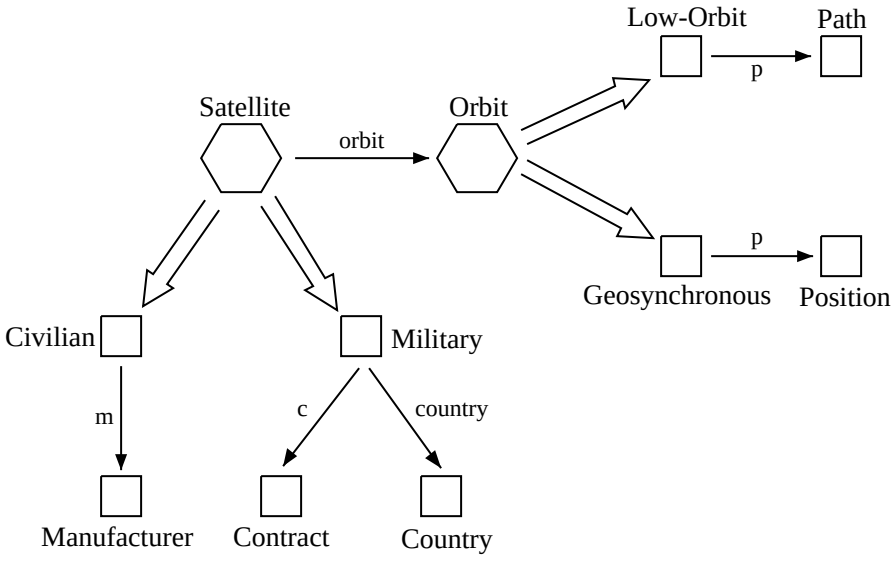


Figure 5: Satellites

Λ = {c, country, m, orbit, p }.

The definition of VC implies that $EA \subseteq VA \times V$, since an alternation edge cannot start at a construction vertex. We use V_ϕ, VC_ϕ, VA_ϕ etc. to refer to the components of class dictionary graph ϕ.

When we draw a class dictionary graph, the vertices are labeled so that we can conveniently refer to particular vertices in our discussion. The standard interpretation implies that the labels on construction vertices are significant. Consider two isomorphic class dictionary graphs each with only a single construction vertex and no edges. 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. On the other hand, changing the labels of the alternation vertices (names of abstract classes in the standard interpretation) does not effect the defined objects. Therefore, we adopt the following convention for labeling the vertices of class dictionary graphs: Labels of alternation vertices are local to the class dictionary graph in which they occur; labels of construction vertices are global. That is, if two class dictionary graphs have construction vertices with the same label, it means that the *same vertex* (same class under the standard interpretation) belongs to both graphs. However, we may in general assume that different class dictionary graphs have disjoint sets of alternation vertices regardless of their labels.

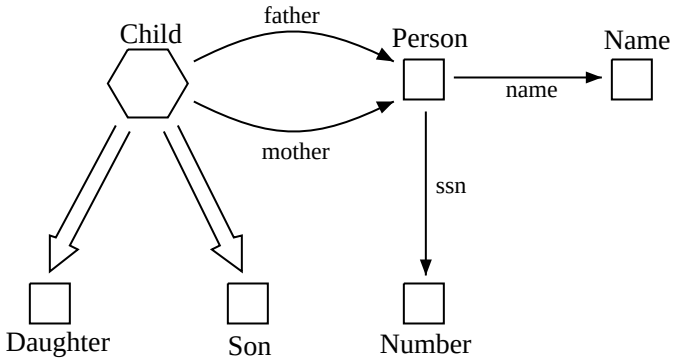


Figure 6: Child

The same semantics apply when we denote the sets of vertices in a class dictionary graph textually. The identifiers we use to denote alternation vertices are of local scope whereas the identifiers we use to denote construction vertices have global scope.

Later we give conditions which make a class dictionary graph into a legal class dictionary graph. The interpretation in Table 1 is only one possible interpretation which we call the standard interpretation. The motivation behind the abstract alternation/construction terminology is that there are several useful interpretations of class dictionary graphs. In one of those interpretations, a construction vertex is interpreted as an operation. We sometimes use the standard interpretation to give intuitive explanations of relationships and algorithms.

The graphical notation presented above is useful for understanding class structures, but a textual notation may be more suitable for implementation purposes. Therefore, we also use a textual notation for class dictionary graphs which serves as an easy to learn, terse input notation for the Demeter CASE tool. To describe class dictionary graphs textually we use an adjacency representation which gives the successors for each vertex. For example, the vertex Child in the graph in Figure 6 is described by:

```
Child :
  // two alternation edges
  Daughter | Son
  *common*
    // two construction edges
    <father> Person
    <mother> Person.
```

and the construction vertex Person is described by:

```
Person =
  // two construction edges
  <name> Name
  <ssn> Number.
```

A "//" introduces a comment line and *common* is syntactic sugar to separate the alternation edges from the construction edges.

Please note that the syntax for an alternation vertex/abstract class, although very natural from a graph-theoretic point of view, appears unnatural from the point of view of today's programming languages: In most programming languages which support the object-oriented paradigm, the inheritance relationships are described in the opposite way. Each class indicates from where it inherits. Of course, we can easily generate this information from class dictionary graphs, but we feel that the Demeter notation is easier to use for design purposes. One reason is that the design notation shows the immediate subclasses of a class and therefore promotes proper abstraction of common parts. Another reason is that a class does not contain information about where it inherits from and therefore the class can be easily reused in other contexts.

Example 2.2. *The following text describes the class dictionary graph in Figure 7:*

```
List : Empty | Nonempty *common*.
Empty = .
Nonempty = <first> Element <rest> List.
Element = .
```

The two edges leaving from List are alternation edges. The labeled edges are construction edges. In this example we have the following class dictionary graph:

```
V = {Empty, Nonempty, Element, List},
VC = {Empty, Nonempty, Element},
VA = {List},
EC = {(Nonempty, List, rest), (Nonempty, Element, first)},
EA = {(List, Nonempty), (List,Empty)},

Λ  =  {first, rest}.
```

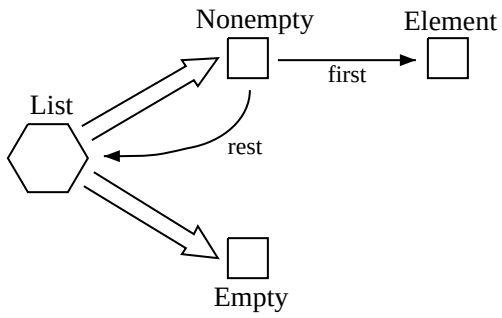


Figure 7: List

Definition 2.3. *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* $v \stackrel{\star}{\Longrightarrow} w$*):*

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

In other words, the **alternation-reachable** relation is the reflexive, transitive closure of the EA relation.

In the standard interpretation, $(v \stackrel{\star}{\Longrightarrow} w)$ means that either w inherits from v or $w = v$.

Sometimes when we want to discuss the inheritance hierarchy, it is convenient to refer to the alternation subgraph of a class dictionary graph. The alternation subgraph contains all of the alternation vertices and alternation edges plus the construction vertices that have incoming alternation edges.

Definition 2.4. *The* **alternation subgraph** *of a class dictionary graph,*
$\phi = (VC, VA, \Lambda;\ EC, EA)$*, is a directed acyclic graph (DAG),* $G = (V', EA)$*,*
where $V' = VA \cup \{v \in VC | \exists u : (u \Longrightarrow v) \in EA\}$*.*

It is often helpful to think of each alternation vertex as representing a set of associated construction vertices. This set, $\mathcal{A}(v)$, consists of all the construction vertices which are alternation reachable the vertex, v. If v is an alternation vertex with an incoming construction edge, $(u \stackrel{l}{\longrightarrow} v)$, the construction vertices in $\mathcal{A}(v)$ represent the concrete classes which might be used to instantiate the l part of u objects. If v has an outgoing construction edge, $(v \stackrel{l}{\longrightarrow} w)$, the construction vertices in $\mathcal{A}(v)$ represent the concrete classes which inherit the l part from v.

Definition 2.5. *The* **associated classes** *of a vertex, v, in a class dictionary graph, $\phi = (VC, VA, \Lambda;\ EC, EA)$, is the set of all construction vertices which are alternation-reachable from v:*

$$\mathcal{A}(v) = \{v' | v \stackrel{\star}{\Longrightarrow} v' \text{ and } v' \in VC\}$$

2.3.1 Legality conditions

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

Definition 2.6. *A class dictionary graph $\phi = (V, \Lambda;\ EC, EA)$ is* **legal** *if it satisfies the following two conditions:*

1. *Cycle-free alternation condition:*

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

2. *Unique labels condition:*

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

When we refer to a class dictionary graph in the following we mean a legal class dictionary graph, unless we specifically mention illegality.

The cycle-free alternation condition 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 condition says that a class may not inherit from itself.

The unique labels condition guarantees that "inherited" construction edges are uniquely labeled and excludes class dictionary graphs which contain the patterns shown in Figure 8. Other mechanisms for uniquely naming the construction edges could be used, e.g., the renaming mechanism of Eiffel and the overriding of part classes [Mey88]. The theory does not seem to be affected significantly by small changes such as this.

2.3.2 Programming with class dictionary graphs

To motivate the usefulness of class dictionary graphs further, we show with a simple example how we use them to simplify programming. We have developed a CASE tool for C++

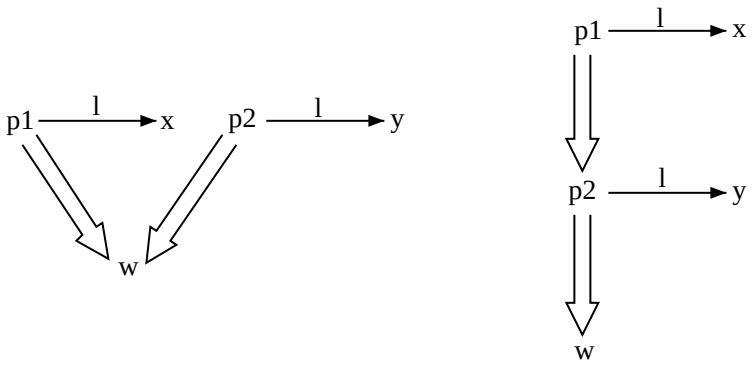


Figure 8: Forbidden subgraphs

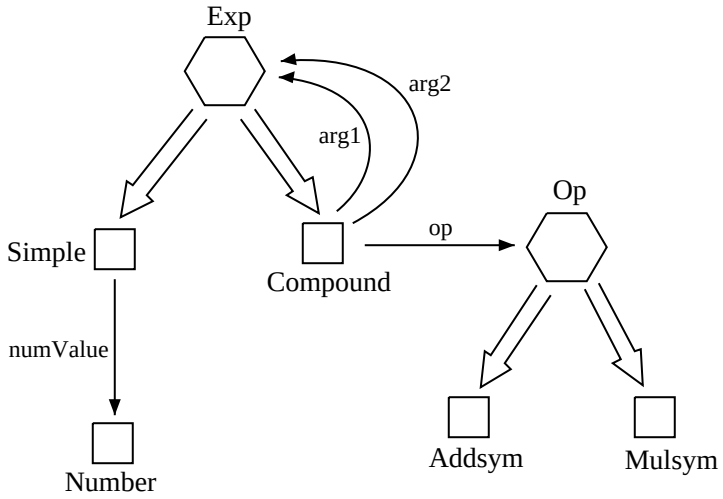


Figure 9: Prefix expression class dictionary graph

[Str86], the C++ Demeter System [LR88], which maps class dictionary graphs into a C++ class library which is then enhanced manually with C++ member functions implementing the application. To each construction vertex corresponds a C++ class with a constructor and to each alternation vertex corresponds an abstract C++ class.

Consider the class dictionary graph in Figure 9. We want to implement a pocket calculator which evaluates the object equivalent of expressions such as `3`, `(+ 3 (+ 2 1))`, `(* 3 (+ 2 1))`. The complete C++ program which has to be written by the user is given in Figure 10. The missing parts of the C++ program are generated from the class dictionary graph in Figure 9 by the Demeter System.

```
int Exp::eval() = 0; // pure virtual

int Simple::eval()
{ return numvalue->eval(); }

int Number::eval()
{ return val; }

int Compound::eval()
{ return op->apply_op(arg1->eval(), arg2->eval()); }

int Op::apply_op(int n1,int n2) = 0; // pure virtual

int Addsym::apply_op(int n1,int n2)
{ return n1 + n2; }

int Mulsym::apply_op(int n1,int n2)
{ return n1 * n2; }
```

(user-written)

Figure 10: Pocket calculator C++ implementation

2.4 Object graphs

We have defined the concept of a class dictionary graph which mathematically captures some of the structural knowledge which object-oriented programmers use. Next we define object graphs and their relation to class dictionary graphs. An object graph defines a hierarchical object and is motivated by the interpretation of an object graph, called the standard interpretation, given in Table 2.

Graph	Object-oriented Design
vertex	object
immediate successor	immediate subpart or component
edge label	part name

Table 2: Standard interpretation for object graphs

Definition 2.7. *An* **object graph**, ψ, *is a directed graph* $\psi = (W, S, \Lambda_\psi;\ E, \lambda)$ *where:*

- W *is a finite set of vertices.*
- S *is an arbitrary finite set.*
- Λ_ψ *is a set of labels.*
- E *is a ternary relation on* $W \times W \times \Lambda_\psi$. *If* $(v \xrightarrow{l} w) \in E$ *we call* l *the label of the edge* $(v \xrightarrow{l} w)$. *No two edges outgoing from the same vertex may have the same label. That is,* $\forall v, w, w' \in W, l \in \Lambda_\psi$ *such that* $w \neq w' : \{(v \xrightarrow{l} w), (v \xrightarrow{l} w')\} \not\subseteq E$
- $\lambda : W \rightarrow S$ *is a function that maps each vertex of* ψ *to an element of* S.

Normally, the set S is a subset of the construction vertices of some class dictionary graph. In the standard interpretation, the function λ maps each object in an object graph to the class of which it is an instance. We use a graphical notation for object graphs similar to that for class dictionary graphs. Vertices are represented by circles and edges by labeled arrows. The vertices are labeled with their class names (their mapping under λ). In case we wish to distinguish more than one instance of a class, the labels may be prefixed with an instance name followed by a ":".

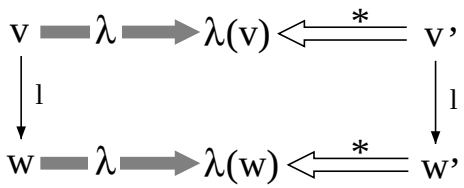


Figure 11: Legality Rule

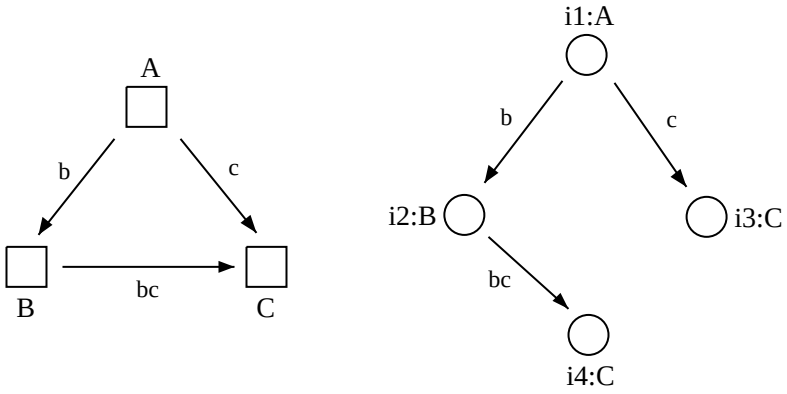


Figure 12: Class dictionary graph, ϕ, and legal object graph, ψ.

Not every object graph with respect to a class dictionary graph is legal; intuitively, the object structure has to be consistent with the class definitions. Each object can only have parts as prescribed in the class definition and the parts prescribed in the class definitions must appear in the objects (see Figure 11).

Definition 2.8. *An object graph, $\psi = (W, S, \Lambda_\psi;\ E, \lambda)$, is* **legal** *with respect to a class dictionary graph, $\phi = (VC, VA, \Lambda;\ EC, EA)$, iff for each vertex, $v \in W$:*

- $\lambda(v) \in VC$
- $\forall w, l$ where $(v \xrightarrow{l} w) \in E : \exists (r \xrightarrow{l} s) \in EC$ such that $r \stackrel{\star}{\Longrightarrow} \lambda(v)$ and $s \stackrel{\star}{\Longrightarrow} \lambda(w)$
- $\forall (r \xrightarrow{l} s) \in EC$ where $r \stackrel{\star}{\Longrightarrow} \lambda(v) : \exists w \in W$ such that $(v \xrightarrow{l} w) \in E$

Example 2.3. *Consider the graphs in Figure 12. The object graph, ψ, is legal with respect to the class dictionary graph, ϕ. The object graph is given by: $W = \{i1, i2, i3, i4\}$, $E = \{(i1 \xrightarrow{b} i2), (i1 \xrightarrow{c} i4), (i2 \xrightarrow{bc} i3)\}$, $\Lambda_\psi = \{b, bc, c\}$, $\lambda = \{i1 \rightarrow A, i2 \rightarrow B, i3 \rightarrow C, i4 \rightarrow C\}$.*

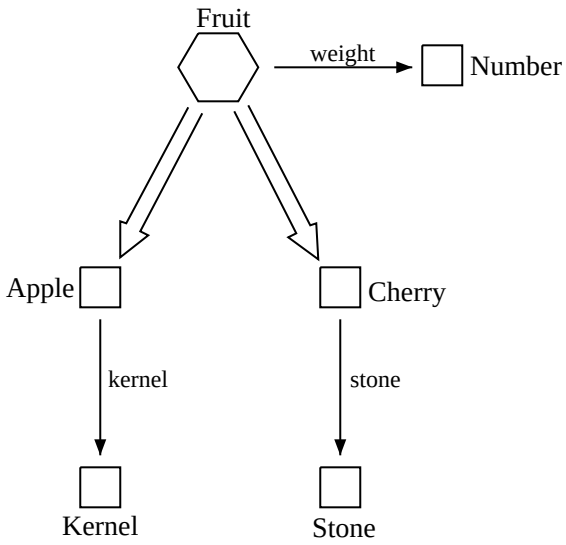


Figure 13: Fruit class dictionary graph

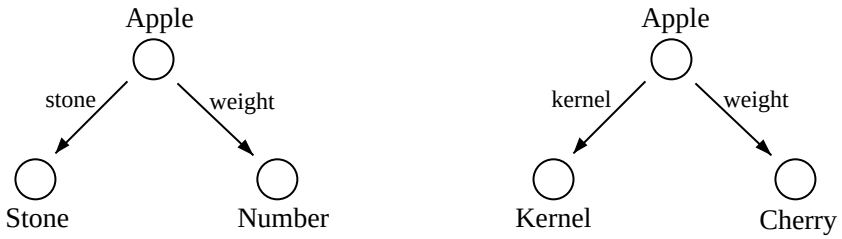


Figure 14: Fruit object graphs

Example 2.4. *Consider object graphs in Figure 14 which are illegal with respect to the class dictionary graph in Figure 13. The first object graph is illegal since apples don't contain stones and the second because* `Cherry` *is not alternation-reachable from* `Number`.

Definition 2.9. *The set of all legal object graphs with respect to a class dictionary graph, ϕ, is called* $\boldsymbol{Objects}(\phi)$.

The graphical notation for object graphs is useful for understanding object structures, but a textual notation may be more suitable for implementation purposes. Therefore, we also use a textual notation for object graphs. To describe object graphs textually we use an adjacency representation which also shows the mapping of object graph vertices to class dictionary graph vertices:

```
inst1:v1(
  <successor1> inst2:v2( ... )
  <successor2> inst3:v3( ... )
  ...
  <successorn> instn:vn( ... ))
```

The vertices correspond to the instance names. The name after the instance name is preceded by a ":" and gives the label assigned by λ. The edge labels are between the < and > signs.

For describing shared objects, we also use the notation:

```
inst1:v1(
  <successor1> inst2)
```

where inst2 is an object identifier defined elsewhere. Each object identifier has to be defined once. Vertices which are not shared need not be named explicitly. Therefore the instance name and ":" may be omitted.

Example 2.5. *The object graph in Figure 12 can be represented textually as:*

```
i1:A(
  <b> i2:B(
    <bc> i3:C())
  <c> i4:C())
```

The definitions above relate a class dictionary graph with a set of object graphs. In object-oriented programming language terminology, a class dictionary graph corresponds to a set of class definitions and the object graphs correspond to the objects which can be created calling "constructor" functions of the classes. In some languages, e.g., C++, the class definitions considerably restrict the objects which can be created. The definitions above demand even more discipline than C++.

In the context of evolution, we often wish to discuss object graphs that are not legal with respect to the current class dictionary graph. We sometimes refer to these object graphs as *object example graphs* since our goal is often to modify a class dictionary graph so that it will become compatible with a new set of objects based on examples.

2.5 Related work

The axiomatic model which is used in this paper is new but similar data models exist in the literature. In particular, the notions of "alternation" and "construction" appear as "classification" and "aggregation" in both Hull and Yap's Format Model [HY84] and Kuper and Vardi's LDM [KV84]. Ait-Kaci's feature structures [AKN86] are also related to the Demeter kernel model. Our abstraction algorithms presented in Chapter 3 can be adapted to abstract feature structures from examples.

Other related work in the data base field is described in: [AH88, BMW86, TL82].

Chapter 3

Extending a class organization

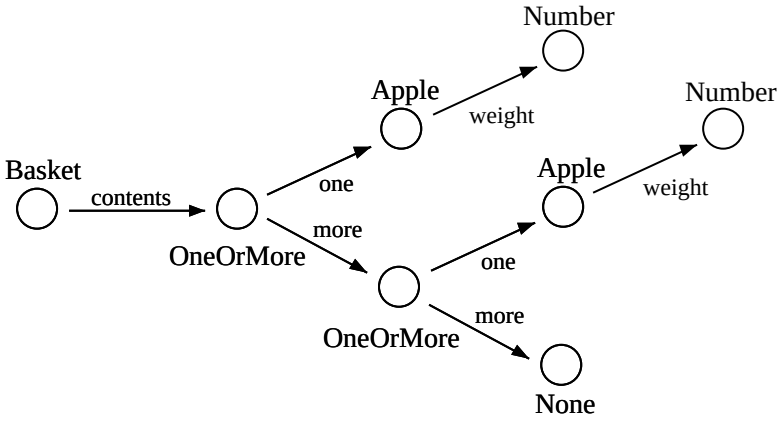
In class-based object-oriented languages, the user has to define classes before objects can be created. For the novice as well as for the experienced user, the class definitions are a non-trivial abstraction of the objects. We claim it is easier to initially describe certain example objects and to get a proposal for an optimal set of class definitions generated automatically than to write the class definitions by hand.

In this section an algorithm for learning a class dictionary graph from a set of object examples is presented. This algorithm learns a correct (but not optimal) class dictionary graph from a list of object example graphs. An algorithm for learning class dictionary graphs incrementally is also presented. The ability to expand a class dictionary incrementally as new object examples are presented is an important consideration in software engineering. The algorithm is then extended to incrementally learn an optimal class dictionary graph when the optimum is a single inheritance class dictionary.

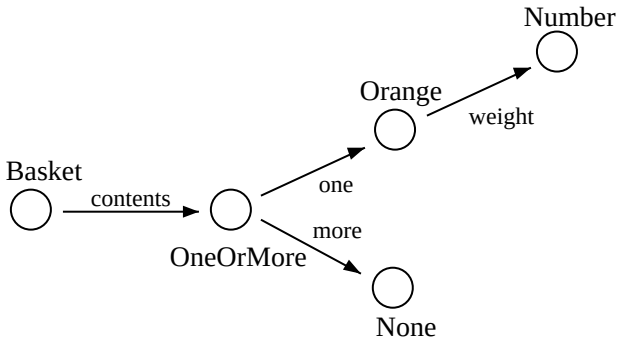
3.1 A simple example of incremental class dictionary graph learning

Example 3.1. *Consider the two object example graphs in Figure 15 which represent a basket containing two apples and a basket with an orange, respectively.*

After seeing the first object example graph, the learning algorithm generates the class dictionary graph in Figure 16a. Now when the second object example is presented, the algorithm will learn the class dictionary graph in Figure 16b.



(a)



(b)

Figure 15: Fruit basket objects

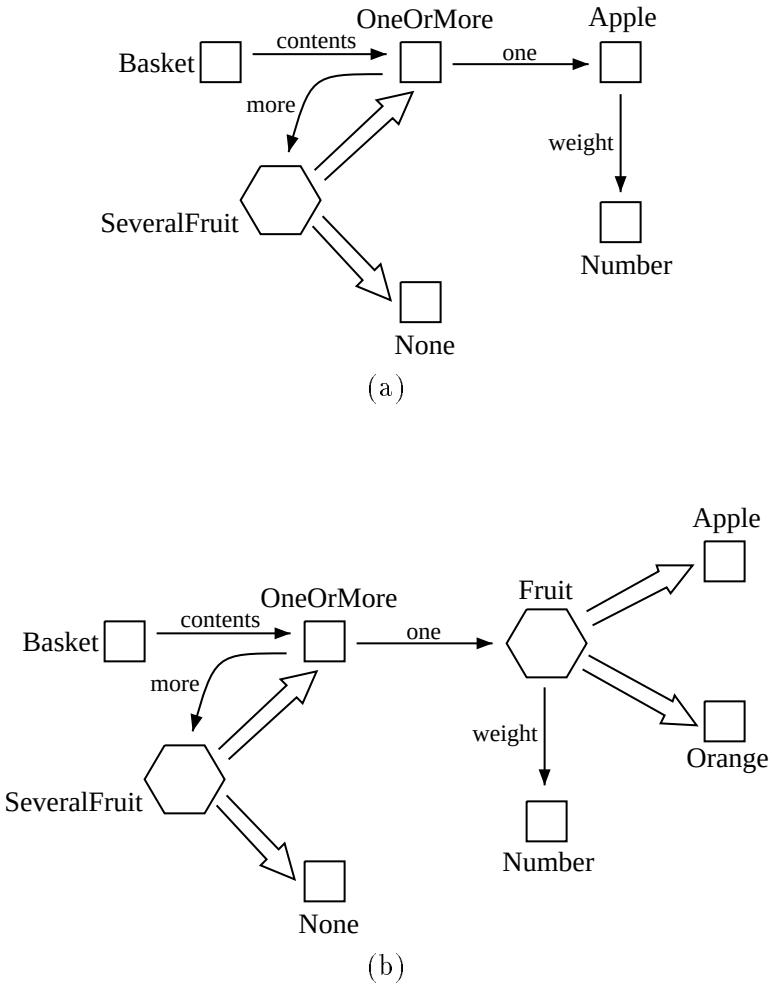


Figure 16: Fruit basket class dictionary graphs

Notice that the algorithm "invents" two abstract classes, *SeveralFruit* and *Fruit*. Since both subclasses of *Fruit* have a *weight* part, that part is attached to the *Fruit* class and is inherited in the *Apple* and *Orange* classes.

A sample program to calculate the weight of a fruit basket is given below. All of the user written code is shown. The class definitions and remaining code are generated automatically from the class dictionary by the Demeter System CASE tool.

```
// Basket = <contents> SeveralFruit.
Number Basket::get_weight()
{ return contents->get_weight(); }
```

```
// SeveralFruit : None | OneOrMore.
virtual Number SeveralFruit::get_weight()
{}

// OneOrMore = <one> Fruit <more> SeveralFruit.
Number OneOrMore::get_weight()
{ return (one->get_weight() + more->get_weight()); }

// None = .
Number None::get_weight()
{ return Number(0); }

// Fruit : Apple | Orange *common* <weight> Number.
Number Fruit::get_weight()
{ return *weight; }
```

3.2 Basic Learning

Given a list of object example graphs, the basic learning algorithm[1] will learn a class dictionary graph, ϕ, such that the set of objects defined by ϕ includes all of the examples. Furthermore, the algorithm insures that the set of objects defined by the learned class dictionary graph is a subset of the objects defined by any class dictionary graph that includes all of the examples. Intuitively, we learn a class dictionary graph that only defines objects that are "similar" to the examples.

Formally, given a legal list of object example graphs, $\psi_1, \psi_2, ..., \psi_n$, we learn a legal class dictionary graph, ϕ, such that $Objects(\phi) \supseteq \{\psi_1, \psi_2, ..., \psi_n\}$, and for all legal class dictionary graphs, ϕ', where $Objects(\phi') \supseteq \{\psi_1, \psi_2, ..., \psi_n\}$, $Objects(\phi) \subseteq Objects(\phi')$.

If there is no legal class dictionary graph that defines a set of objects that includes all of the examples, we say that the list of object example graphs is not legal. The following definition gives the conditions under which a list of object example graphs is legal.

Definition 3.1. *A list of object example graphs $\psi_1, ..., \psi_n$ is* **legal** *if all vertices which have*

[1] An informal description of the algorithm appears in [LBSL90].

the same element $s \in S$ as label (under λ_{ψ_i} for some $i, 1 \leq i \leq n$) have either outgoing edges with the same labels (under E for ψ_i) or no outgoing edges at all.

A legal list of object example graphs $\psi_1, ..., \psi_n$ of the form $\psi = (W_\psi, S_\psi, \Lambda_\psi; E_\psi, \lambda_\psi)$ is translated into a class dictionary graph $\phi = (V, \Lambda;\ EC, EA)$ as follows:

1. $\Lambda = \bigcup_{1 \leq i \leq n} \Lambda_{\psi_i}$

   The construction edges of the class dictionary graph are given the same labels as the edges in the object example graph.

2. $VC = \{r \mid r = \lambda_{\psi_i}(v)$ and $v \in W_{\psi_i}$ where $1 \leq i \leq n\}$

   We interpret λ as a function that maps objects to their classes. For each class that appears in an object example, we generate a construction class which is represented as a construction vertex in the class dictionary graph.

3. $VA = \{(r, l) \mid r \in VC, l \in \Lambda, \exists i, j, v1, v2, w1, w2 : (v1 \xrightarrow{l} w1) \in E_{\psi_i},$
   $(v2 \xrightarrow{l} w2) \in E_{\psi_j}, \lambda_{\psi_i}(v1) = \lambda_{\psi_j}(v2) = r, \lambda_{\psi_i}(w1) \neq \lambda_{\psi_j}(w2)\}$

   When we learn that objects of class r have a part labeled l that is not always of the same class, we create an abstract class represented in the class dictionary graph as an alternation vertex (r, l). In step 6, we will make each of the part's possible classes a subclass of the new abstract class.

4. $V = VC \cup VA$

   The vertices of the class dictionary graph are given by the union of the construction vertices and alternation vertices.

5. $EC = \{(r \xrightarrow{l} s) \mid r, s \in V, \exists i, v, w : (v \xrightarrow{l} w) \in E_{\psi_i}, \lambda_{\psi_i}(v) = r, \lambda_{\psi_i}(w) = s,$
   $(r, l) \notin VA\} \cup \{(r \xrightarrow{l} (r, l)) \mid r \in V, (r, l) \in VA\}$

   If an object of class r has a part of class s with label l, then we create a construction edge from the construction vertex representing r to the construction vertex representing s with label l. But if the part can have more than one class, in which case an alternation vertex representing all of the possible classes was created in step 3, we instead create a construction edge to that alternation vertex.

6. $EA = \{((r,l) \Longrightarrow s) | (r,l) \in VA, s \in V, \exists i, v, w : (v \xrightarrow{l} w) \in E_{\psi_i},$
   $\lambda_{\psi_i}(v) = r, \lambda_{\psi_i}(w) = s\}$

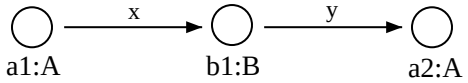
   Finally, we create an alternation edge from each alternation vertex (representing an abstract class) to each vertex which represents a subclass.

The following example serves to illustrate the operation of the algorithm:

Example 3.2.

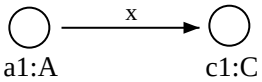
ψ_1: *a1:A(⟨x⟩ b1:B(⟨y⟩ a2:A))*

- $W = \{a1, a2, b1\}$
- $S = \{A, B\}$
- $\Lambda = \{x, y\}$
- $E = \{(a1, b1, x), (b1, a2, y)\}$
- $\lambda_W = \{a1 \rightarrow A, a2 \rightarrow A, b1 \rightarrow B\}$



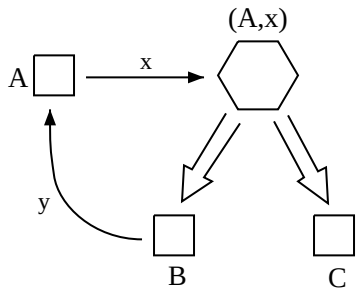
ψ_2: *a1:A(⟨x⟩ c1:C())*

- $W = \{a1, c1\}$
- $S = \{A, C\}$
- $\Lambda = \{x\}$
- $E = \{(a1, c1, x)\}$
- $\lambda_W = \{a1 \rightarrow A, c1 \rightarrow C\}$



ϕ :

- $\Lambda = \{x, y\}$
- $VC = \{A, B, C\}$
- $VA = \{(A, x)\}$
- $V = \{A, B, C, (A, x)\}$
- $EC = \{(B, A, y), (A, (A, x), x)\}$
- $EA = \{((A, x), B), ((A, x), C)\}$



3.3 Correctness of Basic Learning

In this section we prove that the set of objects defined by a class dictionary graph, ϕ, learned from a legal list of object example graphs, $\psi_1, \psi_2, ...\psi_n$, is a subset of the objects defined by any class dictionary graph that includes all of the examples.[2]

Consider the construction vertices in ϕ and note that $r \in VC_\phi$ iff there is some ψ_i for $1 \leq i \leq n$ with a vertex, $v \in E_{\psi_i}$, such that $r = \lambda(v)$.

Next, consider the construction edges in ϕ. First, note that every construction edge has a construction vertex as its source (step 5). Second, a construction vertex, r, has an outgoing construction edge, $(r \xrightarrow{l} s)$, with label l iff there is at least one ψ_i in the list with an edge, $(v \xrightarrow{l} w)$ where $\lambda(v) = r$ (step 5). Finally, note that the set of construction vertices, $\mathcal{A}(s)$, which are alternation reachable from the target, s, of a construction edge, $(r \xrightarrow{l} s) \in EC_\phi$, is the union of all vertices, $\lambda(w)$, where $(v \xrightarrow{l} w) \in E_{\psi_i}$, $1 \leq i \leq n$, and $\lambda(v) = r$ (steps 3,5,6).

Let ϕ' be a class dictionary graph such that $Objects(\phi) \supset Objects(\phi') \supseteq \{\psi_1, \psi_2, ...\psi_n\}$, and let $\psi \in Objects(\phi)$ but $\psi \notin Objects(\phi')$. Since ψ is not legal with respect to ϕ', one of the following conditions must hold (by definition 2.8.):

A. $\exists v \in W_\psi$ such that $\lambda(v) \notin VC_{\phi'}$

B. $\exists v \in W_\psi, (v \xrightarrow{l} w) \in E_\psi$ such that $\forall (r \xrightarrow{l} s) \in EC_{\phi'} : r \stackrel{*}{\not\Longrightarrow} \lambda(v)$ or $s \stackrel{*}{\not\Longrightarrow} \lambda(w)$

C. $\exists v \in W_\psi, (r \xrightarrow{l} s) \in EC_{\phi'}$ such that $r \stackrel{\star}{\Longrightarrow} \lambda(v)$ and $\forall w \in W_\psi : (v \xrightarrow{l} w) \notin E_\psi$.

[2]It follows directly from definition 2.8. that each ψ_i, $1 \leq i \leq n$, is included in $Objects(\phi)$.

Case A. Since ψ is legal with respect to ϕ, $\lambda(v) \in VC_\phi$, so there must be some ψ_i, $1 \leq i \leq n$, which contains a vertex, v' such that $\lambda(v') = \lambda(v) \notin VC_{\phi'}$. But ψ_i is legal with respect to ϕ', so $\lambda(v') \in VC_{\phi'}$, a contradiction.

Case B. Since ψ is legal with respect to ϕ, $\exists(r \xrightarrow{l} s) \in EC_\phi$ such that $r = \lambda(v)$ and $s \stackrel{\star}{\Longrightarrow} \lambda(w)$. Therefore, there must be some ψ_i, $1 \leq i \leq n$, such that $(v' \xrightarrow{l} w') \in E_{\psi_i}$, $\lambda(v') = \lambda(v)$, $\lambda(w') = \lambda(w)$. But ψ_i is legal with respect to ϕ', so $\exists(r \xrightarrow{l} s) \in EC_{\phi'}$ such that $r \stackrel{\star}{\Longrightarrow} \lambda(v') = \lambda(v)$ and $s \stackrel{\star}{\Longrightarrow} \lambda(w') = \lambda(w)$, a contradiction.

Case C. Since ψ is legal with respect to ϕ, there must exist some ψ_i, $1 \leq i \leq n$, which contains a vertex, $v' \in W_{\psi_i}$, such that $\lambda(v) = \lambda(v')$. Since ψ_i and ψ are both legal with respect to ϕ:

- $\forall w', l$ where $(v' \xrightarrow{l} w') \in E_{\psi_i} : \exists(r \xrightarrow{l} s) \in EC_\phi$ such that $r = \lambda(v') = \lambda(v)$.
- $\forall(r \xrightarrow{l} s) \in EC_\phi$ where $r \stackrel{\star}{\Longrightarrow} \lambda(v) : \exists w \in W_\psi$ such that $(v \xrightarrow{l} w) \in E_\psi$.

Therefore:

- $\forall w', l$ where $(v' \xrightarrow{l} w') \in E_{\psi_i} : \exists w \in W_\psi$ such that $(v \xrightarrow{l} w) \in E_\psi$.

But ψ_i is also legal with respect to ϕ', so $\forall(r \xrightarrow{l} s) \in EC_{\phi'}$ where $r \stackrel{\star}{\Longrightarrow} \lambda(v') : \exists w' \in W_{\psi_i}$ such that $(v' \xrightarrow{l} w') \in E_{\psi_i}$ and therefore $\exists w \in W_\psi$ such that $(v \xrightarrow{l} w) \in E_\psi$, a contradiction.

3.4 Incremental Learning

Given a class dictionary graph, ϕ, and an object example graph, ψ, the incremental learning algorithm will learn a class dictionary graph, ϕ', such that the set of objects defined by ϕ' includes ψ and all of the objects defined by ϕ. Furthermore, the algorithm insures that the set of objects defined by ϕ' is a subset of the objects defined by any class dictionary graph that includes ψ and all of the objects defined by ϕ. Intuitively, we extend the set of objects defined by ϕ only enough to include objects "similar" to ψ.

Formally, given a class dictionary graph, ϕ_1, and an object example graph, ψ, we learn a legal class dictionary graph, ϕ_2, such that $Objects(\phi_2) \supseteq Objects(\phi_1) \cup \psi$, and for all legal class dictionary graphs, ϕ_3 where $Objects(\phi_3) \supseteq Objects(\phi_1) \cup \psi : Objects(\phi_2) \subseteq Objects(\phi_3)$.

If there is no legal class dictionary graph that defines a set of objects that includes ψ and all of the objects defined by ϕ, we say that the object example graph ψ is not incrementally legal with respect to ϕ.

Definition 3.2. *An object example graph ψ is* **incrementally legal** *with respect to a class dictionary graph ϕ if there exists a legal class dictionary ϕ' such that $Objects(\phi') \supseteq Objects(\phi) \cup \psi$.*

If a list of object example graphs $\psi_1, ..., \psi_n$ is legal, then each ψ_i in the list must be incrementally legal with respect to the class dictionary graph learned from $\psi_1, ..., \psi_{i-1}$. Therefore a class dictionary graph can be learned incrementally from a legal list of object example graphs.

Denote the intermediate class dictionary learned from $\psi_1, \psi_2, ..., \psi_m$ by ϕ_m, and let $\phi_0 = (\emptyset, \emptyset; \emptyset, \emptyset)$. Then ϕ_m is learned from ϕ_{m-1} and ψ_m, where $1 \leq m \leq n$, as follows:

1. $\Lambda = \Lambda_{\phi_{m-1}} \cup \Lambda_{\psi_m}$

   For each edge in the object example graph there is a construction edge in the class dictionary graph with the same label.

2. $VC = VC_{\phi_{m-1}} \cup \{r \mid \exists v \in W_{\psi_m} : \lambda_{\psi_m}(v) = r\}$

   We interpret λ as a function that maps objects to their classes. For each new class that appears in the object example graph, we add a construction class which is represented as a construction vertex in the class dictionary graph.

3. $VA = VA_{\phi_{m-1}}$
   $\cup \{(r, l) \mid r \in VC, l \in \Lambda, \exists v1, v2, w1, w2 \in W_{\psi_m} :$
   $\lambda_{\psi_m}(v1) = \lambda_{\psi_m}(v2) = r, \lambda_{\psi_m}(w1) \neq \lambda_{\psi_m}(w2),$
   $(v1 \xrightarrow{l} w1), (v2 \xrightarrow{l} w2) \in E_{\psi_m}\}$
   $\cup \{(r, l) \mid r \in VC, l \in \Lambda, \exists v, w \in W_{\psi_m}, s \in VC :$
   $\lambda_{\psi_m}(v) = r, \lambda_{\psi_m}(w) \neq s, (v \xrightarrow{l} w) \in E_{\psi_m}, (r \xrightarrow{l} s) \in EC_{\phi_{m-1}}\}$

   The first term represents the alternation vertices already learned in ϕ_{m-1}. The second term adds the alternations we learn from ψ_m alone (this is the same term as in the Basic Algorithm, where $\psi_i = \psi_j = \psi_m$). The last term adds alternations that are learned in the Basic Algorithm when $\psi_i \neq \psi_j$. In the case of incremental learning we rely on the fact that the edges of ψ_1, ..., ψ_{m-1} are recorded in ϕ_{m-1} as construction edges.

4. $V = VC \cup VA$

   The vertices of the class dictionary graph are given by the union of the construction vertices and alternation vertices.

5. $EC = (EC_{\phi_{m-1}} - \{(r \xrightarrow{l} s) \mid (r,l) \in (VA - VA_{\phi_{m-1}})\})$
   $\cup \{(r \xrightarrow{l} (r,l)) \mid (r,l) \in (VA - VA_{\phi_{m-1}})\}$
   $\cup \{(r \xrightarrow{l} s) \mid r,s \in V, \exists v,w \in W_{\psi_m} :$
   $\lambda_{\psi_m}(v) = r, \lambda_{\psi_m}(w) = s, (v \xrightarrow{l} w) \in E_{\psi_m}, (r,l) \notin VA\}$

   We start with the construction edges in ϕ_{m-1}, but if we learned a new abstract class, represented by (r,l), we remove any construction edges to vertices representing subclasses of the new abstract class (first term) and replace them with construction edges to (r,l) (second term). Finally, the third term adds new construction edges learned from ψ_m.

6. $EA = EA_{\phi_{m-1}}$
   $\cup \{((r,l) \Longrightarrow s) \mid (r,l) \in VA, s \in V, \exists v,w \in W_{\psi_m} :$
   $\lambda_{\psi_m}(v) = r, \lambda_{\psi_m}(w) = s, (v \xrightarrow{l} w) \in E_{\psi_m}\}$
   $\cup \{((r,l) \Longrightarrow s) \mid (r,l) \in VA, s \in V, (r \xrightarrow{l} s) \in EC_{\phi_{m-1}}\}$

   Here we start with the alternation edges from the previous class dictionary graph and add edges learned from ψ_m alone, and from ψ_m and ϕ_{m-1}. The three terms correspond to the three terms used to learn the alternation vertices in step 3.

The following theorem can be easily proven by induction on the length of the object example graph list:

Theorem 3.1. *A class dictionary graph learned incrementally is identical to the class dictionary graph learned using the basic learning algorithm.*

3.5 Optimum class dictionary graph learning

The learning algorithms presented in sections 3.2 and 3.4 produce class dictionary graphs that are correct but not optimal. There are two major problems with the learned class dictionary graphs. The first problem is that alternation vertices are never reused. For each part that can be instantiated by objects of more than one construction class, a new alternation vertex is generated that has each of the construction classes as immediate successors

in the inheritance hierarchy. In other words, each alternation vertex has only one incoming construction edge and no incoming alternation edges. The result is a class dictionary graph with too many alternation vertices, too many alternation edges, and an inheritance hierarchy with great deal of unnecessary multiple-inheritance.

The second problem is that parts are never inherited; instead, they are attached directly to each construction class. In other words, none of the alternation vertices have any outgoing construction edges. The result is a class dictionary graph with too many construction edges.

Class dictionary graph optimization is discussed formally in Chapter 5 and algorithms for optimizing the class dictionary graphs produced by the learning algorithms are developed. In this section, we discuss informally how the incremental learning algorithm can be extended to learn optimal single-inheritance class dictionary graphs. Consideration of the optimization problem leads to some important observations regarding class dictionary design.

Informally, we say that a class dictionary graph is in common normal form (CNF) if it has no redundant parts. If two different vertices, v and v', in a class dictionary graph have outgoing construction edges with the same label, l, and the same target, w, then we say that the part, l, is redundant in classes v and v'. We observe that we can always avoid redundant parts by introducing additional inheritance. That is, we only need to define the part l once in a common superclass of v and v'. It may be necessary to add a new abstract class if v and v' do not already have a suitable common superclass, and the addition may cause the introduction of multiple inheritance.

Sometimes, we can avoid multiple inheritance by introducing redundant parts, but other times we can not eliminate multiple inheritance while maintaining object equivalence. When faced with a choice, multiple inheritance will generally produce the better class dictionary graph since redundant parts hide the commonalities between classes and often lead to poor software organization and duplication of code.

In Chapter 5 an efficient algorithm is presented for abstracting optimum single-inheritance class dictionary graphs from class dictionary graphs learned using the basic learning algorithm (section 2). It is shown that a class dictionary graph with no redundant parts (i.e., it is in class dictionary common normal form, or CNF), no unnecessary alternation vertices, and with a single-inheritance hierarchy is guaranteed optimal[3]. An alternation vertex is *unnecessary* if it is a singleton (i.e. has only one outgoing alternation edge) or if it has no

[3]according to the metrics introduced in Chapter 5

incoming or outgoing construction edges.

Therefore, an incremental learning algorithm will produce an optimum single-inheritance class dictionary graph if with each new example the algorithm maintains a class dictionary graph that has a single-inheritance hierarchy with no unnecessary alternation vertices and no redundant parts. We define the Incremental Single-Inheritance Optimum Class Dictionary Learning problem as follows:

Instance:

An optimum single-inheritance class dictionary graph, ϕ, and an object example graph, ψ, where ψ is incrementally legal with respect to ϕ.

Problem:

Find an optimum single-inheritance class dictionary graph, ϕ', such that $Objects(\phi') \supseteq (Objects(\phi) \cup \psi)$.

It is always possible to avoid unnecessary alternation edges. If an alternation vertex, v, has no incoming or outgoing construction edges it can be deleted after transferring its outgoing alternation edges to its predecessor in the inheritance hierarchy. If there is no predecessor, the outgoing alternation edges can just be deleted. A singleton can be similarly removed after first transferring any incoming or outgoing construction edges to its successor in the inheritance hierarchy.

Clearly, however, it is not always possible to maintain a single-inheritance hierarchy, particularly with the added constraint that no redundant parts are introduced. After eliminating unnecessary alternation vertices, each remaining alternation vertex, v, can be thought of as representing a set of construction classes, $\mathcal{A}(v)$. If the alternation vertex has an incoming construction edge, it represents the set of classes that can be used to instantiate a part. If it has an outgoing construction edge, it represents the set of classes that share a common part. In an optimal class dictionary graph, ϕ, these sets of construction classes are called the $ConstructionClusters(\phi)$. If the vertices of ϕ are arranged in a single-inheritance hierarchy, there must only be one vertex for each element of $ConstructionClusters(\phi)$ and each pair of elements must either be disjoint or in a superset/subset relation.

Consider an instance of the Incremental Single-Inheritance Optimum Class Dictionary Learning problem with class dictionary graph, ϕ and object example graph, ψ and assume

there is a solution, ϕ'. Since ϕ is optimal, we can easily compute $ConstructionClusters(\phi)$. Next, consider the changes that will need to be made in $ConstructionClusters(\phi)$ when we learn the new objects in ψ. Some of the sets in $ConstructionClusters(\phi)$ need to be expanded when we learn a new possibility for the kind of object that can instantiate a part. We will also need to expand (or add) sets when we learn that there are additional classes that share a common part. The Incremental Single-Inheritance Optimum Class Dictionary Learning problem has a solution only if the conditions for a single-inheritance hierarchy still hold after making the necessary changes to $ConstructionClusters(\phi)$. That is, there is a solution if each pair of elements is still either disjoint or in a superset/subset relation.

It is easy to see how the incremental learning algorithm presented in Section 3.4 can be extended to produce optimum single-inheritance class dictionaries. Only one alternation vertex is created for each set of construction classes that needs to be represented. Alternation vertices are created not only to express the different possibilities for instantiating a part, but also to implement inheritance of parts common to more than one class. Each new alternation vertex is inserted into the inheritance-hierarchy according to the order imposed by the superset/subset relations.

3.6 Extending a class dictionary graph based on object examples

The algorithms presented in sections 3.2 and 3.4 are useful for object-oriented design, especially when used in combination with the optimization techniques of Section 3.5 and Chapter 5. However, even the incremental learning algorithm is not ideal for use in the context of schema evolution. The algorithm can learn new classes and can learn an expanded set of classes that can instantiate a part, but it cannot learn new parts for existing classes.

If an object example graph, $\psi = (W, S, \Lambda_\psi;\ E, \lambda)$, contains an edge $(v \xrightarrow{l} w) \in E$, and the existing class dictionary graph contains vertex $v' = \lambda(v)$ but v' does not already have an l part, then ψ is not incrementally legal with respect to ϕ since there is no class dictionary graph, ϕ', such that $Objects(\phi') \supseteq Objects(\phi)$.

One solution is to expand the data model to allow classes to have *optional* parts. That is, different instances of the same class may be allowed to differ in their number of attributes. This is the approach that we took in implementing the Demeter case tool. With the addition

of optional parts to the data model, every object example graph is incrementally legal with respect to every class dictionary graph. If the object example graph, $\psi = (W, S, \Lambda_\psi;\ E, \lambda)$, contains an edge $(v \xrightarrow{l} w) \in E$, and the class dictionary graph, $\phi = (\ VC,\ VA, \Lambda;\ EC, EA)$, contains vertex $v' = \lambda(v) \in VC$ but v' does not already have an l part, then an *optional* l part is added. If ϕ contains an edge, $(v' \xrightarrow{l} w')$, and the object example graph contains a vertex, v, such that $v' \stackrel{*}{\Rightarrow} \lambda(v)$ but there is no outgoing edge from v with label l in ψ, then the l part is made optional in ϕ.

A second solution is to simply drop the legality requirement and reformulate the learning problem. That is the approach taken in this section. The **incremental evolution** problem is formulated as follows:

Instance:

A legal class dictionary graph, $\phi = (\ VC,\ VA, \Lambda;\ EC, EA)$, and a legal object example graph, $\psi = (W, S, \Lambda_\psi;\ E, \lambda)$.

Problem:

Find a class dictionary graph, ϕ', such that every element of $Objects(\phi) \cup \psi$ is a subgraph of an element of $Objects(\phi')$ and every element of $Objects(\phi')$ is *similar* to the objects $Objects(\phi) \cup \psi$.

Definition 3.3. *An object example graph, $\psi = (W, S, \Lambda_\psi;\ E, \lambda)$, is* **similar** *to a set of object example graphs, $O = \{\psi_1, \psi_2, ...\psi_n\}$, if for each edge $(v \xrightarrow{l} w) \in E$ there exists an element of O, ψ_i, such that: $(v' \xrightarrow{l} w') \in E_i$, $\lambda(v) = \lambda_i(v')$, $\lambda(w) = \lambda_i(w')$.*

Intuitively, we add only those classes and parts that are warranted by the object example graph. Unlike the incremental learning problem, the instance is not constrained by an incremental legality requirement. We accept any legal class dictionary graph and any legal object example graph as input.

In this approach we change our interpretation of object example graphs. The graphs are considered examples of *partial* objects. That is, the examples need not show all of the parts of the objects. The incremental evolution algorithm is identical to the incremental learning algorithm given in Section 3.4. Only the interpretation changes.

Example 3.3. *Consider the class dictionary graph and object example graph shown in Figure 17. The* `Student` *object is not legal with respect to the class dictionary graph for two reasons: It has an* `advisor` *part and is missing the* `gpa` *part. When the incremental*

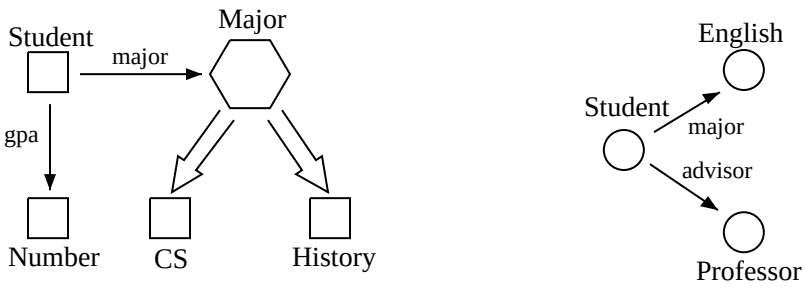


Figure 17: Class dictionary graph and object graph for incremental evolution

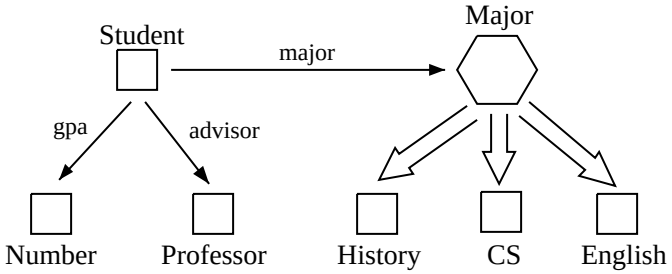


Figure 18: Class dictionary graph after incremental evolution

learning algorithm is applied the missing `gpa` *part is ignored, the* `advisor` *part is added to the* `Student` *definition, and the* `major` *part is extended as expected. The new class dictionary graph shown in Figure 18 is exactly what is desired for incremental evolution.*

In Chapter 6 we define a complete set of primitive transformations that are useful for extending a class dictionary graph.

3.7 Training set

Class dictionary graph learning requires only a small easily generated training set. To learn a class dictionary graph that defines the same objects as a given class dictionary graph, ϕ, we need to see examples for each construction class with all possible parts. But since we can vary the parts simultaneously, we need at most $|\ VC|$ examples for each construction class, and $|\ VC|^2$ examples to learn the whole class dictionary graph.

Example 3.4. *Consider the class dictionary graph in Figure 19. Class* `A` *has three parts and there are three alternatives for each part, so there are a total of 27 different combinations*

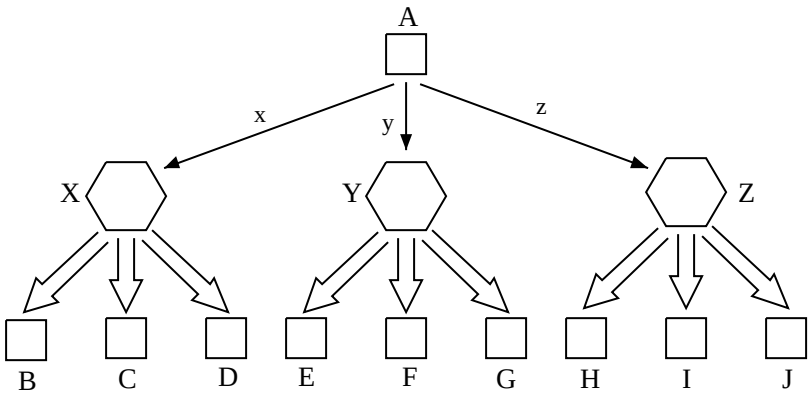


Figure 19: A class dictionary graph defining 27 different `A` objects

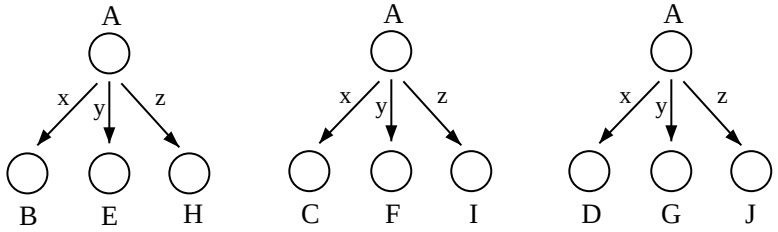


Figure 20: The training set

that could occur in object graphs. Even so, the class dictionary graph can be learned from just the three examples shown in Figure 20 where the parts are varied simultaneously.

3.8 Related work

The problem of learning classes from object examples has been studied earlier in AI (e.g., [SM86], [CF82],). Clustering algorithms have been applied to build a tree of mutually exclusive classes from a given object set. Our work extends this earlier work since we have more structure in our classes, e.g., the capability to define a language. Related work has been done in the area of learning context-free grammars from examples and syntax trees [AS83].

Winston's work [Win70] is concerned with learning visual concepts in a world of 3-dimensional structures comprised of bricks, wedges, and other simple objects. A scene is represented by a semantic net with relations such as has-part, supported-by, in-front-of, a-kind-of, has-property-of, etc.

There are several ways in which objects can be grouped. The most relevant to our work are by common properties and by identification with a known model. The following example serves to illustrate the differences.

Example 3.5. *Consider the world with objects A, B, C, X, Y, and Z with the following properties:*

```
A: has-part X, has-part Y, has-part Z
B: has-part X, has-part Y
C: has-part X, has-part Z
```

In our notation:

```
A(<x> X  <y> Y  <z> Z)
B(<x> X  <y> Y)
C(<x> X         <z> Z)
```

A, B, and C are candidates for a group since they all have the same (has-part) relationship to X.

In Winston's work, a program would learn the new object:

```
A&B(<a> A <b> B)
```

In our system, we learn (after optimization) three new abstract classes:

```
AorB_or_AorC : AorB | AorC *common* <x> X.
AorB : A | B *common* <y> Y.
AorC : A | C *common* <z> Z.
```

and remove all the parts from classes A, B, and C.

Another major difference is that Winston's work deals with properties that describe the relationships between objects other than "part-of" relations. That is, where we might learn an abstract class based on the information that two objects share *parts* "length" and "width", Winston would be concerned with whether or not two objects had the same values for their *properties* "length" and "width".

Since Winston needs to learn relations other than part-of, his system is necessarily much more complex than ours. (Winston presents a lot of ideas about learning, but no formal

algorithms.) Another complicating factor is that Winston wants a system that, given some examples of a type of object (class), builds a model that can recognize objects of that type even if they have properties that are different from any of the examples.

For example, given an example of an "arch" that has two uprights supporting a brick, and a second example of an arch that has two uprights supporting a wedge, the system should recognize an object consisting of two uprights supporting some other type of object as an arch.

In [LM91] several ways in which conceptual database evolution can occur through learning are discussed. One of these, the expansion of a type into subtypes, is similar to the introduction of alternation vertices which occurs during the basic learning phase of our algorithm. Another, the generalization of types to form supertypes, is a special case of our technique for removing redundant parts discussed in Chapter 5.

A major difference in our work is that we focus on learning from *examples*, while in [LM91] the emphasis is on learning from observation of *instances* (e.g., noticing that some of the instances of a type object have null values for a given attribute). Our examples are more general than instances since we don't supply values for attributes.

Chapter 4

Object preserving transformations

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 often 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 chapter 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.

4.1 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**.

Definition 4.1. *Two class dictionary graphs, ϕ_1 and ϕ_2, are* **object-equivalent** *if the set of legal object graphs with respect to ϕ_1 is the same as the set of legal object graphs with respect to ϕ_2; that is, $Objects(\phi_1) = Objects(\phi_2)$.*

The following theorem provides a convenient means for mechanically checking two class dictionary graphs for object-equivalence:

Theorem 4.1. *Given a class dictionary graph $\phi = (VC, VA, \Lambda;\ EC, EA)$, for $v \in V$ let $PartClusters_\phi(v) = \{(l, \mathcal{A}(w)) \mid \exists v' : v' \stackrel{\star}{\Longrightarrow} v$ and $(v', w, l) \in EC\}$.*

$$where\ \mathcal{A}(w) = \{w' | w \stackrel{\star}{\Longrightarrow} w' \text{ and } w' \in VC\}$$

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

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

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.

The proof of theorem 4.1. is straightforward:

1. The conditions of theorem 4.1. are necessary.

   - Let ϕ_1 and ϕ_2 be class dictionary graphs such that $VC_{\phi_1} \neq VC_{\phi_2}$. Now construct an object graph, $\psi = (W, VC_{\phi_1}, \Lambda;\ E, \lambda)$, as follows. For each vertex, $v \in VC_{\phi_1}$, we place a corresponding vertex, v' in W, and map v' to v by adding (v', v) to λ. For each construction edge, $(v \stackrel{l}{\longrightarrow} w)$ in ϕ_1, add an outgoing construction edge $(v' \stackrel{l}{\longrightarrow} w')$ from each vertex, v' in ψ where $v \stackrel{\star}{\Longrightarrow} \lambda(v')$ to some vertex w' such that $w \stackrel{\star}{\Longrightarrow} \lambda(w')$. The resulting object graph, ψ, is legal with respect to ϕ_1 but not with respect to ϕ_2, so the class dictionary graphs are not object-equivalent.
   - Let ϕ_1 and ϕ_2 be class dictionary graphs such that $VC_{\phi_1} = VC_{\phi_2}$, but for some $v \in VC$, there exists (l, S) such that $(l, S) \in PartClusters_{\phi_1}(v)$ but $(l, S) \notin PartClusters_{\phi_2}(v)$. Now construct an object graph as before, but when adding

outgoing edge, $(v' \xrightarrow{l} w')$, to the vertex, v', corresponding to v (under λ) choose w' such that there is no $(l, S) \in PartClusters_{\phi_2}(v)$ where $\lambda(w') \in S$. The resulting object graph, ψ, is legal with respect to ϕ_1 but not with respect to ϕ_2, so the class dictionary graphs are not object-equivalent.

2. The conditions of theorem 4.1. are sufficient.

   - Let ϕ_1 and ϕ_2 be class dictionary graphs such that $VC_{\phi_1} = VC_{\phi_2}$ and for all vertices, $v \in VC$, $PartClusters_{\phi_1}(v) = PartClusters_{\phi_2}(v)$. Now assume that there exists an object graph, $\psi = (W, S, \Lambda_\psi;\ E, \lambda)$, such that ψ is legal with respect to ϕ_1 but not with respect to ϕ_2.

     By the definition of legality for object graphs, each vertex, v in ψ, has one outgoing edge, $(v \xrightarrow{l} w)$ for each corresponding construction edge, $(v' \xrightarrow{l} w')$ in ϕ_1, where $v' \stackrel{*}{\Longrightarrow} \lambda(v)$, such that $w' \stackrel{*}{\Longrightarrow} \lambda(w)$. There can be no other outgoing edges from v.

     But there must be some vertex, v, in ψ which either has an outgoing edge, $(v \xrightarrow{l} w)$, with no corresponding edge in ϕ_2, or else has no outgoing edge corresponding to a construction edge in ϕ_2. In either case, $PartClusters_{\phi_1}(v) \neq PartClusters_{\phi_2}(v)$, a contradiction.

Example 4.1.

The two class dictionary graphs in figures 21 and 22, ϕ_1 and ϕ_2, are object-equivalent since:

$VC_{\phi_1} = VC_{\phi_2}$

$= \{$`Undergrad, Grad, Prof, TA, Admin_asst, Coach, Num, Real_Num`$\}$

$PartClusters_{\phi_1}$(`Undergrad`)

$= PartClusters_{\phi_2}$(`Undergrad`)

$= \{$`(ssn, {Num}), (gpa, {Real_Num})`$\}$

$PartClusters_{\phi_1}$(`Grad`)

$= PartClusters_{\phi_2}$(`Grad`)

$= \{$`(ssn, {Num}), (gpa, {Real_Num})`$\}$

$PartClusters_{\phi_1}$(`TA`)

$= PartClusters_{\phi_2}$(`TA`)

$= \{$`(ssn, {Num}), (salary, {Real_Num}), (assigned, {Course, Committee})`$\}$

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

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

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

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

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

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

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

One final formulation of the object-equivalence criteria is provided by theorem 4.2..

Theorem 4.2. *Two class dictionary graphs, ϕ_1 and ϕ_2, are* **object-equivalent** *iff:*

- $VC_{\phi_1} = VC_{\phi_2}$
- $\forall v, w \in VC$:
  $\left(\exists(v_1 \xrightarrow{l} w_1) \in EC_{\phi_1} : v_1 \stackrel{\star}{\Longrightarrow} v, w_1 \stackrel{\star}{\Longrightarrow} w\right)$
  $\iff \left(\exists(v_2 \xrightarrow{l} w_2) \in EC_{\phi_2} : v_2 \stackrel{\star}{\Longrightarrow} v, w_2 \stackrel{\star}{\Longrightarrow} w\right)$
  That is, v can have w as an l part in ϕ_1 iff v can have w as an l part in ϕ_2.

The second condition in theorem 4.2. is equivalent to the second condition in theorem 4.1., and the proof is essentially the same.

Now we can formally define object-preserving class dictionary graph transformations.

Definition 4.2. *A class dictionary graph* **transformation**, *T, is a rule which defines an allowable modification of class dictionary graphs. Let*

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

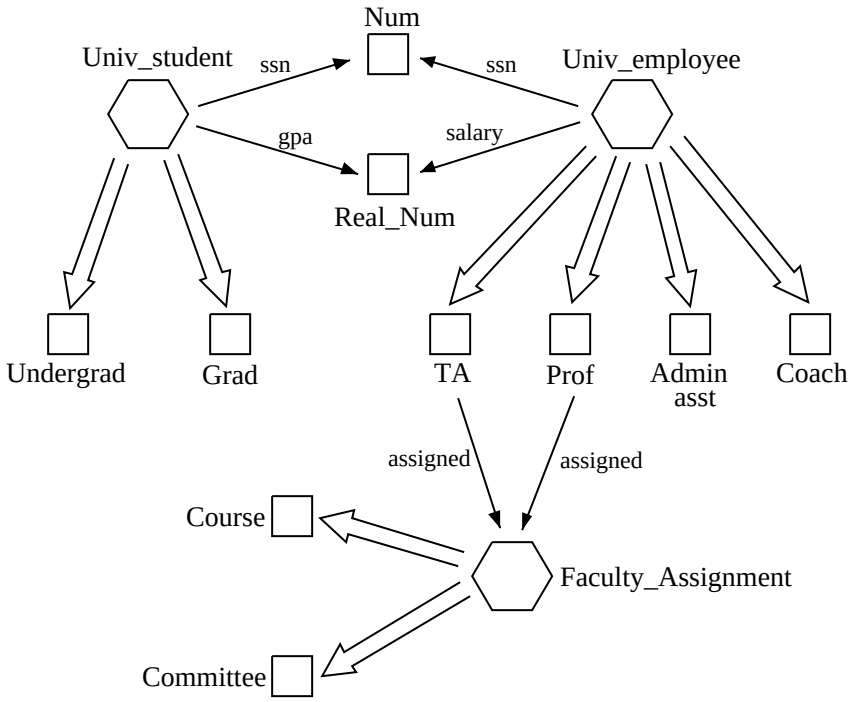


Figure 21: Class Dictionary Graph ϕ_1

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

4.2 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 its outgoing alternation edges.

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

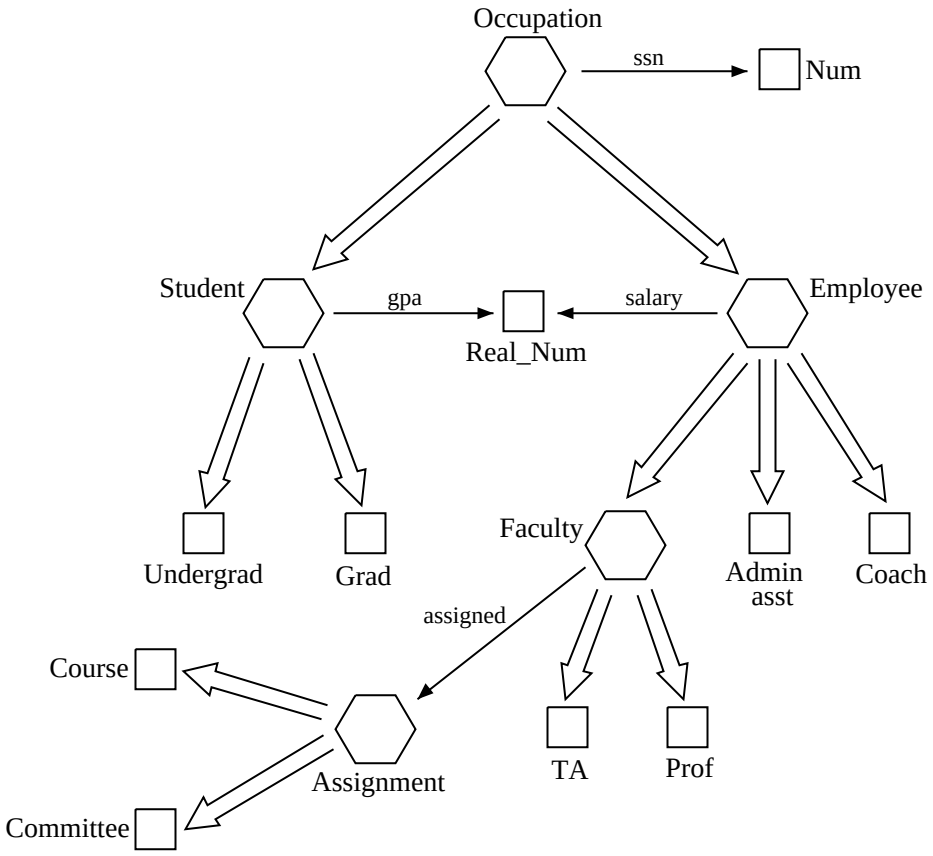


Figure 22: Class Dictionary Graph ϕ_2

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

   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, $(v \xrightarrow{l} w)$ can be deleted from an alternation vertex, v, if for each $(v \Longrightarrow v') \in EA$ a new construction edge, $(v' \xrightarrow{l} w)$, 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 $(w \xrightarrow{l} v) \in EC$ can be deleted and replaced with a new construction edge, $(w \xrightarrow{l} v')$.

   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. Note that the inverse of part replacement is just another instance of the transformation.

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.

4.3 Proofs

4.3.1 Correctness

Each primitive operation preserves object-equivalence. This fact follows directly from theorem 4.1. ($PartClusters$).

4.3.2 Completeness

Given two arbitrary object-equivalent class dictionary graphs, $\phi_1 = (VC, VA_1, \Lambda_1; EC_1, EA_1)$ and $\phi_2 = (VC, VA_2, \Lambda_2; EC_2, EA_2)$, it is possible to transform ϕ_1 to ϕ_2 using only primitive operations. Before transforming ϕ_1, we record the original graph for reference; i.e., let $\phi_0 = (VC, VA_0, \Lambda_0; EC_0, EA_0) = \phi_1$. Now ϕ_1 is transformed to ϕ_2 as follows:

1. Assume, without loss of generality, that the sets of alternation vertices of ϕ_1 and ϕ_2 are disjoint; that is, $VA_1 \cap VA_2 = \emptyset$. Let $T = v_1, v_2, ...v_n$ be a topological sorting of the alternation subgraph of ϕ_2, such that if $v_i \stackrel{\star}{\Longrightarrow} v_j$ in ϕ_2 then $i > j$.

   For $j \leftarrow 1$ to n:

   - If $v_j \in VA_2$ then $VA_1 \leftarrow VA_1 \cup \{v_j\}$
   - $EA_1 \leftarrow EA_1 \cup \{(v_j \Longrightarrow w) | (v_j \Longrightarrow w) \in EA_2\}$
     by addition of useless alternation.

   Since w is either an alternation vertex or a construction vertex that has an incoming alternation edge, w must be included in T. Furthermore since $v_j \stackrel{\star}{\Longrightarrow} w$, $w = v_i$ for some $i < j$. Therefore w must already be a vertex in ϕ_1.

   Now $EA_1 = EA_0 \cup EA_2$ and $VA_1 = VA_0 \cup VA_2$.

2. While $\exists (v \stackrel{l}{\longrightarrow} w) \in EC_1$ such that $v \in VA_1$:

   - Select one such edge, $(v \stackrel{l}{\longrightarrow} w)$. Delete that edge and for every vertex v' such that $(v \Longrightarrow v') \in EA_1$ add an edge $(v' \stackrel{l}{\longrightarrow} w)$, by distribution of common parts.

3. Consider each construction edge, $(v_1 \stackrel{l}{\longrightarrow} w_1) \in EC_1$ such that $w_1 \in VA_1$. Note that $v_1 \in VC$ since all common parts have been distributed. There must be an edge $(v_2 \stackrel{l}{\longrightarrow} w_2) \in EC_2$ such that $v_2 \stackrel{\star}{\Longrightarrow} v_1$ and $\mathcal{A}_{\phi_1}(w_1) = \mathcal{A}_{\phi_2}(w_2)$ since $PartClusters_{\phi_1}(v_1) = PartClusters_{\phi_2}(v_1)$. Furthermore, $\mathcal{A}_{\phi_1}(w_2) = \mathcal{A}_{\phi_2}(w_2)$ after step 1. Replace each $(v_1 \stackrel{l}{\longrightarrow} w_1)$ with $(v_1 \stackrel{l}{\longrightarrow} w_2)$ in EC_1 by part replacement.

4. Consider the construction edges that would be obtained if distribution of common parts was applied exhaustively to ϕ_2. They are exactly the construction edges that we now have in ϕ_1 after distribution of common parts and part replacement. Since each step in the sequence of distribution of common parts operations is reversible by an abstraction of common parts operation, there is a sequence of abstraction of common

part operations to transform EC_1 to EC_2. Abstraction of common parts is applied to move construction edges up the inheritance hierarchy until they are attached to the same vertices as in ϕ_2.

Now we have $EC_1 = EC_2$, $EA_1 = EA_0 \cup EA_2$, and $VA_1 = VA_0 \cup VA_2$. All that remains is to remove the original alternation vertices, VA_0, and alternation edges, EA_0.

5. After steps 2 and 3, there are no construction edges (either incoming or outgoing) attached to any of the original alternation vertices; that is, $\{(v \xrightarrow{l} w) | v \in VA_0 \text{ or } w \in VA_0\} = \emptyset$. Also, there are no cycles in the alternation subgraph of ϕ_1, and ϕ_1 has no alternation edges from the new alternation vertices (VA_2) to the original alternation vertices (VA_0). Therefore, if $(VA_0 \cap VA_1) \neq \emptyset$, then at least one of the elements of $VA_0 \cap VA_1$ must be "useless". The "useless" alternation vertex is deleted from ϕ_1 along with its outgoing alternation edges. This step is repeated until $(VA_0 \cap VA_1) = \emptyset$.

   Now $VA_1 = VA_2$. But since outgoing alternation edges were deleted along with the useless alternation vertices and every alternation edge in EA_0 had a vertex in VA_0 as its source, all the edges in EA_0 are deleted from ϕ_1 and $EA_1 = EA_2$, so $\phi_1 = \phi_2$.

In summary, a class dictionary graph, ϕ_1 can be transformed to an arbitrary object-equivalent class dictionary graph, ϕ_2 as follows:

1. Use addition of useless alternation to "superimpose" the alternation subgraph of ϕ_2 onto ϕ_1.

2. Exhaustively apply distribution of common parts until all outgoing construction edges have been removed from the original alternation vertices in ϕ_1 and are attached directly to construction vertices.

3. Use part replacement to move any construction edge with an "old" alternation vertex as its target so that its target corresponds to the proper vertex in ϕ_2.

4. Use 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.

5. Use deletion of useless alternation to remove the "old" alternation subgraph from ϕ_1.

4.3.3 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 4.2. *This example illustrates the construction of the completeness proof with the class dictionary graphs of figures 21 and 22. 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 23.

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 24. In a deeper inheritance hierarchy some parts might need to be distributed repeatedly until they are attached directly to construction classes.

Part Replacement. The "old" alternation vertex `Faculty_Assignment` still has incoming construction edges from the new construction vertices `TA` and `Prof`. In ϕ_2 the corresponding

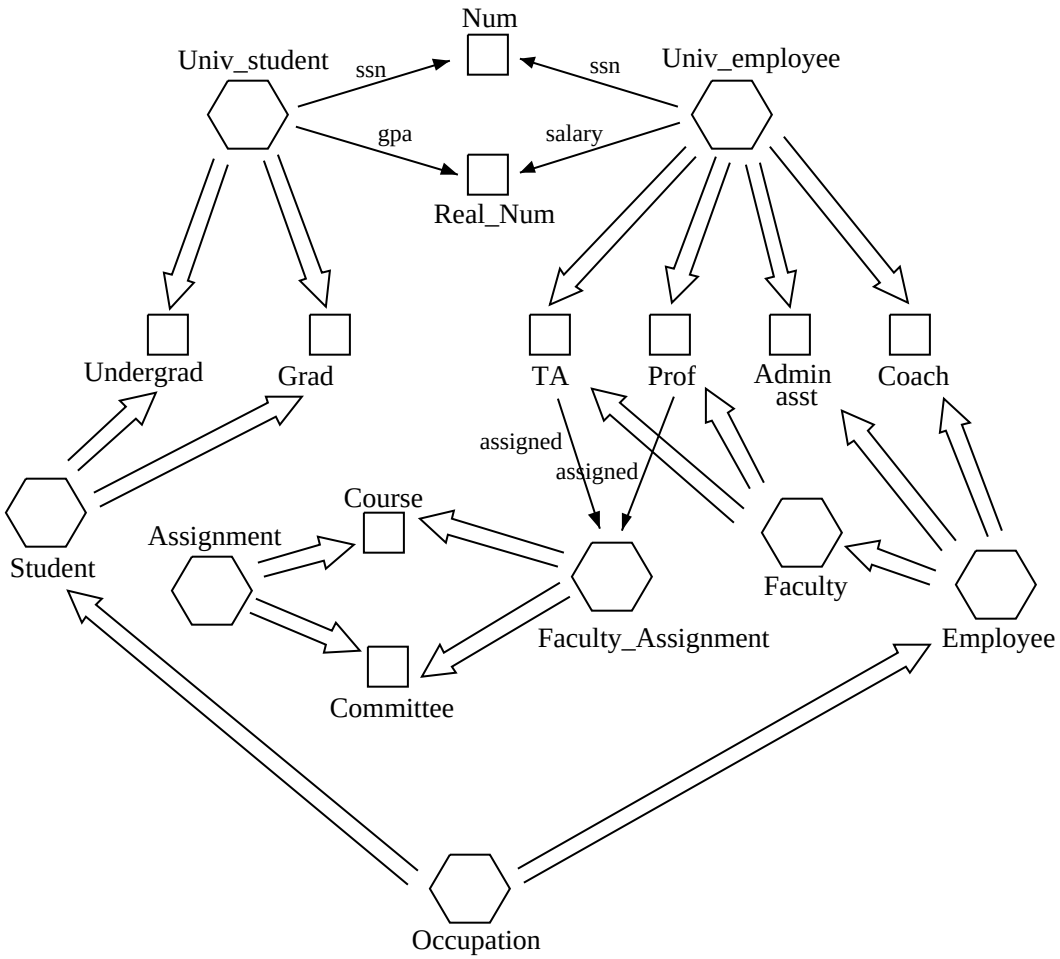


Figure 23: Addition of Useless Alternations

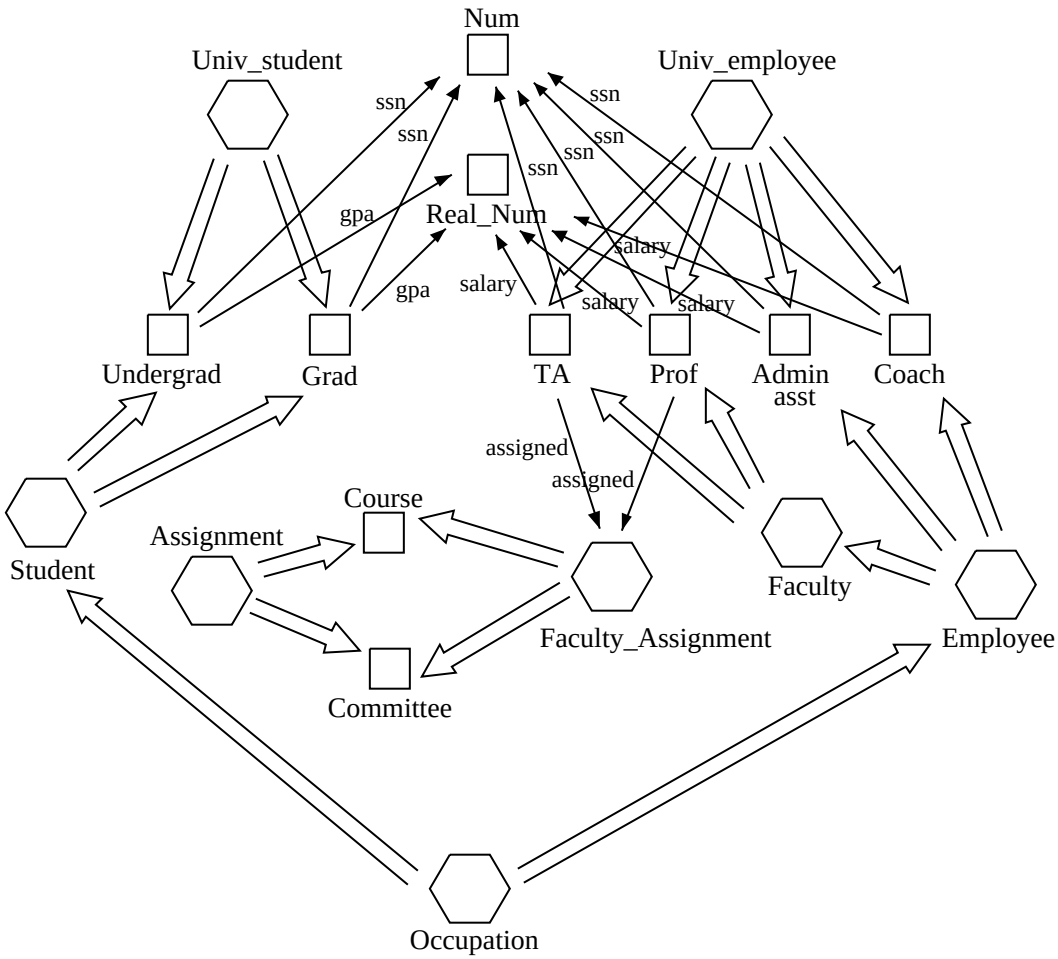


Figure 24: Distribution of Common Parts

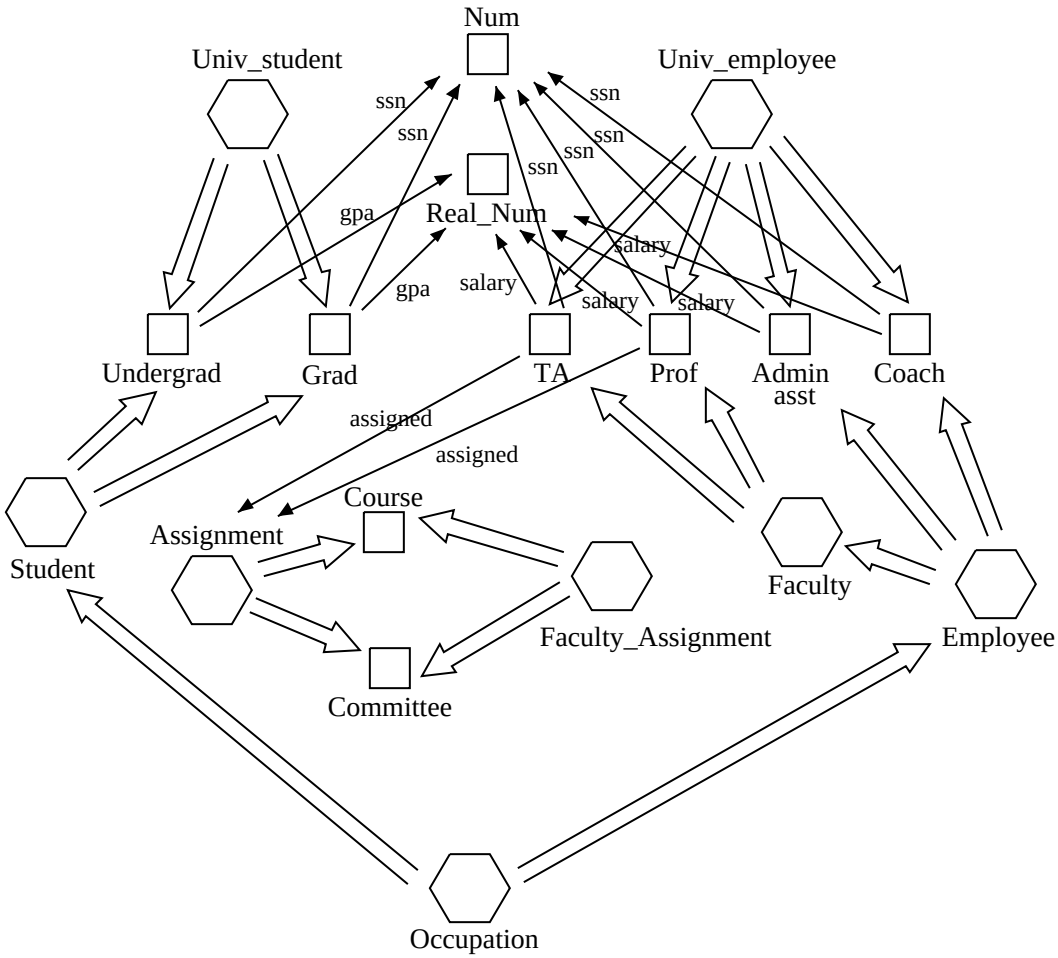


Figure 25: Part Replacement

edges are to vertex `Assignment`, so the edges are 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 25.

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 26.

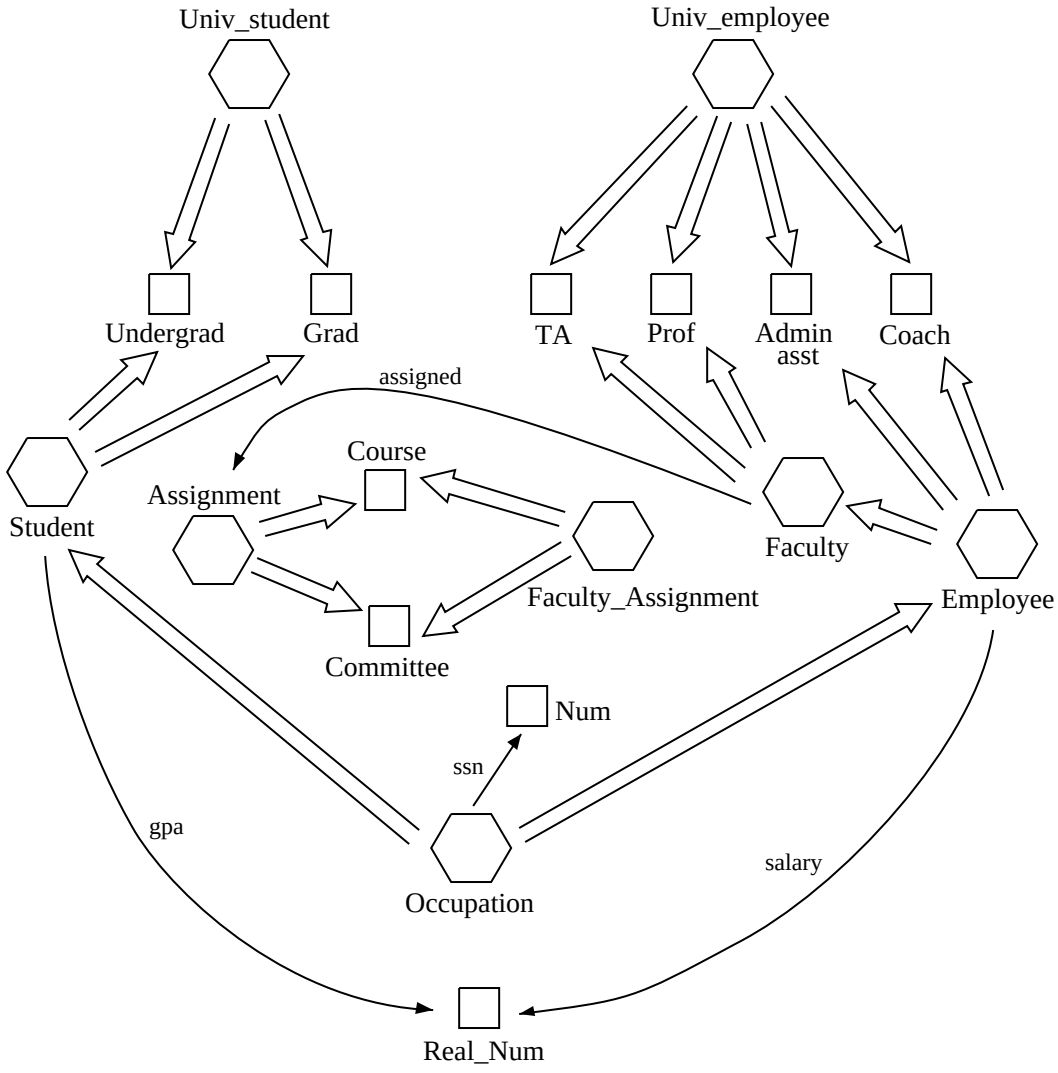


Figure 26: Abstraction of Common Parts

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.

4.4 Related work

Opdyke and Johnson [OJ90, Opd92] are investigating methods for *refactoring* object-oriented systems to support reuse. Refactorings are defined as restructuring plans and can be applied by performing a small set of basic refactorings.

Casais [Cas89, Cas90, Cas91] introduces global and incremental class hierarchy reorganization algorithms. Those algorithms differ from our work in a number of ways:

- The models used are different. Casais uses general graphs while we use graphs with a special structure which has to satisfy three axioms needed for data modeling. For example, we distinguish between abstract and concrete classes.
- The goal of Casais' algorithms is to restructure class hierarchies to avoid explicit rejection of inherited properties. In our work we currently avoid rejected properties.

Wegner describes informally the idea of a class dictionary transformation in his section on transformations of concept hierarchies in [Weg90]. He writes: "Such a calculus has interesting possibilities as an object-oriented design technique ..." We agree with Wegner and give a mathematical treatment of a calculus of class transformations.

Chapter 5

Class dictionary graph optimization

5.1 Practical applications of the object-preserving transformations

There are many useful rules which can be derived from the primitive transformations and are therefore guaranteed object-preserving. The examples in this section 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. Later, we introduce formal metrics and methods for class dictionary graph optimization.

5.1.1 Elimination of redundant parts

If a vertex, v, has two incoming construction edges with the same label, $(u \xrightarrow{l} v)$ and $(u' \xrightarrow{l} v)$, then those edges should be replaced by a single edge $(w \xrightarrow{l} v)$ 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 Figure 27.)

5.1.2 Removal of singleton alternation vertices

If an alternation vertex, v, has only one outgoing alternation edge, $(v \Longrightarrow w)$, then that vertex should be removed. Incoming construction edges $(u \xrightarrow{l} v)$, and alternation edges,

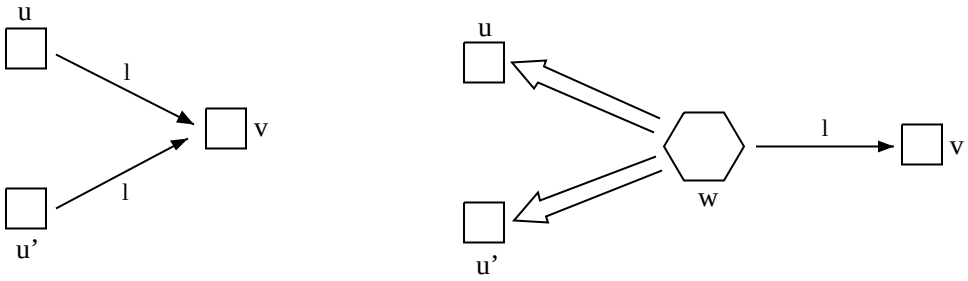


Figure 27: Elimination of redundant parts

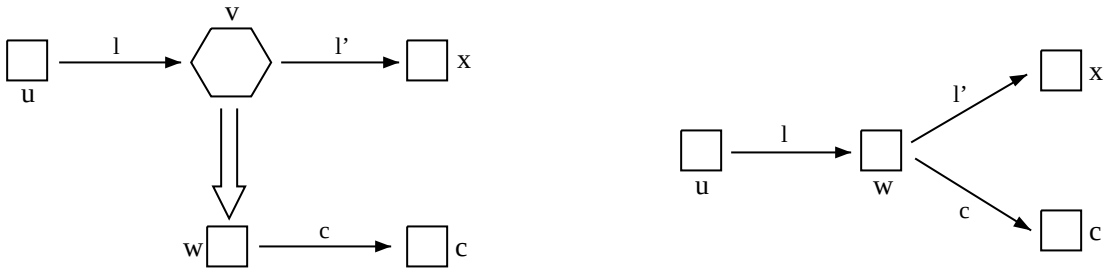


Figure 28: Removal of singleton alternation vertex

$(u \Longrightarrow v)$, are replaced by edges $(u \xrightarrow{l} w)$ and $(u \Longrightarrow w)$ respectively. Outgoing construction edges, $(v \xrightarrow{l} x)$, are replaced by edges $(w \xrightarrow{l} x)$. 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 Figure 28.)

5.1.3 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 4.3.2. (See Figure 29.)

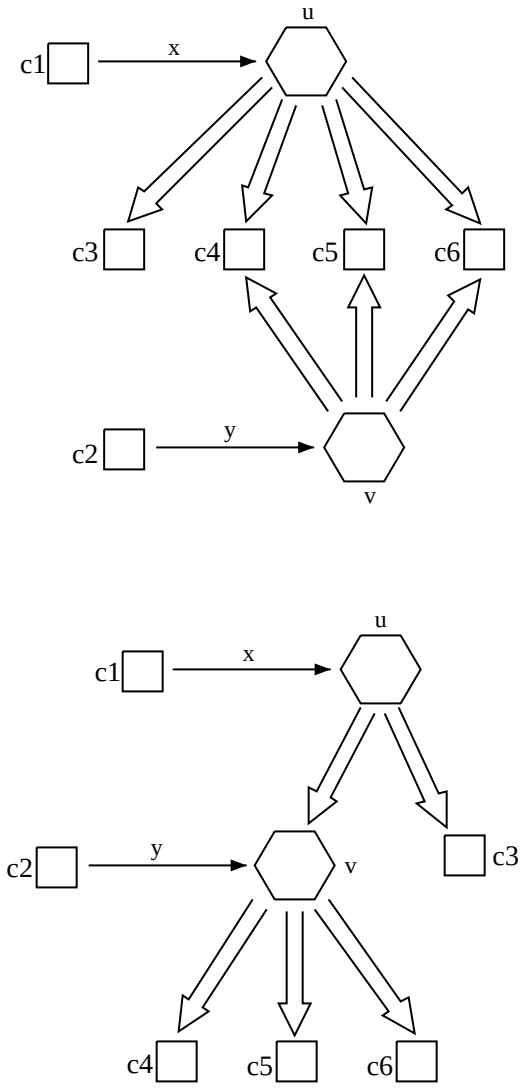


Figure 29: Complete Cover

5.1.4 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 $(u \Longrightarrow w)$ and $(v \Longrightarrow w)$. For each edge $(w \Longrightarrow x)$ which is added, the corresponding edges $(u \Longrightarrow x)$ and $(v \Longrightarrow x)$ are deleted. (See Figure 30.)

5.1.5 MI Minimization

If there are alternation edges, $(u \Longrightarrow w)$ and $(v \Longrightarrow w)$ such that for all other alternation edges from v, $(v \Longrightarrow w')$, w' is alternation reachable from u, then replace the edge $(u \Longrightarrow w)$ with the edge $(u \Longrightarrow v)$. This rule reduces the amount of multiple inheritance without changing the edge size. However, it introduces repeated inheritance. (See Figure 31.)

5.2 Metrics for class organizations

We propose a metric (minimizing the number of edges) for measuring class hierarchies [LBSL91]. We propose to minimize the *edge-size* of a class dictionary graph while keeping the set of objects invariant.

Definition 5.1. *The* **edge-size** *of a class dictionary graph, $\phi = (VC, VA, \Lambda; EC, EA)$, is defined by:*

$$size_\phi = |EC| + 1/4|EA|$$

The edge-size is the number of construction edges plus one quarter of the number of alternation edges. Note that the 1/4 constant is arbitrary. Any constant $c < 1/2$ would be appropriate. We want alternation edges to be cheaper than construction edges since alternation edges express commonality between classes explicitly and lead to better software organization through better abstraction and less code duplication.

This metric is quite rough: we just minimize the number of edges. We could minimize other criteria, such as the amount of multiple inheritance or the amount of repeated inheritance. A class B has repeated inheritance from class A, if there are two or more

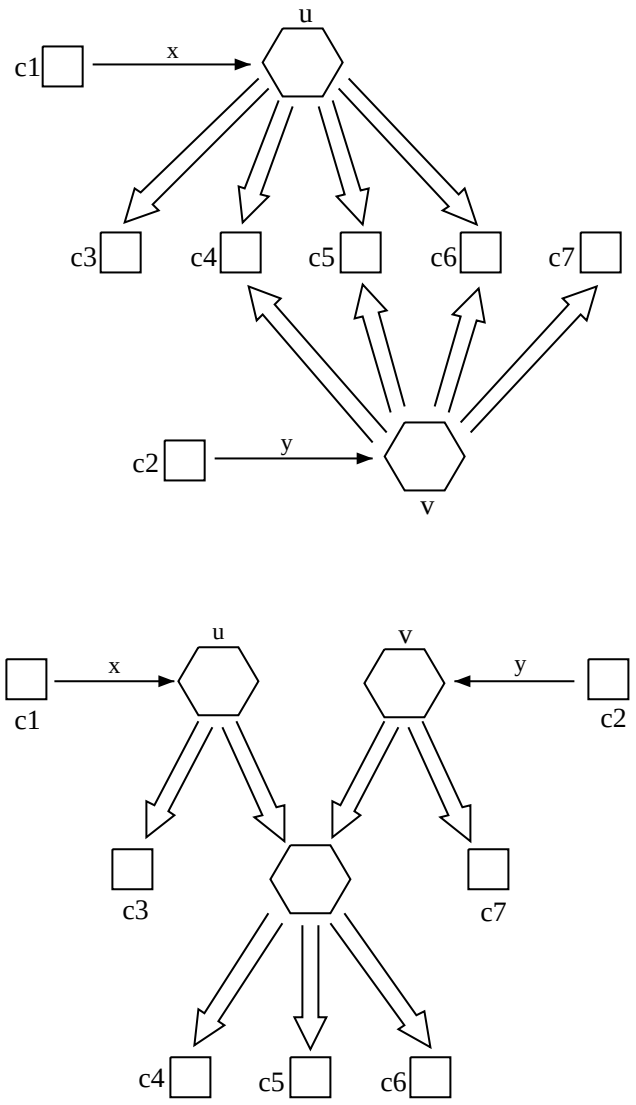


Figure 30: Partial cover

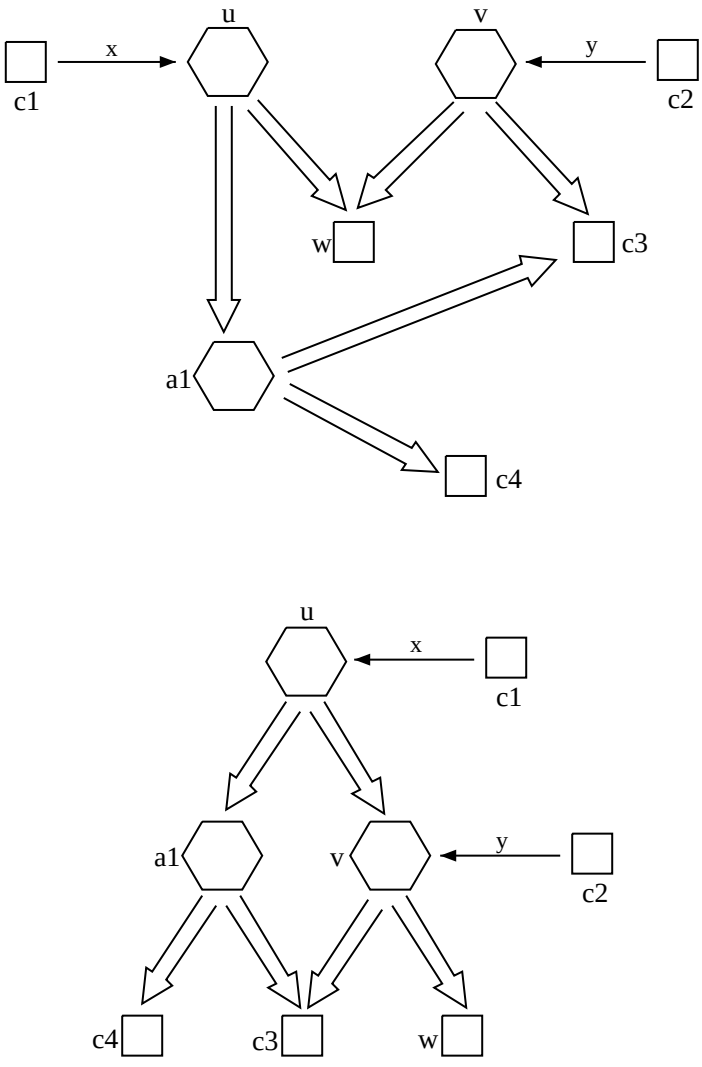


Figure 31: MI minimization

edge-disjoint alternation paths from A to B. The study of other metrics is left for future research.

5.3 Minimizing construction edges

The only way that the number of construction edges in a class dictionary graph can be reduced without changing the set of defined objects is by introducing additional inheritance. Two classes have a part in common if each class has a part with the same name, and that part may be instantiated by the same set of objects in instances of each class. If two or more classes share a common part which is not inherited from a common superclass, the number of construction edges may be reduced by introducing such inheritance. A class dictionary graph where all common parts are implemented through inheritance from common superclasses is said to be in **common normal form**[1] (CNF). Formally, a class dictionary graph is defined to be in common normal form if each equivalence class of the **part-equivalence** relation has only one member.

Definition 5.2. *For class dictionary graph* $\phi = (VC, VA, \Lambda;\ EC, EA)$ *with edges* $(v \xrightarrow{l} w), (v' \xrightarrow{l} w') \in EC$: $(v \xrightarrow{l} w)$ *is* **part-equivalent** *to* $(v' \xrightarrow{l} w')$ *iff* $\mathcal{A}(w) = \mathcal{A}(w')$, *where* $\mathcal{A}(x) = \{x' | x \stackrel{\star}{\Longrightarrow} x', x' \in VC\}$.

Of course, none of the equivalence classes of the part-equivalent relation can be eliminated from a class dictionary graph without changing the set of defined objects. Therefore a class dictionary graph in common normal form must have the minimum number of construction edges. Minimization of the construction edges can be accomplished in polynomial time using the primitive object-preserving transformations by exhaustive application of the **CNF Rule**.

5.3.1 CNF Rule

The CNF Rule says that if a class dictionary graph has equivalence classes of the part-equivalence relation with more than one member:

1. Select an equivalence class, R, such that $|R| > 1$.

[1] The concept of a common normal form free of redundant parts was first introduced informally by Ignacio Silva-Lepe.

2. Choose one edge, $(v \xrightarrow{l} w) \in R$, and replace every other edge, $(v' \xrightarrow{l} w') \in R$, with an edge $(v' \xrightarrow{l} w)$ by *part-replacement*.

3. Introduce a new alternation vertex, v', and for each $(v \xrightarrow{l} w) \in R$ introduce an alternation edge $(v' \Longrightarrow v)$ by *addition of useless alternation*. Then replace all edges $(v \xrightarrow{l} w) \in R$ with the single new edge $(v' \xrightarrow{l} w)$ by *abstraction of common parts*.

When the CNF Rule is applied with an equivalence class, R, of size n, n new alternation edges are added and $n-1$ construction edges are eliminated. The total number of edges increases by one, but the edge-size decreases since $(n > 1) \Rightarrow (n - 1 > n/4)$. Thus, the $1/4$ constant in the edge-size definition guarantees that a class dictionary graph with a minimum edge-size also has the minimum number of construction edges. That is, it is never possible to decrease the edge-size by eliminating alternation edges at the expense of adding construction edges.

The minimization technique depends upon the consistent use of part names in a class dictionary graph. If the input does not contain the structural key abstractions of the application domain then the optimized hierarchy will not be useful either, following the maxim: garbage in – garbage out.

However if the input uses names consistently to describe either example objects or a class dictionary graph then our metric is useful in finding "good" hierarchies. However, we don't intend that our algorithms be used to restructure class hierarchies without human control. We believe that the output of our algorithms makes valuable proposals to the human designer who then makes a final decision.

Even simple functions cannot be implemented properly if a class dictionary graph is not in CNF. By properly we mean with resilience to change.

Consider the class dictionary graph shown in Figure 32, which is not in CNF. Suppose we implement a print function for Coin and Brick. Now assume that several hundred years have passed and that we find ourselves on the moon where the weight has a different composition: a gravity and a mass. We then have to rewrite our print function for both Coin and Brick.

After transformation to CNF we get the class dictionary graph in Figure 33. Now we implement the print function for Coin:

```
void Coin::print() {
  radius -> print(); Weight_related::print();}
```

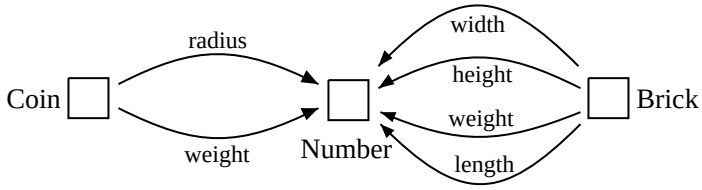


Figure 32: Before transformation to CNF

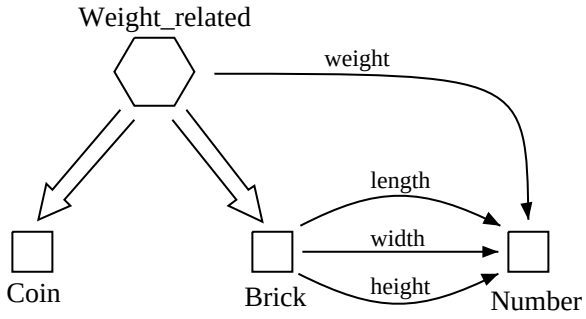


Figure 33: After transformation to CNF

After the change of the weight composition, we get the class dictionary graph in Figure 34. We reimplement the print function for this new class and no change is necessary for classes Brick and Coin.

In summary: if the class dictionary graph is in CNF and the functions are written following the strong Law of Demeter [LHR88], the software is more resilient to change. The strong Law of Demeter says that a function f attached to class C should only call functions of the *immediate* part classes of C, of argument classes of f, including C, and of classes

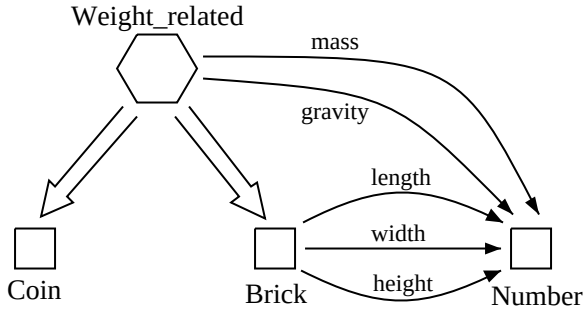


Figure 34: After change of weight composition

which are instantiated in f.

5.4 Minimizing alternation edges

A class dictionary graph with a minimal edge-size may be obtained by first minimizing the number of construction edges, and then minimizing the number of alternation edges while holding the number of construction edges constant. In this section a procedure for minimizing the alternation edges (i.e. computing an optimal alternation subgraph) is developed which provides the means for overall optimization. The problem of minimizing the alternation edges is NP-hard[2], but algorithms for achieving a fast approximation to the optimum are provided in the next section. In the next section we also present a fast algorithm for an exact solution in the special case where the minimum is a single-inheritance hierarchy.

An optimal class dictionary graph cannot have any more alternation edges than any object-equivalent class dictionary graph which is in CNF. Since each application of the CNF rule adds only one alternation edge for each construction edge in some part-equivalence class, conversion to CNF adds $|EC|$ alternation edges in the worst case, and the total number of alternation edges in the optimal class dictionary graph is bounded by $|EA| + |EC|$. But every alternation vertex in an optimal class dictionary graph must have at least two outgoing alternation edges, so the number of alternation vertices in an optimal solution is bounded by $1/2(|EA| + |EC|)$ and the total number of vertices (including construction vertices) is bounded by $1/2(|EA| + |EC|) + |VC|$. In other words, the size of the solution is linearly related to the size of the input.

We can represent any optimal inheritance graph by a binary $s \times s$ matrix, M, where $s = 1/2(|EA| + |EC|) + |VC|$. If there is an alternation edge from vertex v_i to vertex v_j the value of $M_{i,j}$ is 1, otherwise it is 0. The total number of alternation subgraphs to consider must therefore be bounded by 2^{s^2}. In fact, the upper bound is significantly lower. Since the alternation subgraph is a directed acyclic graph, let $v_1, v_2, ...v_m, v_{m+1}, v_{m+2}, ...v_s$ where $m = |VC|$, be a topological sorting of vertices such that if $a_i \stackrel{\star}{\Longrightarrow} a_j$ then $i > j$. Then we need not consider elements of the matrix $M_{i,j}$ where $i \leq j$. Since the construction vertices have no outgoing alternation edges, and each v_i, $(1 \leq i \leq m)$, must be a construction vertex, we can also disregard elements of the matrix, $M_{i,j}$, where $i \leq m$. If there are m construction vertices and n alternation vertices in the matrix, then the upper bound is

[2]See [LBSL91] for a formal proof due to Ignacio Silva-Lepe.

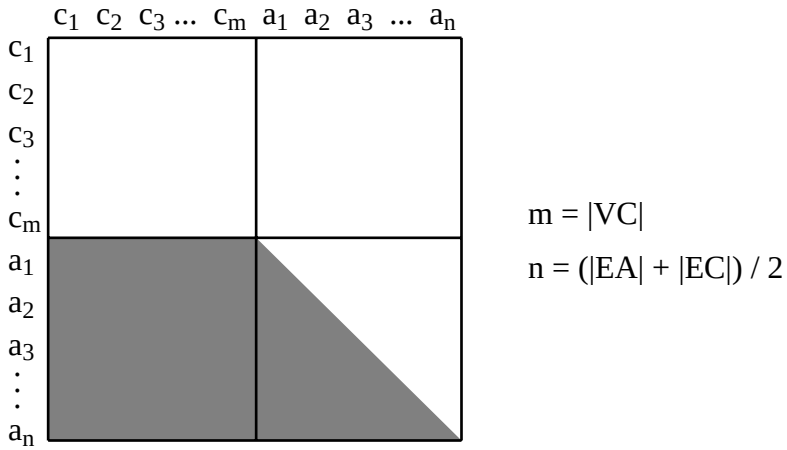


Figure 35: Matrix representation of alternation subgraph

reduced from $2^{s^2} = 2^{n^2+2nm+m^2}$ to $2^{n^2/2+nm}$ as shown by the shaded portion of Figure 35.

In order to compute an optimal alternation subgraph, we need to be able to determine if a given alternation subgraph contains all of the alternation vertices and edges which are required to place the class dictionary graph in CNF. We can think of each alternation vertex as representing (or covering) a set of construction vertices. This set, $\mathcal{A}(v)$, consists of all the construction vertices which are alternation reachable from the alternation vertex, v. If an alternation vertex, v, has an incoming construction edge, $(u \xrightarrow{l} v)$, the construction vertices in $\mathcal{A}(v)$ represent the concrete classes which might be used to instantiate the l part in u objects. If v has an outgoing construction edge, $(v \xrightarrow{l} w)$, the construction vertices in $\mathcal{A}(v)$ represent the concrete classes which inherit the l part from v.

For each construction edge, $(v \xrightarrow{l} w)$ in the optimal class dictionary graph, we must be able to represent the set of classes associated with the source of the edge, $\mathcal{A}(v)$, and the set of classes associated with the target of the edge, $\mathcal{A}(w)$. An optimal inheritance hierarchy meets this requirement with the minimum number of alternation edges.

The only primitive object-preserving transformation that changes the target of a construction edge is part replacement, but part replacement does not change the set of classes associated with the target. Therefore, the sets of classes associated with construction edge targets in the optimal class dictionary graph can be calculated from any object-equivalent class dictionary graph.

In any class dictionary graph with a minimal number of construction edges, there is exactly one edge in each part-equivalence class. Furthermore, if an edge, $(v \xrightarrow{l} w)$, in

one class dictionary graph with a minimum number of construction edges is part-equivalent to an edge, $(v' \xrightarrow{l} w')$, in an object-equivalent class dictionary graph which also has a minimum number of construction edges, then the vertices v and v' must have the same set of associated classes. The part represented by the equivalent edges must be inherited in exactly the same set of construction classes in each case since the class dictionary graphs are object-equivalent. Therefore, the sets of classes associated with edge sources in the optimal class dictionary graph can be calculated from the class dictionary graph that results from transformation to common normal form. As long as the number of construction edges is kept minimal these sets remain constant. In practice, it is not necessary to actual convert to CNF in order to perform the computation. The following algorithm can be used to compute the sets of classes, T, associated with the targets of construction edges and the sets of classes, S, associated with the sources of construction edges in an optimal solution:

Algorithm to compute sets of associated classes:

1. Let $T = \emptyset$. (Target sets)
2. Let $S = \emptyset$. (Source sets)
3. For each edge, $(u \xrightarrow{l} v) \in EC$:
   (a) Let $T_{(u,v,l)} = \{v' \mid v' \in VC \text{ and } v \stackrel{*}{\Longrightarrow} v'\}$
   (we associate $T_{(u,v,l)}$ with edge $(u \xrightarrow{l} v)$)
   (b) $T = T \cup \{T_{(u,v,l)}\}$
4. For each element, $t \in T$:
   (a) Let $e_t = \{(u \xrightarrow{l} v) \in EC \mid T_{(u,v,l)} = t\}$.
   (b) For each label, l, such that $\exists (u \xrightarrow{l} v) \in e_t$: $S = S \cup \{\{u' \mid u' \in VC \text{ and } \exists (u \xrightarrow{l} v) \in e_t : u \stackrel{*}{\Longrightarrow} u'\}\}$.
   (we associate this element of S with edges (u, v, l))

Now we have an algorithm to find an optimal inheritance hierarchy:

1. Compute the sets classes associated with targets and sources of construction edges in the optimal class dictionary, T and S.

2. Let $m = |\ VC|, n = 1/2(|\ EA\ | + |\ EC|)$.

   For $k \leftarrow 0$ to $2n$ consider, in turn, each inheritance graph, g, with one of the $\begin{pmatrix} n^2/2 + nm \\ k \end{pmatrix}$ combinations of k edges.

   - If for each set, $r \in S \cup T$, there exists a vertex v in g such that $\mathcal{A}(v) = r$ then stop and return g.

Note that there may be more than one optimal solution; that is, there may be more than one inheritance graph with the same minimum number of edges that contains the required vertices. In such cases, we arbitrarily return the first one we find.

An arbitrary class dictionary graph, ϕ, can be optimized by computing an optimal inheritance hierarchy and then following the construction of the completeness proof for the object-preserving transformations:

1. Compute an optimal alternation subgraph.
2. Superimpose the optimal alternation subgraph on the old class dictionary graph (by addition of useless alternation).
3. Apply distribution of common parts exhaustively (until all parts are attached directly to construction vertices).
4. Apply part replacement to insure that the target of every construction edge is in the optimum inheritance hierarchy.
5. Delete the old inheritance hierarchy (by deletion of useless alternation).
6. Apply abstraction of common parts exhaustively (until there is only one edge in each part-equivalence class).

5.5 Fast algorithms for optimization

5.5.1 Single-inheritance hierarchies

There is a fast algorithm to minimize the alternation edges in a class dictionary graph when the solution is a single-inheritance hierarchy [LBSL90]. Given a class dictionary graph, ϕ, in CNF, delete all useless alternation vertices and consider the associated classes, $\mathcal{A}(v)$, for

each remaining alternation vertex, v. We say that ϕ has the *tree property* if $\forall v, v' \in VA$ one of the following conditions holds:

- $\mathcal{A}(v) \cap \mathcal{A}(v') = \emptyset$
- $\mathcal{A}(v) \cap \mathcal{A}(v') = \mathcal{A}(v)$
- $\mathcal{A}(v) \cap \mathcal{A}(v') = \mathcal{A}(v')$

When a class dictionary graph has the tree property, we can restructure the alternation subgraph as a tree by inspecting the containment relationship between the sets of associated vertices[3]. Finally, the tree is collapsed by eliminating any singleton alternation vertices (Section 5.1.2) and any alternation vertices with no incoming or outgoing construction edges. Now there is only one alternation vertex for each set of classes that must be represented in the class dictionary graph. Furthermore, since the result is a tree, each set is optimally expressed in terms of subsets, so the result has the minimum number of alternation edges.

5.5.2 Common normal form

We give a fast algorithm, called "ACP" (for abstraction of common parts), for transforming to common normal form. This algorithm is not as simple as applying the CNF rule but it behaves well in practice in combination with the "consolidation of alternatives" algorithm (below) for minimizing alternation edges.

Algorithm ACP (Abstraction of Common Parts)

1. Add to the original class dictionary graph an alternation vertex, v, which has as alternatives all vertices of the original class dictionary graph which do not have any incoming alternation edges. This insures that every vertex in the class dictionary graph is alternation-reachable from v.
2. Call AR(v) to compute for each vertex the set of alternation-reachable construction vertices.
3. Call ACP-Vertex(v).
4. If there are no construction edges outgoing from v, delete vertex v and all of its outgoing edges $(v \Longrightarrow w) \in EA$.

[3] If two alternation vertices have the *same* set of associated classes, so that there is no proper containment relationship, one of them can be eliminated if its attached edges are transferred to the other.

Algorithm AR(v) (Alternation Reachable)

1. If v is marked "AR-DONE", return S_v.
2. If $v \in VC$ then $S_v = \{v\}$.

   Else

   (a) $S_v = \emptyset$

   (b) For each w, where $(v \Longrightarrow w) \in EA$, $S_v = S_v \cup$ AR(w)
3. Mark v "AR-DONE" and return S_v.

Algorithm ACP-Vertex(v)

1. If $v \notin VA$ or v is marked "ACP-DONE", return.
2. For each w, where $(v \Longrightarrow w) \in EA$, call ACP-Vertex($w$)
3. While there is a label l and a set of construction classes, S, such that $\forall (v \Longrightarrow w) \in EA : \exists (w \stackrel{l}{\longrightarrow} u) \in EC$ such that $\mathcal{A}(u) = S$ (we say the part (l, u) is redundant in every alternative of v), replace one such construction edge, $(w \stackrel{l}{\longrightarrow} u)$, with the new construction construction edge $(v \stackrel{l}{\longrightarrow} u)$, and delete all other such construction edges.
4. While there is a part that is redundant in two or more alternatives of v:

   (a) Select a part, (l, u), which is redundant in at least as many alternatives as any other part.

   (b) Introduce a new alternation vertex, v' and add construction edge $(v' \stackrel{l}{\longrightarrow} u)$ and alternation edge $(v \Longrightarrow v')$.

   (c) For each $w \neq v'$ such that $(v \Longrightarrow w) \in EA$ and $(w \stackrel{l}{\longrightarrow} u') \in EC$ where $\mathcal{A}(u) = \mathcal{A}(u')$, delete edges $(w \stackrel{l}{\longrightarrow} u')$ and $(v \Longrightarrow w)$ and add the alternation edge $(v' \Longrightarrow w)$.

   (d) Call ACP-Vertex(v').
5. While there is a part (l, u) which is redundant in two or more vertices which are alternation-reachable from v:

   (a) Introduce a new alternation vertex, v' and add construction edge $(v' \stackrel{l}{\longrightarrow} u)$.

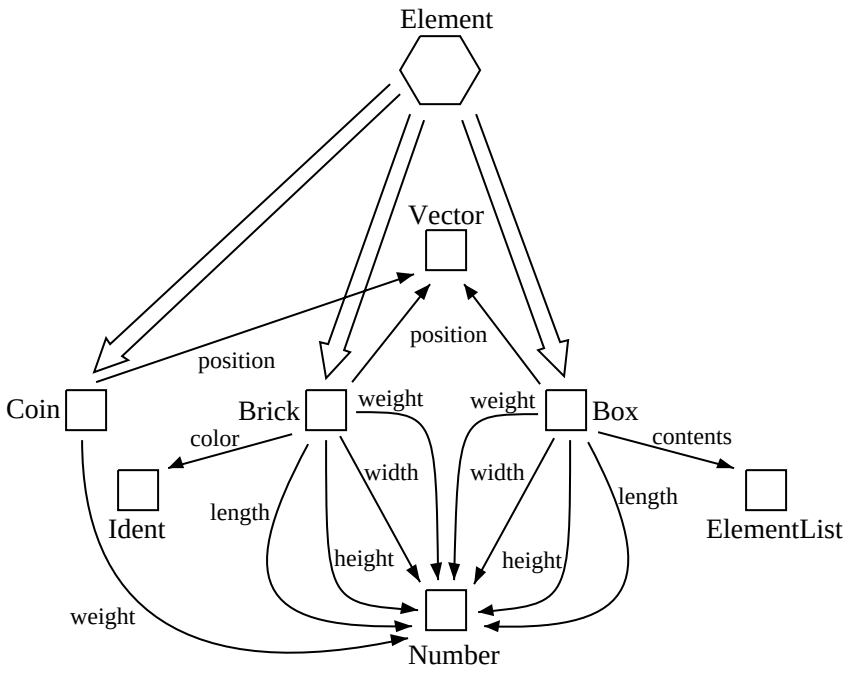


Figure 36: Original class dictionary graph

(b) For each w alternation-reachable from v where $(w \xrightarrow{l} u') \in EC$ such that $\mathcal{A}(u) = \mathcal{A}(u')$, delete edge $(w \xrightarrow{l} u')$ and add the alternation edge $(v' \Longrightarrow w)$. If w is an alternative of v, then also replace the alternation edge $(v \Longrightarrow w)$ with $(v \Longrightarrow v')$.

(c) Call ACP-Vertex(v').

6. Mark v "ACP-DONE" and return.

Example 5.1. *We demonstrate the normal form transformation by algorithm ACP with the class dictionary graph in Figure 36. This class dictionary graph is not in common normal form since weight and position are redundant in* Coin, Brick *and* Box. *Therefore we factor them, i.e.:*

```
Element : Coin | Brick | Box
  *common* <weight> Number <position> Vector.
```

The resulting class dictionary graph, which is still not in common normal form, is shown in Figure 37. We introduce a new class:

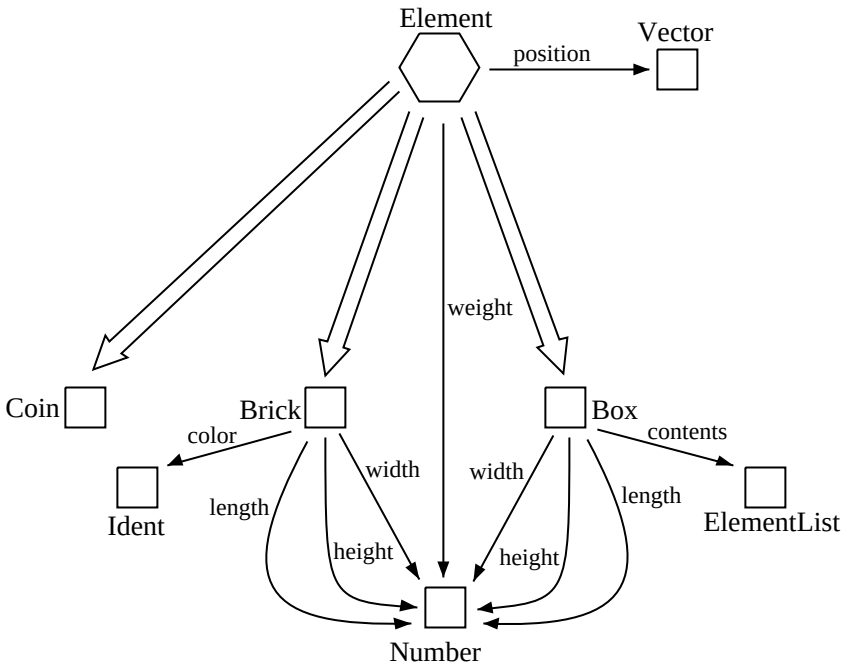


Figure 37: After factoring weight and position

```
QuadrangularElement : Brick | Box
  *common* <width> Number <height> Number <length> Number.
```

The resulting class dictionary graph, shown in Figure 38, is now in common normal form.

5.5.3 Minimizing alternation edges

The algorithm presented in this section is fast but only provides an approximate solution for minimizing the alternation edges.

There are two aspects to minimizing alternation edges:

- Inventing new alternations. We try to find new alternation vertices which allow us to decrease the number of outgoing alternation edges of existing alternation vertices. This leads to a deepening of the inheritance hierarchy. For example (cf. Figure 39):

  ```
  A1 : A | B | C | D | E.
  B1 : A | B | C.
  ```

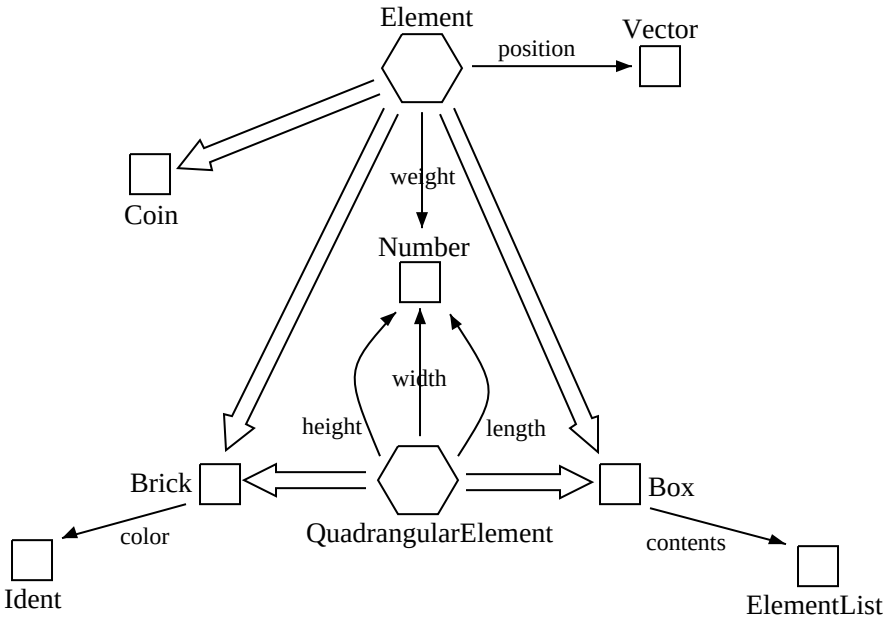


Figure 38: After factoring width, height, and length

```
C1 : A | B |     D.
D1 : A | B |         E.
```

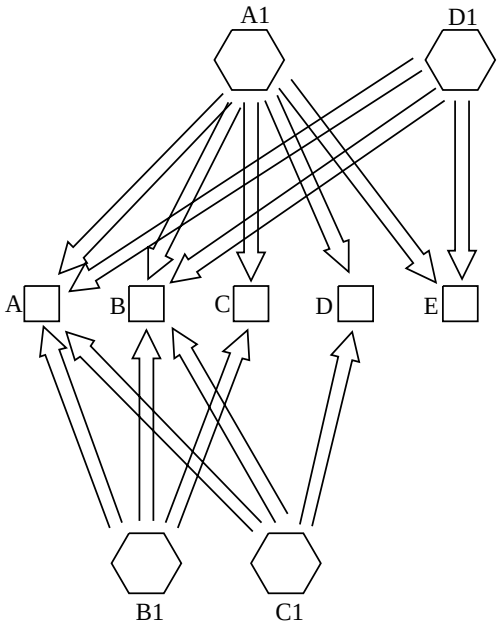
can be abbreviated to:

```
A1 : N1 | C | D | E.
B1 : N1 | C.
C1 : N1 |     D.
D1 : N1 |         E.
N1 : A  | B.
```

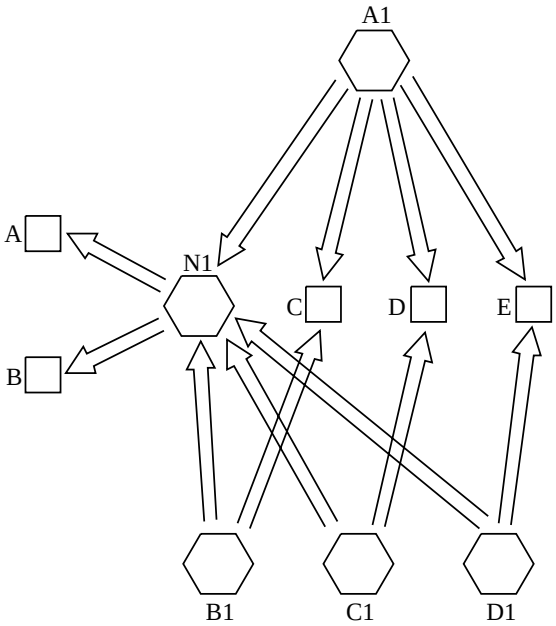
- Using alternations for "covering" existing alternations. We try to express a given alternation in terms of existing alternations. For example (cf. Figure 40):

```
Cover  : A | B | C | D | E.
A1 :     A |     C |     E.
B1 :         B |     D .
```

can be abbreviated to



Example inheritance hierarchy



Deepened inheritance hierarchy

Figure 39: Inventing new alternations

```
Cover : A1 | B1.
A1 :     A |     C |     E.
B1 :         B |     D .
```

In this example multiple inheritance is removed since we found an "exact cover" of Cover with A1 and B1.

The algorithm "Consolidate" we give next is better at introducing new alternations than at optimally reusing existing alternations.

Algorithm: Consolidate Alternatives

The algorithm considers for all alternation vertices the set of all possible unordered pairs of alternatives defined by the same alternation vertex.

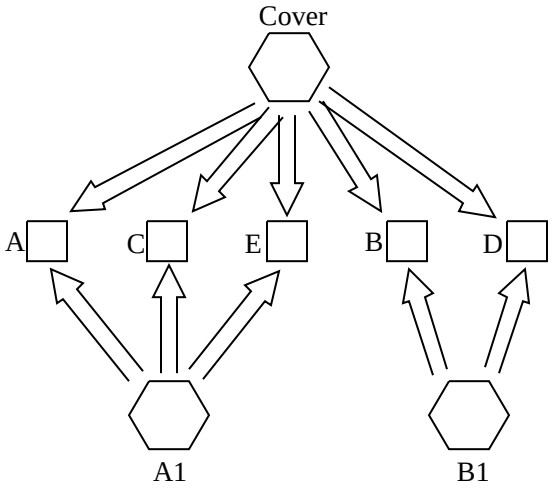
If there are pairs that are defined by two or more alternation vertices:

1. Select the pair of alternatives (α_1), (α_2), defined by the most alternation vertices.
2. Create a new alternation vertex α_3. Create two new alternation edges with source vertex α_3 and target vertices α_1 and α_2, respectively.
3. For each alternation vertex α_i that defines the pair,
   - Delete the two outgoing alternation edges with targets α_1 and α_2 and source α_i.
   - Add a new alternation edge with source α_i and target α_3.
4. Consolidate alternatives in the new class dictionary graph. (Call this algorithm recursively).

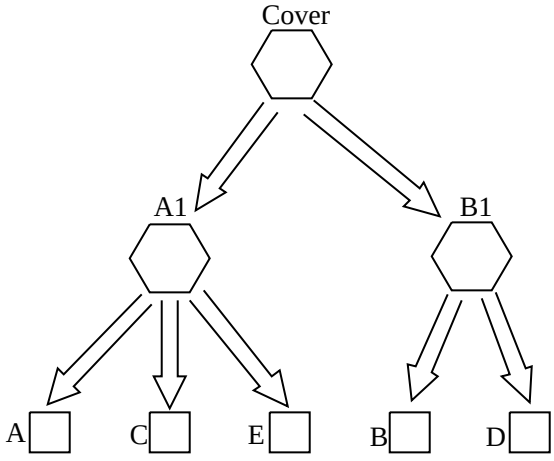
For each alternation vertex α_j in the class dictionary graph,

- If α_j has only one incoming alternation edge with source vertex α_k and no incoming or outgoing construction edges then:
  1. Make α_k the source vertex of each and every outgoing alternation edge of α_j.
  2. Delete the alternation edge (α_k,α_j).
  3. Delete the alternation vertex α_j.

Example 5.2. *The following example shows how the algorithm recognizes common triples:*



(a)



(b)

Figure 40: Covering existing alternations

```
Z : A | B | C | D.
Y : A | B | C |    E.
X : A | B | C |       F.
```

The algorithm first learns:

```
AOrB : A | B.
Z    : AOrB | C | D.
Y    : AOrB | C |    E.
X    : AOrB | C |       F.
```

Then:

```
AOrBOrC : AOrB | C.
AOrB : A | B.
Z    : AOrBOrC | D.
Y    : AOrBOrC |    E.
X    : AOrBOrC |       F.
```

Now, AOrB *is eliminated:*

```
AOrBOrC : A | B | C.
Z    : AOrBOrC | D.
Y    : AOrBOrC |    E.
X    : AOrBOrC |       F.
```

Analysis:

- Running time

  If we start out with p alternation vertices the algorithm might add $O(p^2)$ alternation vertices. In the worst case we have to look at all pairs of alternatives and therefore the running time of the algorithm is $O(Size(input)^4)$.

- Correctness

  The algorithm does not change the set of objects defined by the class dictionary graph.

We now turn to class dictionary graph minimization in general.

A useful approximation algorithm is to first use algorithm ACP followed by algorithm Consolidate Alternatives.

Consider the following non-minimal class dictionary graph:

```
Occupation :
  Undergrad_student | TA | Professor | Adm_assistant
  *common* <ssn> Number.
Student : Undergrad_student | TA *common* <gpa> Real.
Faculty : Professor | TA *common* <course_assigned> Course.
Professor = .
TA = .
Adm_assistant = .
Course = .
Undergrad_student = <major> Area.
Area : Economics | Comp_sci.
Economics = .
Comp_sci = .
University_employee : TA | Professor | Adm_assistant
                      *common* <salary> Real.
```

Change the class definitions for Occupation and University_employee to

```
Occupation : Student | University_employee *common* <ssn> Number.
University_employee : Faculty | Adm_assistant *common* <salary> Real.
```

We have now reduced the number of alternation edges by 3 at the expense of adding repeated inheritance. By repated inheritance we mean that a class is inherited several times in the same class. In the above example, class Occupation is inherited twice in class TA:

```
Occupation -> University_employee -> Faculty -> TA
           -> Student ->                        TA
```

However, not only alternation edges are reduced, also the amount of multiple inheritance, which we propose as another metric to produce "good" schemas from the software engineering point of view.

Repeated inheritance is undesirable under certain situations. For example, when we implement the class hierarchy in C++ using virtual base classes, we can no longer cast an Occupation object to a TA object.

Another indication that our class dictionary graph optimization algorithm MCDL is useful is that it succeeds in finding single-inheritance solutions. We can prove the following statement: If we give a class dictionary graph which is object-equivalent to a single-inheritance class dictionary graph to the optimization algorithm MCDL, it will return such a single-inheritance class dictionary graph. From a software engineering standpoint, a single inheritance hierarchy is simpler than a multiple-inheritance hierarchy and our optimization algorithm will find such a hierarchy, if there is one.

5.6 Related work

Automatic structuring of classes is studied in [Pir89]. Cardelli [Car84] proposes a technique for inferring multiple inheritance from objects (records). But he does not deal with the optimality question addressed in this paper: what is the optimum way of inferring inheritance?

Pun and Winder [PW89] discuss automatic class hierarchy construction. They describe the process of factoring out common parts from an existing set of classes to form superclasses, but do not include a learning component to obtain the initial set of classes.

Instead of providing algorithms, Pun and Winder suggest that a factorization engine could be built based on an existing computer algebra system. A "normalized class hierarchy" is obtained when there are no more common parts to factor. Thus, the factorization engine performs an operation similar to the CNF transformation.

The CNF transformation presented in this chapter extends the work of Pun and Winder in several ways. First, our model allows composite objects including recursion while Pun and Winder allow only objects with a flat list of attributes. Second, we give an algorithm for the CNF transformation and show that the time complexity is a polynomial of low degree. Finally, we introduce the concept of object-equivalence which defines the legal transformations on a class hierarchy.

Pun and Winder propose construction of a "normalized expression filter" to produce a "most desirable" normalized class hierarchy. The filter would be constructed as an expert system allowing users to input rules and constraints which might, for example, specify the

priority of certain parts in the factorization process. In contrast, we introduce the concept of an optimal class dictionary and show how the optimization can be fully automated. We further show that the time complexity for optimization is in P for the single inheritance case, but that the multiple inheritance case is NP-hard.

In software engineering, program reorganization based on the degree of coupling has been used in [KK88, LH89, Cas90]. Inductive inference techniques are reported in [DJ88]. In the relational database field various algorithms for deriving schemas in normal form have been developed to help the application builder to pin-point design flaws [Lie85].

Chapter 6

Maintaining Behavioral Consistency

Most of the recent work on schema[1] evolution and transformations, [Opd92, Ber92, Ber91, Cas91, CPLZ91, DZ91, Bar91, LH90, AH88, BKKK87, PS87, SZ86], 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] update the persistent instances in a database to guarantee structural consistency with a transformed schema. However, none them considers code updates on existing programs to restore behavioral consistency.

In this chapter[2] the problem of behavioral consistency is considered for an important subset of possible class dictionary graph transformations. The transformations in this subset are the object-preserving transformations defined in chapter 4 plus three additional transformations [LHX94] that do not preserve objects.

These transformations have three desirable properties. First, the transformed class dictionary graph'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 not need to be repopulated, or that the repopulation can be easily automated. Second, the extension transformations are powerful enough to allow the learning and incremental extension of class dictionary graphs defined in Chapter 3 and the optimizations defined in Chapter 5 as well as other transformations that commonly occur in practice. Third, they can be decomposed

[1] Class dictionary graphs are one model of object schemas

[2] The work in this chapter was completed in collaboration with Walter Hürsch and first appears in [BH93].

into a sequence of primitive transformations.

Our strategy for solving the behavioral consistency problem relies heavily on the third property. A given extension is decomposed into a sequence of primitives, and the problem is solved for each of the primitives in turn.

We consider (informally) two very different language models: strongly typed and untyped. We compare solutions to the behavioral consistency problem in the two models using C++ and CLOS (Common Lisp Object System), respectively, as representative examples. As one might expect, the problem is much more difficult for the strongly typed model. 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 straight-forward 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 41 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;
};
```

6.1 Features of language models

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++.

6.2 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 ϕ_1 and ϕ_2 are **object-equivalent** if they both define the same set of objects. Consequently, ϕ_1 and ϕ_2 must satisfy these conditions: (1) ϕ_1 and ϕ_2 have the same set of construction classes. (2) A construction class A of ϕ_1 has a (inherited or direct) part b if and only if its corresponding class in ϕ_2 has a (inherited or direct) part b . (3) An instance can be assigned to part b of class A in ϕ_1 if and only if the instance can also be assigned to part b of class A in ϕ_2.

As an example of two class dictionary graphs in an object-equivalence relation, consider Figures 41 and 42. 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 ϕ_1 and ϕ_2 are in an **extension relation**, such that ϕ_2 extends ϕ_1, if they satisfy these conditions: (1) The set of construction classes of ϕ_2 is a superset of the set of construction classes of ϕ_1. (2) If a construction class A of ϕ_1 has a (inherited or direct) part b, then its corresponding class in ϕ_2 has a (inherited or direct)

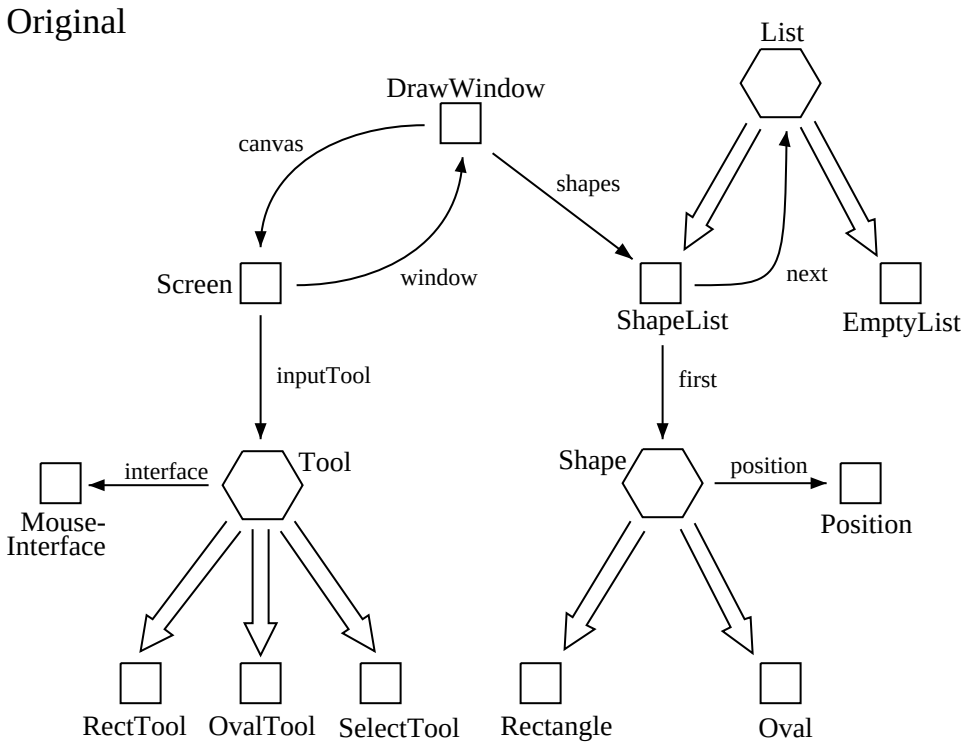


Figure 41: Original class dictionary graph

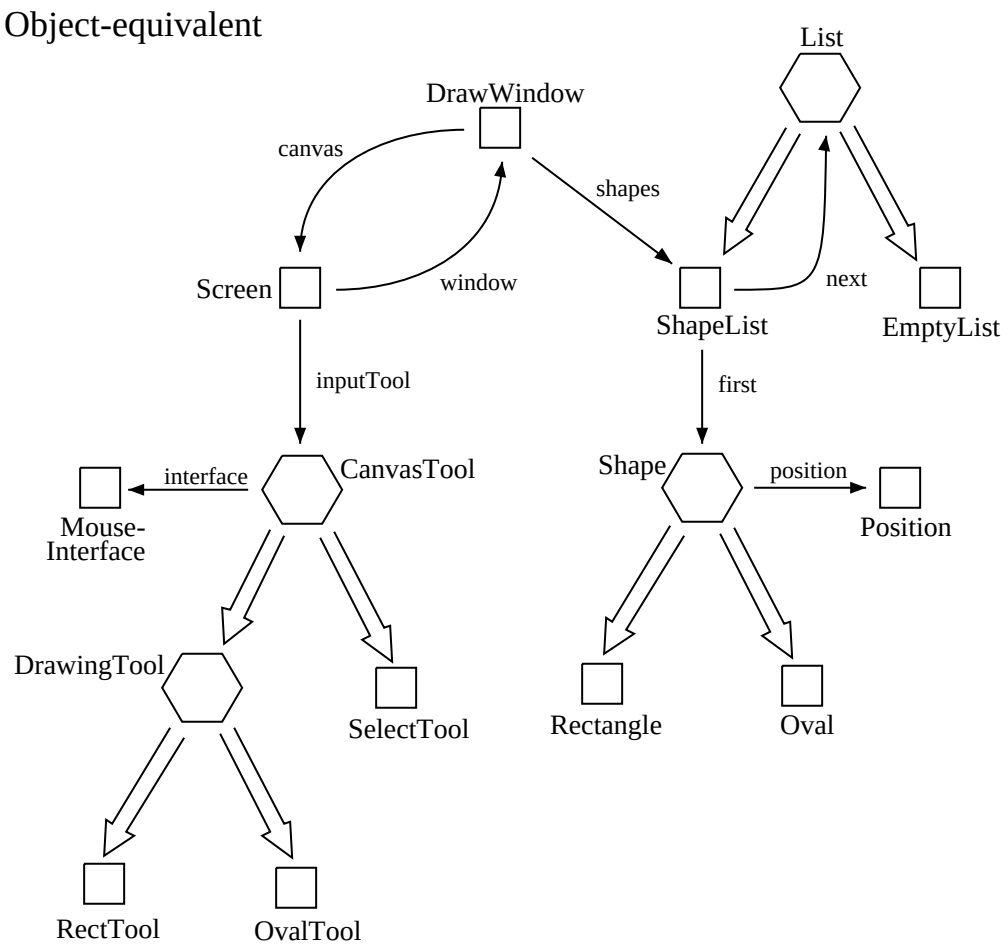


Figure 42: Object-equivalent class dictionary graph

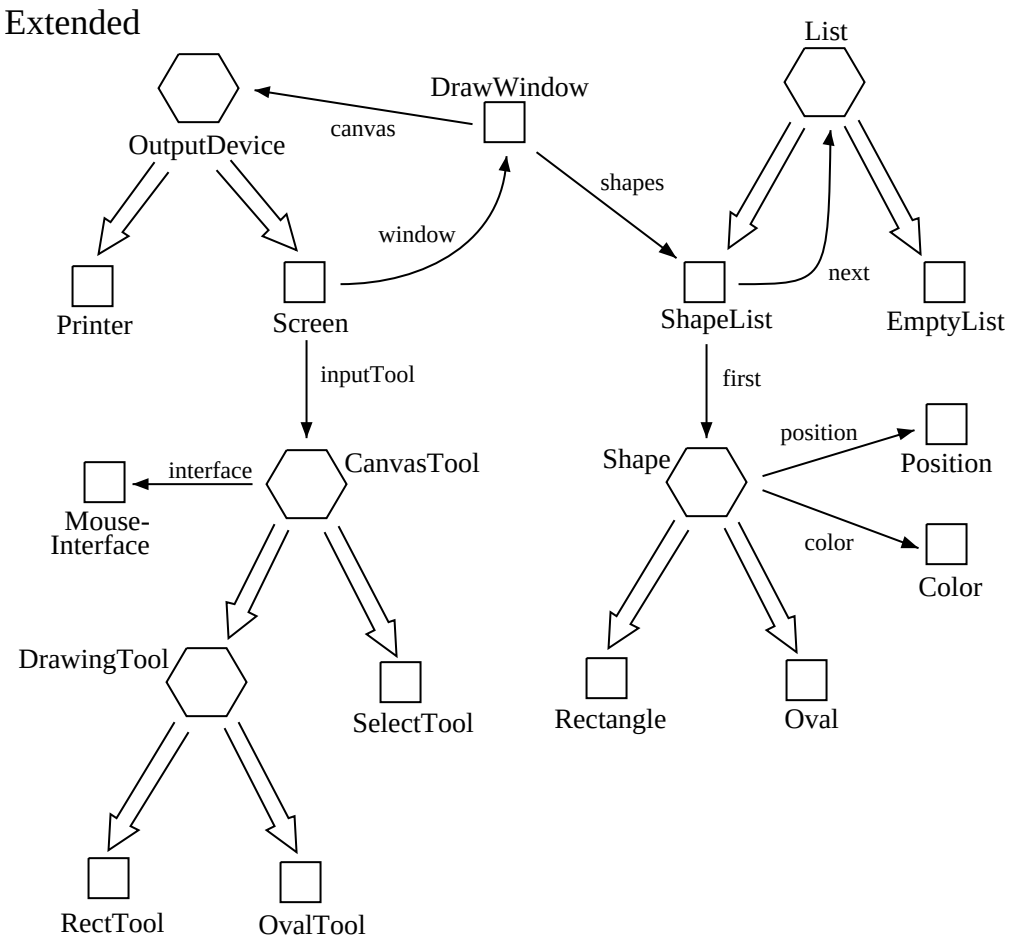


Figure 43: Extended class dictionary graph

part b . (3) If an instance can be assigned to part b of class A in ϕ_1, then the instance can also be assigned to part b of class A in ϕ_2. An example of two class dictionary graphs in an extension relation is given in Figures 42 and 43.

As a consequence of the above definitions the following relationship holds between extension and object-equivalence. Class dictionary graph ϕ_1 is object-equivalent to class dictionary graph ϕ_2 if and only if ϕ_1 is extended by ϕ_2 and ϕ_2 is extended by ϕ_1.

6.3 Class dictionary graph Extension Transformations

This section informally reviews the class dictionary graph extension transformations. The first five of these are the object-preserving transformations that were formally defined in

Chapter 4. The last three are not object-preserving and are presented here for the first time. Together, the eight primitive transformations comprise the **class dictionary graph extension transformations** as summarized below.

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 44–PRP to the class dictionary graph in Figure 43.

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 41 to Figure 44–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 interface from the classes RectTool, OvalTool, SelectTool to their common superclass CanvasTool (Figure 44–DCP to Figure 44–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 44–AUA to Figure 44–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

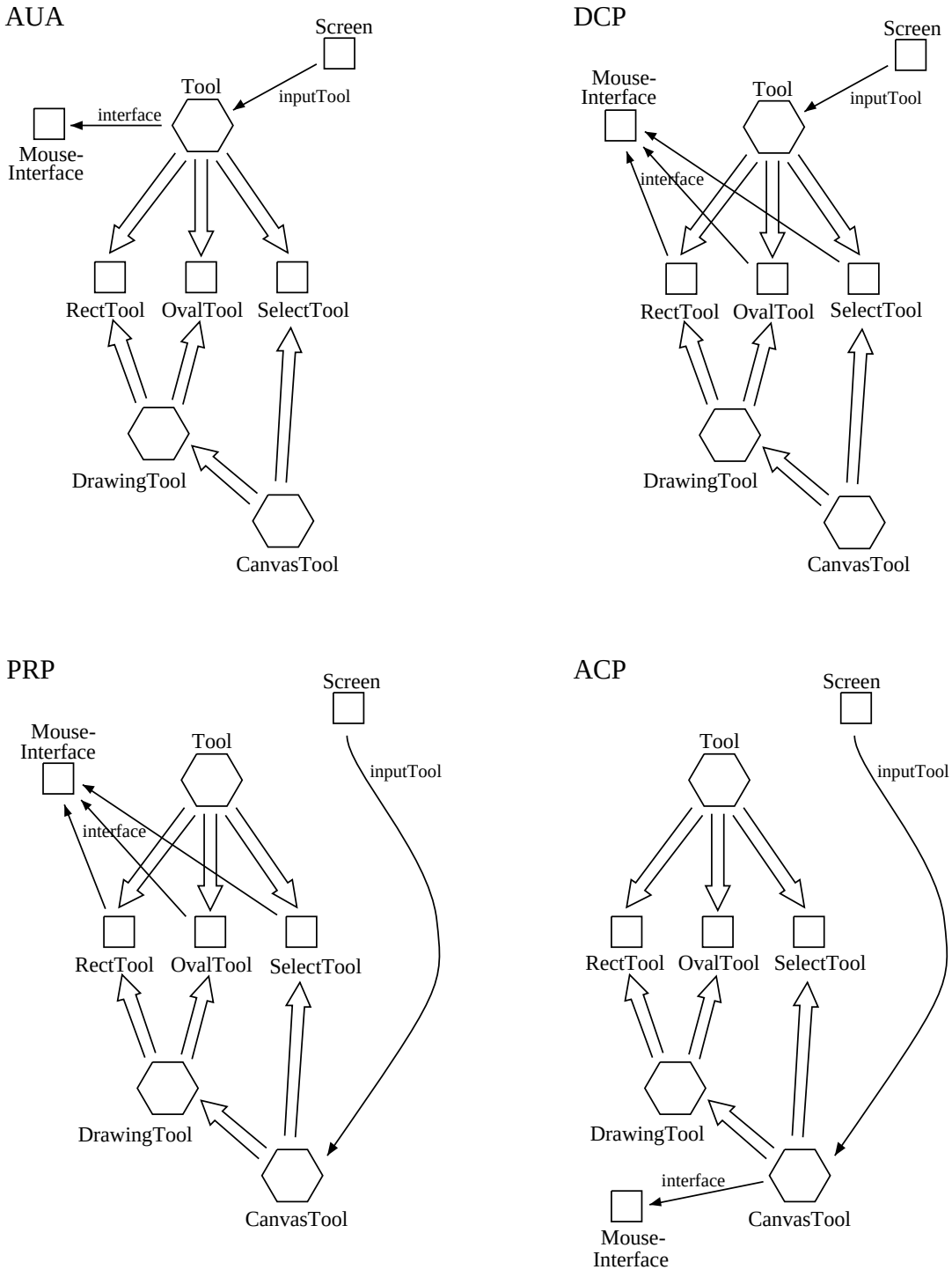


Figure 44: Steps in the object-preserving transformation

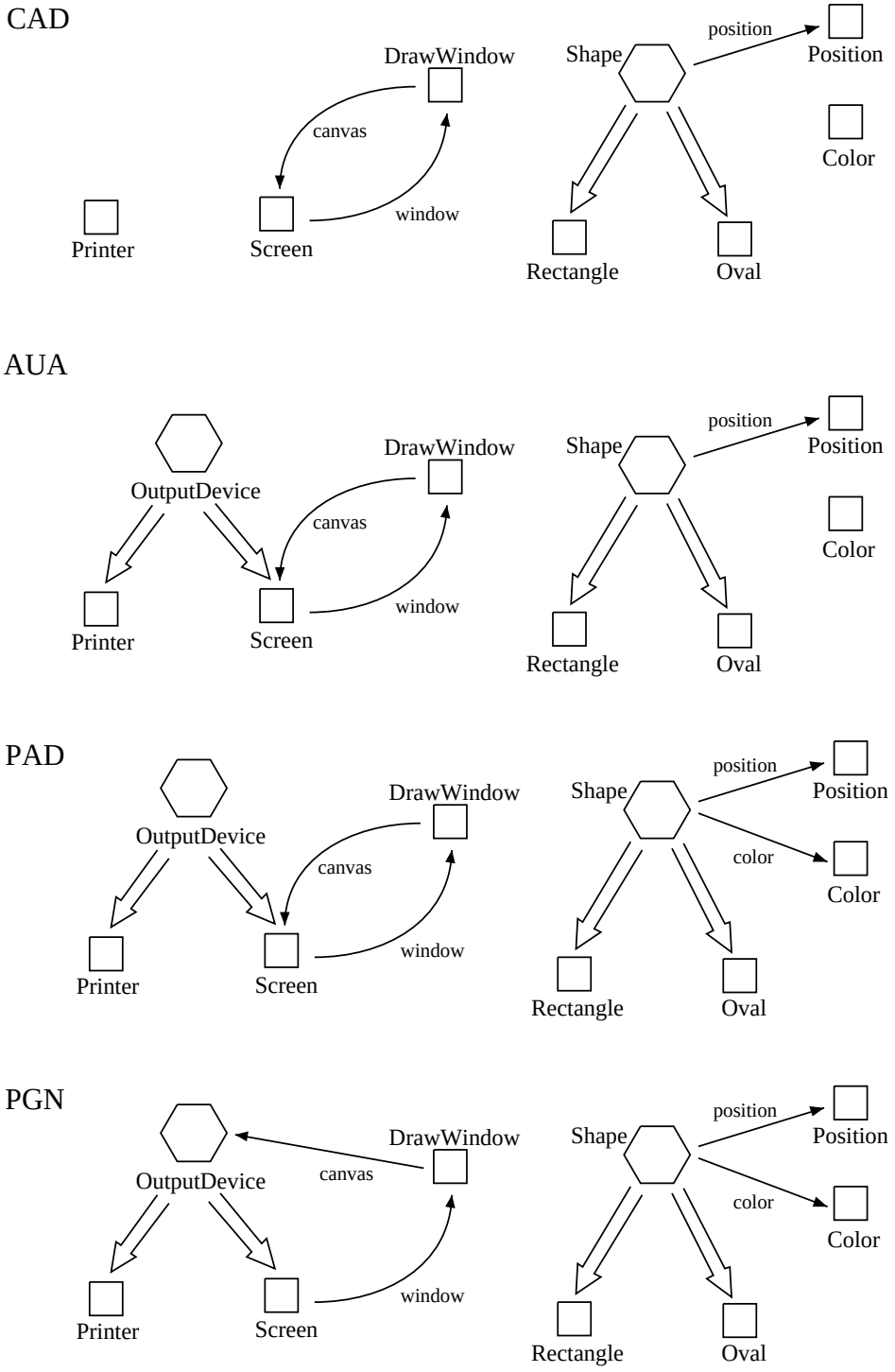


Figure 45: Steps in the object-extending transformation

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 44–ACP to Figure 44–PRP).

Class addition (CAD) CAD adds to the existing class dictionary graph a single new construction class with no incoming or outgoing edges. Examples of CAD are the addition of the classes Printer and Color to the class dictionary graph in Figure 42 as shown in the partially drawn class dictionary graph in Figure 45–CAD.

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 45–AUA to Figure 45–PAD).

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 45–PAD to Figure 45–PGN).

Each of the primitive transformations defines a relation on class dictionary graphs. Together, the eight primitive relations comprise the object-extension relation. The primitives have been shown to be correct, minimal and complete [Ber91, LHX94]. The completeness guarantees that for any two class dictionary graphs in an object-extending relation there exists a sequence of primitive transformations that transforms the original into the extended class dictionary graph. Since the completeness proofs are constructive, there also exists an algorithm to find the sequence. The primitive class dictionary graph transformations will be used in the subsequent section to determine their impact on the behavioral consistency of a program.

6.4 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

class dictionary graph. 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 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.

6.5 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 [LHX94] 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.

6.5.1 Untyped Language Model

In the untyped language model the code transformations are very simple. Consider the example of the transformation of the class dictionary graph in Figure 41 to the extended class dictionary graph in Figure 43.

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 44–AUA). The addition of these abstract classes does not require any modification of the code.

Distribution of common parts

In the next step (Figure 44–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.

Part replacement

In the next step, the part class of Screen's inputTool is changed from Tool to CanvasTool by part replacement (Figure 44–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.

Abstraction of common parts

When the part is moved up the new inheritance hierarchy to the CanvasTool class (Figure 44–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.

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 modeling point of view, since the class may have important methods attached. If the class is deleted to produce the class dictionary graph in Figure 42, 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 occurrence of call-next-method can be removed and the "next-method" defined in the "useless" class inlined in its place.

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 43) 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.

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 43), 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 6.4) after the class dictionary graph 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.

6.5.2 Typed Language Model

For the discussion of code transformations in the typed language model, illustrated for the example of C++, we assume that the code conforms to our data and language models. In particular, make the following simplifying assumptions: (1) All parts are defined as protected data members. (2) 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.

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.

Distribution of common parts

As we have seen, 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 methods that access a distributed part will no compile, since the part is not defined within the scope of the superclass.

To restore behavioral consistency, every superclass method that accesses the part must be distributed, along with the part, to each subclass where the method is not overridden. Since instances of the subclasses may be declared with the static type of the superclass, we must replace the original method with a "pure virtual" method so that messages to those instances will be understood.

Constructor and destructor methods in C++ may be treated much like the distribution of before and after methods in deletion of useless alternation classes in the untyped model since their behavior is similar. The body of a superclass constructor accessing a distributed part is inlined at the beginning of each subclass constructor and replaced with an empty body. The body of a superclass destructor accessing a distributed part is inlined at the end of each subclass destructor 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 44–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(); }
```

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, there are two problems that occur as a result of the implied change in the type declaration of the part. 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 a pure virtual function in the part's new class for each corresponding method defined in the part's old class. Since each of the instantiable (construction) subclasses now inherits both the original method and the "new" pure virtual method, it must supply its own method to resolve the ambiguity by calling 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. Unfortunately, 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(); }
```

Abstraction of common parts

As in the untyped model, no changes are required by abstraction of common parts. Any class that accesses a part that has been abstracted to a superclass will still have access to the part through inheritance since data members are defined to be protected in our language model.

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.

Methods (including constructors and destructors) attached to the useless alternation class, A, are distributed to subclasses in the same manner as when required by distribution of common parts. Since objects of static type A may be sent messages, we must either keep a class definition of A and attach pure virtual methods, or else find a substitute type. If there is an alternation class B with the same set of associated 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 member functions and no data members. We regard A as a type rather than as a class.

If anywhere in the program an explicit call is made through the scope resolution operator "::" to a method $A :: m$, 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.

As an example, consider what happens when the Tool class is deleted in the transformation from Figure 44–PRP to Figure 42. 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.

Class and part addition

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

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 so that messages sent to the generalized part will be understood. This is done by defining empty virtual functions in B wherever necessary. In this case we use an empty "default" method rather than a pure virtual method since subclasses of B other than C may not have suitable methods. Note that the part generalization indicates

that behavior extension (supplying methods for the other subclasses of B) is in order, but our goal is simply to ensure behavior preservation.

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.

The above transformation achieves the goal of preserving behavior, 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.

6.6 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 class dictionary graph 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.

6.7 Related work

Opdyke and Johnson [OJ90, Opd92] have investigated the *refactoring* of object-oriented systems for reuse. They define refactorings as restructuring plans which are useful in the iterative design of an application framework. As in our approach, refactorings preserve behavior and can be performed by applying a small set of basic refactorings.

The Integrity Consistency Checker (ICC) is a tool designed by Delcourt and Zicari [DZ91] which ensures structural consistency while performing schema updates. The ICC does not allow any update which would introduce structural inconsistency. However, it allows behavioral inconsistencies that do not result in run-time errors. Our methods, in contrast, guarantee behavioral consistency by automatically adapting programs to the transformed schema.

Chapter 7

Object grammars

In this chapter, the data model of Chapter 2 is extended so that a class dictionary graph may define not only a set of objects, but also a language for representing the objects textually. The concept of a "part" is extended to mean either another object or a token of concrete syntax. The parts are ordered and an object's textual representation is obtained by concatenation of the concrete syntax as it is encountered during a depth first traversal of the object's parts.

7.1 Extended data model

7.1.1 Extended class dictionary graphs

Definition 7.1. *An* **extended class dictionary graph**, ϕ, *is a directed graph* $\phi = (V, VS, \Lambda;\ EC, EA, ES, Ord)$ *where the definitions of* V, Λ, EC, *and* EA *are the same as for class dictionary graphs.* VS *is a set of strings called the syntax vertices.* ES *is a binary relation on* $V \times VS$ *called the syntax edges:* $(v \rightarrow w) \in ES$ *iff there is a syntax edge from* v *to* w. $Ord : (EC \cup ES) \rightarrow \mathcal{N}$ *is a function that maps each construction and syntax edge to a natural number.*

The set of vertices, V, is partitioned into construction and alternation vertices, VC and VA, respectively, for extended class dictionary graphs as it is for class dictionary graphs. Sometimes, it is more convenient to describe an extended class dictionary graph as a tuple which contains explicit references to VC and VA: $\phi = (VC, VA, VS, \Lambda; EC, EA, ES, Ord)$.

Graph	Object-oriented Design	Context Free Language
Vertex	**Class**	**Symbol**
construction	instantiable class with members defined by construction edges (including "inherited" edges)	Concatenation of languages defined by construction and syntax edges (including "inherited" edges)
alternation	abstract class with subclasses defined by alternation edges	union of languages defined by alternation edges
syntax	no meaning	terminal
Edge	**Class Relationship**	**Operator**
construction	part-of relationship, "uses", "knows", — labels are part names	concatenation — numbers define order
alternation	inheritance relationship, specialization, classification	union
syntax	no meaning	concatenation — numbers define order

Table 3: Extended Interpretation

A *legal* extended class dictionary graph must satisfy the the *cycle-free alternation condition* and the *unique labels condition*. Additionally, it must satisfy the *unique numbering condition*.

Definition 7.2. *The* **unique numbering condition** *states that:*
$\forall u, v, v' \in V$ *and* $e, e' \in (EC \cup ES)$ *where* $v \stackrel{*}{\Longrightarrow} u$, $v' \stackrel{*}{\Longrightarrow} u$, $e \neq e'$ *:*
If $\exists w, w', l, l'$ *such that* $e = (v \stackrel{l}{\longrightarrow} w)$ *or* $e = (v \rightarrow w)$, *and* $e' = (v' \stackrel{l'}{\longrightarrow} w')$ *or* $e' = (v' \rightarrow w')$ *then* $Ord(e) \neq Ord(e')$

The unique numbering condition is similar to the unique labels condition. It guarantees that the construction and syntax edges inherited at any vertex are totally ordered.

7.1.2 Extended object graphs

Object graphs are extended in similar fashion.

Definition 7.3. *An* **extended object graph**, ψ, *is a directed graph*
$\psi = (W, W_s, S, \Lambda_\psi;\ E, E_s, \lambda, Ord)$ *where the definitions of* W, S, Λ_ψ, E, *and* λ *are the same as for object graphs.* W_s *is a set of strings called the syntax vertices.* E_s *is a binary relation on* $W \times W_s$ *called the syntax edges.* $Ord : (EC \cup ES) \rightarrow \mathcal{N}$ *is a function that maps each edge to a natural number.*

Definition 7.4. *Let $p_1, p_2, ...p_n$ be the outgoing edges (including syntax edges) from a vertex, $v \in W$, of an extended object graph such that $Ord(p_i) < Ord(p_{i+1}), 1 \leq i < n$. Then the* $\boldsymbol{PartOrder(v, p_i) = i}$.

Let $q_1, q_2, ...q_n$ be the construction and syntax edges outgoing from all vertices, v', from which a vertex, $v \in VC$ of an extended class dictionary graph is alternation reachable, such that $Ord(q_i) < Ord(q_{i+1}), 1 \leq i < n$. Then the $\boldsymbol{PartOrder(v, q_i) = i}$.

Definition 7.5. *An extended object graph, $\psi = (W, W_s, S, \Lambda_\psi;\ E, E_s, \lambda, Ord)$, is* **legal** *with respect to an extended class dictionary graph, $\phi = (VC, VA, \Lambda;\ EC, EA)$, iff for each vertex, $v \in W$:*

- $\lambda(v) \in VC$
  Each vertex in the object graph maps to a construction vertex in the class dictionary graph.
- $\forall (r \xrightarrow{l} s) \in EC$ where $r \stackrel{\star}{\Longrightarrow} \lambda(v) : \exists w \in W$ such that $(v \xrightarrow{l} w) \in E$
  Each object has all of the sub-objects prescribed by the class dictionary graph.
- $\forall (r \rightarrow s) \in ES$ where $r \stackrel{\star}{\Longrightarrow} \lambda(v) : (v \rightarrow s) \in E_s$
  Each object has all of the concrete syntax prescribed by the class dictionary graph.
- $\forall w, l$ where $(v \xrightarrow{l} w) \in E : \exists (r \xrightarrow{l} s) \in EC$ such that $r \stackrel{\star}{\Longrightarrow} \lambda(v), s \stackrel{\star}{\Longrightarrow} \lambda(w)$, and $PartOrder(v, (v \xrightarrow{l} w)) = PartOrder(\lambda(v), (r \xrightarrow{l} s))$
  Each object has only *the sub-objects prescribed by the class dictionary graph and has them in the proper order.*
- $\forall s$ where $(v \rightarrow s) \in E_s : \exists r$ such that $(r \rightarrow s) \in ES, r \stackrel{\star}{\Longrightarrow} \lambda(v)$, and $PartOrder(v, (v \rightarrow w)) = PartOrder(\lambda(v), (r \rightarrow s))$
  Each object has only *the concrete syntax prescribed by the class dictionary graph and has it in the proper order.*

The language of an extended class dictionary graph ϕ, is formally defined in terms of *sentences* which are defined, in turn, by the extended object graphs which are legal with respect to ϕ.

Definition 7.6. *An acyclic extended object graph, ψ, rooted at some vertex, v, has a textual representation, called* $\boldsymbol{sentence(\psi)}$, *which is the string of syntax vertices encountered*

during a depth first traversal of ψ starting from v. If an extended object graph, ψ, is cyclic or unrooted, $sentence(\psi)$ is undefined.

We say that an extended object graph, $\psi = (W, W_s, S, \Lambda_\psi;\ E, E_s, \lambda, Ord)$, is **rooted** *at vertex v if v is the only vertex in W with an in-degree of 0.*

Definition 7.7. *The language defined by an extended class dictionary graph, $\phi = (V, VS, \Lambda;\ EC, EA, ES, Ord)$, is given by:*

$$L(\phi) = \{s | \exists \psi : sentence(\psi) = s, \text{ and } \psi \text{ is legal with respect to } \phi\}$$

7.2 Working with extended graphs

The theory and techniques of the preceding chapters are easily extended with the addition of syntax edges in object and class dictionary graphs. Syntax edges may generally be treated as though they were construction edges with implicit labels. The implicit label is the syntax vertex which is the target of the edge – uniquely labeling each syntax edge would defeat the abstraction of common syntax to alternation vertices; uniformly labeling all syntax edges would violate the unique labels condition. The ordering of parts introduces additional complexity, but it is easily handled as illustrated below for class dictionary graph learning and the object-preserving transformations.

7.2.1 Learning extended class dictionary graphs

The basic learning algorithm for extended graphs requires that the example object graphs satisfy two additional constraints:

- The ordering of the parts must be consistent between examples.
- The concrete syntax must be the same in each example (since the target of a syntax edge cannot be an alternation vertex).

The algorithm to translate a list of object example graphs $\psi_1, ..., \psi_n$ of the form $\psi_i = (W_i, W_{s_i}, S_i, \Lambda_{\psi_i};\ E_i, E_{s_i}, \lambda_i, Ord_i)$ into an class dictionary graph, $\phi = (V, VS, \Lambda;\ EC, EA, ES, Ord)$, is extended by adding three steps to learn the syntax vertices, the syntax edges, and the ordering function:

- $VS = \bigcup_{1 \le i \le n} W_{s_i}$

- $ES = \{(r \rightarrow s) | r \in VC, \exists v, i : (v \rightarrow s) \in E_{s_i}, \lambda_i(v) = r\}$
- $Ord = \{(r \rightarrow s, j) | \exists v, i : (v \rightarrow s) \in E_{s_i}, \lambda_i(v) = r, PartOrder(v, v \rightarrow s) = j\}$
  $\cup \{(r \xrightarrow{l} s, j) | \exists v, w, i : (v \xrightarrow{l} w) \in E_i, \lambda_i(v) = r, PartOrder(v, v \xrightarrow{l} w) = j\}$

The incremental algorithms are similarly extended.

7.2.2 Object-preserving transformations

The object-preserving transformations for extended class dictionary graphs are almost the same as the object-preserving transformations defined for ordinary class dictionary graphs in Chapter 4. For extended graphs, there is an additional primitive that allows edges to be renumbered as long as the ordering is unchanged. *Abstraction of common parts* and *distribution of common parts* are extended to apply to syntax edges as well as construction edges. The only additional complexity is that abstraction of common parts to a superclass is restricted so that the ordering of parts at each immediate subclass can not be changed. If there is a set of classes that have more than one part in common, but the common parts are ordered differently in the individual classes, then it is not possible to abstract all of the common parts.

The new definitions are as follows:

- **Renumbering of parts**. Any set of construction and syntax edges in a class dictionary graph, ϕ, may be renumbered (by replacing the Ord function) to produce a new class dictionary graph, ϕ', if for all vertices, $v \in V$, and edges, $e \in (EC \cup ES)$, such that v is alternation reachable from the source of e: $PartOrder_{\phi}(v, e) = PartOrder_{\phi'}(v, e)$.

- **Abstraction of common parts**. If $\exists v, w, l, i$ such that $\forall v'$, where $(v \Longrightarrow v') \in EA$: $(v' \xrightarrow{l} w) = e \in EC$ and $Ord(e) = i$, or $(v' \rightarrow w) = e \in ES$ and $Ord(e) = i$, then all of the edges, e, can be replaced by a new edge, e', with v as its source and $Ord(e') = Ord(e)$.

- **Distribution of common parts**. An outgoing construction edge, $e = (v \xrightarrow{l} w)$, or syntax edge, $e = (v \rightarrow w)$, can be deleted from an alternation vertex, v, if for each $(v \Longrightarrow v') \in EA$ a new construction edge $e' = (v' \xrightarrow{l} w)$, or syntax edge $e' = (v' \rightarrow w)$, respectively is added with $Ord(e') = Ord(e)$.

  This is the inverse of abstraction of common parts.

The definitions of *addition of useless alternation*, *deletion of useless alternation*, and *part replacement* are unchanged.

7.3 A Simple Object-oriented Programming Language

This section describes a simple object-oriented programming language based on extended class dictionary graphs. The CDG language will be used to illustrate the method transformations that are required to restore behavioral consistency after a language-preserving class dictionary graph transformation. Although the CDG language is very simple, the same principles can be used to modify programs written in "real" languages.

7.3.1 Overview

A CDG (class dictionary graph) program consists of a set of class definitions in the form of an extended class dictionary graph plus a set of method definitions. There is one built-in class called `Number` for which the language provides the built-in methods `add`, `sub`, `mul`, `div`, `assign`, and `print`. The user may define still more methods for `Number` class.

The class definitions must include a construction class called `Main`, and the method definitions must provide a `main` method for the `Main` class. Program execution begins by parsing the input by recursive descent to construct an instance of the `Main` class (the main object), and invoking its `main` method. In the CDG language, there is no way to create or destroy objects once the initial parsing operation is complete. Thus, the set of objects is fixed during program execution and consists of a tree rooted at the main object.

Each object except the main object has an implied "container" attribute. An object's container is its parent in the object tree. In other words, the attribute links between objects are defined to be bidirectional. This feature of the CDG language is included to simplify some of the code transformations discussed below. In a "real" language, the container attributes[1] would be implemented only when required by a code transformation.

In the CDG language, as in languages such as Smalltalk, each object has direct access only to its own attributes. However, in CDG these attributes include the container attribute. In other words, each object has access to its own parts and to the object of which it is a part.

[1] There may be more than one container for each object.

7.3.2 Methods, Messages, and Expressions

Each method may take any number of objects as arguments and every method returns an object. Both the arguments and the return value are passed by reference. This must be the case, since passing by value would involve the construction of new objects during program execution.

method ::= class : name '(' formals ')' '{' explist '}'
formals ::= name { , name }
This construct is a method definition. When the method is invoked by sending the *name* message to an object of class *class*, the expression *explist* is evaluated after argument values are substituted for the formals and its value is returned.

explist ::= exp { ; exp }
An expression list is evaluated by evaluating each expression in the list from left to right. The value of the list is the value of the last expression.

exp ::= name
Here, name may be either the name of a part of the object for which the method being evaluated was invoked or the name of a formal parameter of the method. The value of the expression is the object instantiating the part or the object passed as the actual argument, respectively.

exp ::= self
The value of the expression *self* is the object for which the method being evaluated was invoked.

exp ::= container
The value of the expression *container* is the object which contains the object for which the method being evaluated was invoked. The expression *container* must not appear in any method attached to the Main class.

exp ::= exp ← name '(' [actuals] ')'
actuals ::= exp { , exp }
This construct denotes the sending of a message. When an object is sent a message, the object's method called *name* is invoked. If the object has no method with the proper name, a run time exception occurs and program execution is terminated. The evaluation order is: The *exp* on the left hand side of the message send operator, ←, is evaluated; each

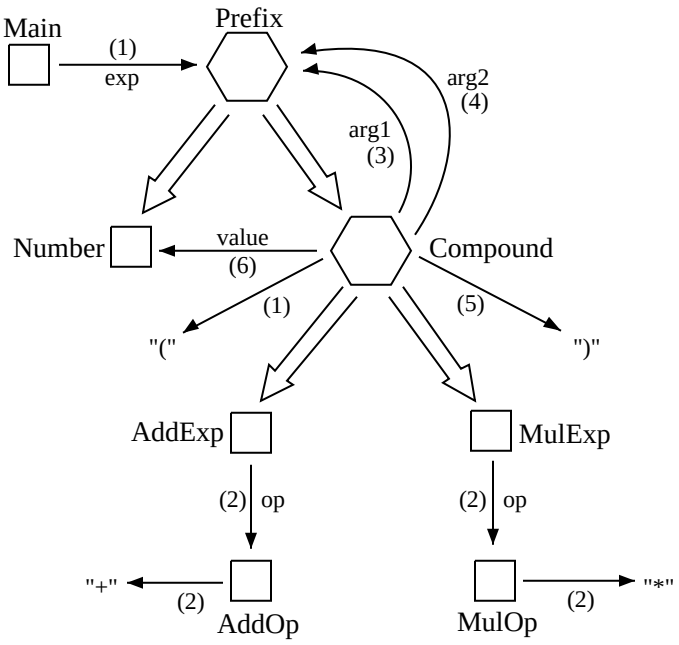
of the actual argument expressions is evaluated and their values are substituted for the corresponding formals in the method body; the method body is evaluated and the result is returned.

Note that the CDG language supports delayed binding but not inheritance of methods. I do not consider inheritance an important issue in the study of code transformations since it can easily be eliminated from an object-oriented program just by copying methods from superclasses to the classes where they are inherited. The inheritance mechanism is merely a convenience for the programmer so that each method only needs to be written in one place.

7.3.3 Built-ins

All of the built-in methods for the `Number` class except `print` take a single argument which must be another `Number`. All of the methods return `self`. A side effect of the methods `add`, `sub`, `mul`, `div`, and `assign` is that the "value" of the `Number` object receiving the message is changed. That is, the state of the object is changed in such a way that subsequent messages to the object may have different results. The value is modified in the obvious way depending on whether the message is `add`, `sub`, `mul`, `div`, or `assign`. When a `Number` is created during the initial parsing, its value is initialized depending on the value denoted by the token parsed. When a `Number` receives the `print` message, a token denoting its value is output.

Example 7.1. *A complete CDG program to evaluate arithmetic prefix expressions is shown in Figure 46.*




```
Main : main ()
{ exp <- eval() <- print() }

Number : eval ()
{ self }

AddExp : eval ()
{
   value <- assign(arg1 <- eval());
   value <- add(arg2 <- eval())
}

MulExp : eval ()
{ value <- assign(arg1 <- eval()) <- mul(arg2 <- eval()) }
```

Figure 46: Program A

Chapter 8

Program Evolution by Analogy

In his classic paper from 1971, *A Paradigm for Reasoning by Analogy*, Kling [Kli71, page 147] states that "a problem solver designed in any of the contemporary paradigms ...solves the same problem the same way each time it is presented. *A fortiori*, it is unable to exploit similarities between new and old problems to hasten the search for a solution to the new one". Although analogical reasoning has been heavily investigated in recent years [Hal89], the research of the last two decades has been almost exclusively concentrated on traditional artificial intelligence problems: automated deduction, general problem solving and planning, natural language processing, and learning. There are still many areas of mathematics and computer science where analogical reasoning has been largely neglected.

One potential application of analogical reasoning (AR) in computer science is in the automatic synthesis of programs. When a new program is required which should provide functionality that is different, but somehow *analogous*, to the functionality of an existing program in the same domain, we might expect the application of analogical reasoning techniques to produce a solution. In practice, this approach has presented some very difficult problems and has met with only modest success ([DM77], [UM77], [Pöt87], [IUT87]).

There is, however, another common situation where AR might be applied to advantage in the automatic synthesis of programs. This is the situation where functionality *identical* to the functionality of an existing program is required, but in a different, *analogous*, domain or environment. For example, in object-oriented systems it is sometimes desirable to rearrange the class structure, but this usually has the undesirable effect of requiring programs to be manually rewritten. This is an important issue in software maintenance, where analogical program synthesis to automate the update of code when the application

environment changes could have a major impact.

This form of analogical reasoning is also interesting from an implementation point of view. The ability to automatically reproduce some of the functionality of an old application in a new environment would be an important contribution in the area of software reuse.

While previous efforts to apply analogical programming to these reuse and maintenance issues in software engineering have met with limited success, the added structure imposed by the object-oriented paradigm could simplify some of the difficult issues and provide an ideal environment for the application of structure-mapping theory, [Gen83] [GT86]. The grammar-based environment of the Demeter Model [LBSL91] supplies a link between the functionality of a program and the class structure by defining a language for inputs and outputs — provided that the inputs and outputs are *objects*.

8.1 Problem Description

The primitive transformations defined in [Ber91] can be used to automate the object-preserving optimization of a given class structure. Furthermore, it is easy to automatically determine whether two arbitrary class structures are object-equivalent, and, if so, produce a sequence of primitive transformations from one to the other. Given such a sequence of primitive transformations, the code can be automatically updated in incremental fashion, following simple update rules for each of the primitives in sequence [Ber93].

Updating the code after an object-preserving class reorganization is a relatively simple problem because although the *class* definitions change, the set of *objects* defined by the classes remain the same. While the environments might be considered *analogous*, the objects (program domains) are *identical*, and analogical reasoning is not required.

Since an extended class dictionary graph defines both a class structure and a grammar, any change in the class structure is reflected in a corresponding change in the grammar. There is an interesting class of transformations that result in a new class structure, a potentially new set of objects, and a new grammar, but which leave the *language* defined by the grammar unchanged. I call such a transformation *language-preserving* and say that the old and new class dictionary graphs are *language-equivalent*. For example, the class dictionary graphs in Figure 47 are language-equivalent but not object-equivalent. Furthermore, they are not in an extension relationship, and cannot be brought into an extension relationship by simple renaming of classes.

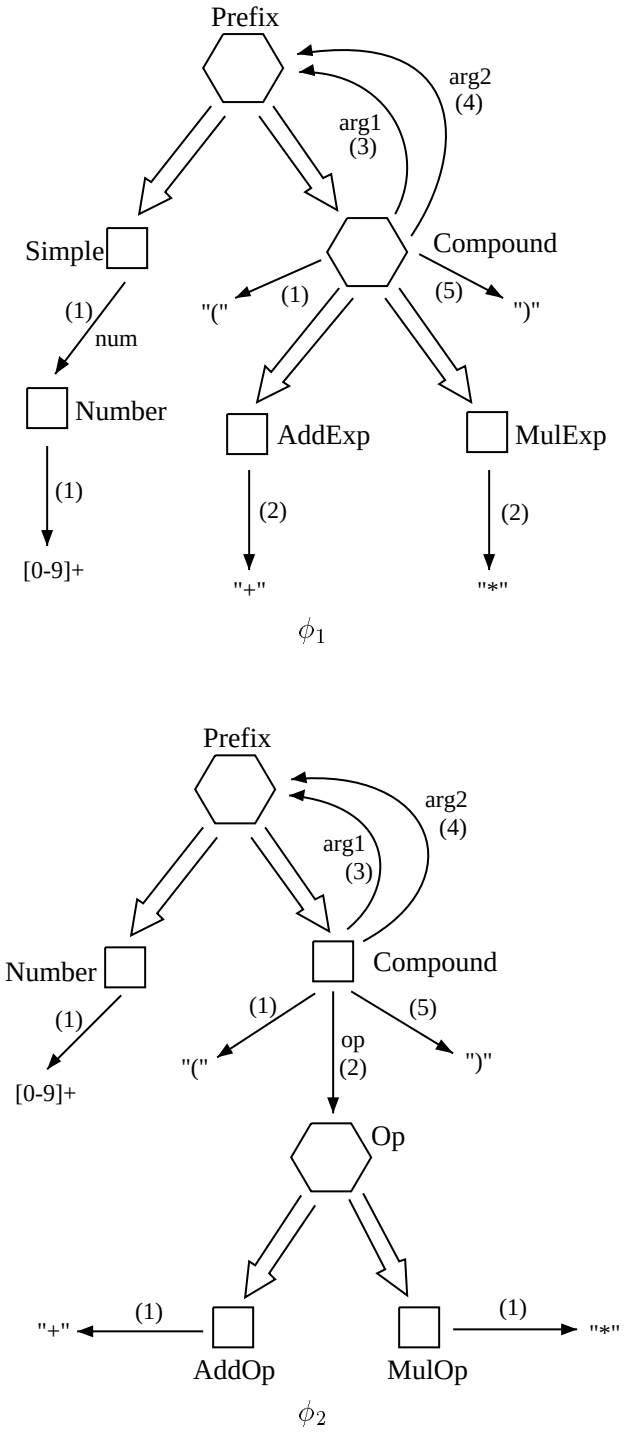


Figure 47: Language-equivalent Class Dictionary Graphs

If the inputs and outputs of a program are objects, then it is reasonable to expect that the code could be automatically updated after a language-preserving transformation so that identical inputs will produce identical outputs. More generally, if two class dictionaries define a common sub-language (i.e. the intersection of the two languages is not empty) then it is reasonable to expect that a program written for one of the class dictionaries could be automatically transformed into a program for the other class dictionary in such a way that any input in the sub-language will produce output identical to the output from the original program.

I call this problem: "Analogical Program Synthesis in a Grammar-Based Environment". The problem is motivated in part by a rewrite of the Demeter system's class dictionary, which required the entire system to be manually ported to the new environment by rewriting all the code. This was true even though the languages defined by the class dictionaries were nearly identical, and the functionality of the programs comprising the system was unchanged.

The problem is more formally stated as follows:

Instance:

Two extended class dictionary graphs, such that the intersection of the languages defined by the class dictionary graphs is non-empty. Also, a program written for the environment of the first class dictionary graph, whose inputs are sentences in the language defined by the class dictionary graph (i.e., the inputs are textual representations of objects). Without loss of generality, the outputs may also be restricted to sentences in the language defined by the class dictionary graph.

Problem:

Find a program for the environment of the second class dictionary graph that is functionally analogous to the given program for the first class dictionary graph. The solution program should produce output identical to the output of the given program on any input which is in the language of both class dictionary graphs.

There are a number of interesting special cases of this problem. For example, the two class dictionaries may define the same language, that is, they may be "language-equivalent". Another special case is where one of the languages is a proper superset of the other, that is, it is a "language-extension". In this chapter, the "language-equivalent" case is examined. In particular, we study CDG programs which have the same input languages. Note that even

if two class dictionaries are language-equivalent, they may define entirely different classes. The more general problems are left for future work.

Definition 8.1. *The* **input language** *of a CDG program, P, with class dictionary graph, ϕ, is given by:*

$$L(P) = \{s | \exists \psi : s = sentence(\psi), \psi \in Objects(\phi), \text{ and } \psi \text{ is rooted at } Main\}$$

The approach taken in this section is the same as was used for studying the object-preserving and object-extending transformations. In this approach the problem is broken down into three manageable sub-problems:

1. **Defining a set of primitive transformations.** In the case of the object-preserving transformations it was possible to find a small set of primitives which was provably complete [Ber91]. For the current case, it is more important that the set of primitives be *useful* than complete. That is, it should be possible to express the kinds of language-preserving transformations which would make sense from a software design point of view as a sequence of primitives.

   The transformations allowed by the primitives considered in this chapter include:

   - The object-preserving transformations
   - Renaming of vertices and edges
   - Addition and deletion of useless symbols
   - Distribution of parts up or down the part-of hierarchy
   - Replacing subclasses with attributes or attributes with subclasses

2. **Providing algorithms for incrementally updating the code.** For each primitive transformation an algorithm must be found for updating the code. Then, given any sequence of primitive transformations, the code can be updated incrementally by performing updates for each of the primitives in sequence.

3. **Reducing an arbitrary language-preserving transformation to a sequence of primitives.** An algorithm to search for a sequence of primitive transformations between two arbitrary language-equivalent class dictionaries must be found. Since a sequence of primitives defines an analogy, this is an algorithm for searching for

an analogy between two language-equivalent class structures. The search may be effectively guided by the concrete syntax of the language.

It is likely that there will be more than one sequence of primitives that can result in the same overall transformation. Each such sequence may represent a slightly different analogy. Depending on which analogy is used, different programs may result, but as long as the rules for updating the code are correct, all of the possible programs should be functionally equivalent. That is, they should all produce the same output for any given input.

8.2 Transforming a CDG Program

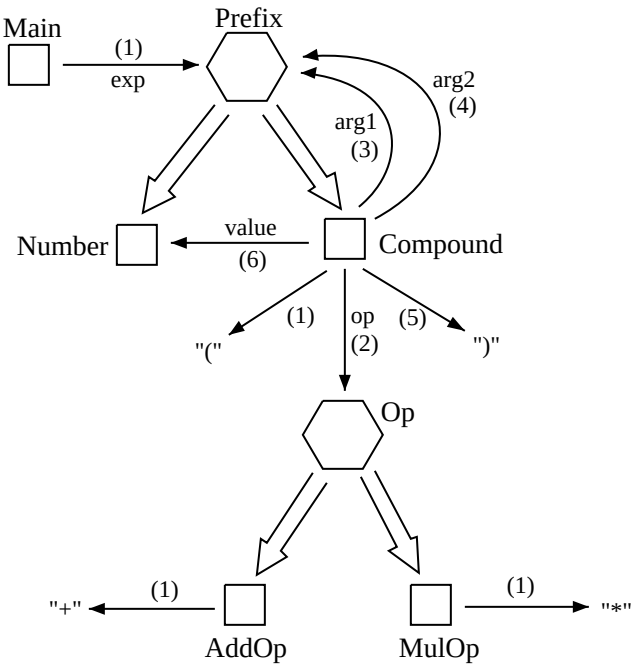
Consider the CDG program in Figure 48[1] which is functionally equivalent to the CDG program for evaluating prefix expressions shown in Figure 46. Notice that there is no object-extending relation (see [Ber93]) between the class dictionary graphs and that simple renaming of classes cannot bring them into such a relation. Still, the two class dictionary graphs define the same language, and the programs accept the same inputs.

The class dictionary graph in Figure 46 has been transformed to the class dictionary graph in Figure 48 by addition of useless alternation (`Op`) followed by the primitive language-preserving transformation I call "subclass-to-attribute". In the original class dictionary graph, the class `Compound` is abstract, with two concrete subclasses, `AddExp` and `MulExp`. When a `Compound` is parsed at program initialization, an instance of either `AddExp` or `MulExp` is created, depending on whether a "+" or "*" is found in the input stream.

In the transformed class dictionary graph, the `Compound` class is concrete; there is only one *kind* of `Compound`. However, each instance has an `op` attribute (part) which can hold an `AddOp` or `MulOp` value, so the information content is the same. When a `Compound` is parsed, an instance of a `Compound` is created. This involves parsing the `op` part of the `Compound` to produce either an `AddOp` or `MulOp` depending on whether a "+" or "*" is found in the input stream. Wherever an object had an `AddExp` or `MulExp` as a part in the original program, the corresponding object will have a `Compound` with either a `AddOp` or `MulOp`, respectively, in the transformed program.

In order to maintain functional equivalence, it is necessary for a `Compound` object in

[1] Program B requires accessor methods for the `value`, `arg1`, and `arg2` parts of the `Compound` class which are not shown in Figure 48.




```
Main : main ()
{ exp <- eval() <- print() }

Number : eval ()
{ self }

Compound : eval ()
{ op <- eval(self) }

AddOp : eval (e)
{
   e <- value() <- assign(e <- arg1() <- eval());
   e <- value() <- add(e <- arg2() <- eval())
}

MulOp : eval (e)
{ e <- value() <- assign(e<-arg1()<-eval()) <- mul(e<-arg2()<-eval()) }
```

Figure 48: Program B

the transformed program to pass along any message it receives to its `op` part with the extra argument `self`. Methods written for the original program are mapped from `AddExp` → `AddOp` and from `MulExp` → `MulOp`. The only modification to the methods is that they must access any parts defined for `Compound` objects indirectly through the extra argument. Methods to allow access to these parts are added to the `Compound` class.

8.3 Primitive Language-Preserving Transformations

The following primitives comprise the language-preserving class dictionary graph transformations:

1. The object-preserving transformations
2. Renaming of vertices and edges
3. Nesting of parts
4. Unnesting of parts
5. Addition of lambda parts
6. Deletion of lambda part
7. Addition of lambda alternative
8. Deletion of lambda alternative
9. Insertion of singleton construction
10. Deletion of singleton construction
11. Attribute to subclass
12. Subclass to attribute

8.3.1 Object-preserving transformations

The object-preserving transformations for extended class dictionary graphs are defined in Chapter 7.

8.3.2 Renaming of vertices and edges

Any construction edge, $(v \xrightarrow{l} w) \in EC$ may be replaced by a construction edge with a different label, $(v \xrightarrow{l'} w)$. Also, any construction vertex, $v \in VC$, may be replaced by a different construction vertex, v', with the same incoming and outgoing edges. When viewing a picture of a class dictionary graph it appears that the vertex has been "renamed" by changing its label. Since the labels or identifiers used to denote construction vertices have a global scope, and the same identifiers may be used to denote vertices in other class dictionary graphs, a changed label implies a changed vertex. On the other hand, since the identifiers used to denote alternation vertices have a scope local to the class dictionary graph, changing the labels of alternation vertices may be done freely, but does not in any way "transform" the class dictionary graph.

8.3.3 Nesting of parts

Given a vertex, $w \in V$ with no incoming alternation edges and a different vertex, $u \in (V \cup VS)$, such that for every construction edge, e_v, from some vertex, v, to some vertex w' where $w \stackrel{*}{\Longrightarrow} w'$, there is a syntax or construction edge, e'_v, from v to u and w' has at most one incoming alternation edge, then:

- If for each v, $PartOrder(v, e'_v) = PartOrder(v, e_v) + 1$, then we may delete each edge, e'_v, and add a single replacement edge, e, from w to u and let $Ord(e) > Ord(e')$ for all other construction and syntax edges, e' outgoing from any w' where $w \stackrel{*}{\Longrightarrow} w'$.

  Intuitively, if every class which has w as a part has u as a part immediately after w, then we may remove the u part from all of those classes and instead make u the last part of class w. See, for example, Figure 49.

- If for each v, $PartOrder(v, e'_v) = PartOrder(v, e_v) - 1$, then we may delete each edge, e'_v, and add a single replacement edge, e, from w to u and let $Ord(e) < Ord(e')$ for all other construction and syntax edges, e' outgoing from any w' where $w \stackrel{*}{\Longrightarrow} w'$.

  Intuitively, if every class which has w as a part has u as a part immediately before w, then we may remove the u part from all of those classes and instead make u the first part of class w.

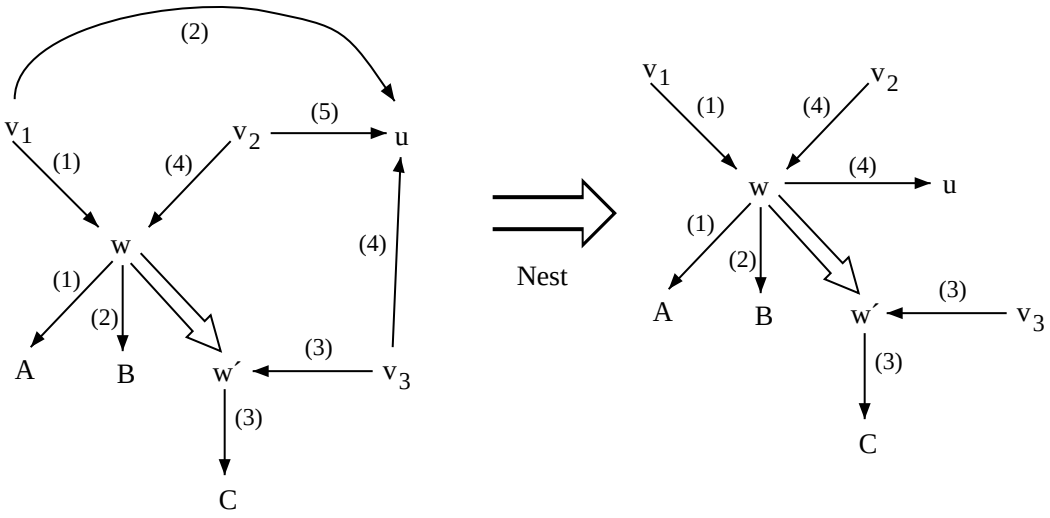


Figure 49: Nesting of parts

8.3.4 Unnesting of parts

Given a vertex, $w \in V$ with no incoming alternation edges and an outgoing construction edge or syntax edge, e, with target u:

- If for every construction or syntax edge, $e' \neq e$, with source w' such that $w \stackrel{*}{\Longrightarrow} w'$, w' has at most one incoming alternation edge and $Ord(e) > Ord(e')$ (so e is the last part of every w object), then we may delete edge e, if for each construction from some vertex, v, to w, e_v, we add a replacement edge, e'_v, from v to u such that $PartOrder(v, e'_v) = PartOrder(v, e_v) + 1$. In other words, we remove the last part, p, from every w object, and insert the part p just after the w part of every object that *contains* a w object.

  or

- If for every construction or syntax edge, $e' \neq e$, with source w' such that $w \stackrel{*}{\Longrightarrow} w'$, w' has at most one incoming alternation edge and $Ord(e) < Ord(e')$ (so e is the first part of every w object), then we may delete edge e, if for each construction from some vertex, v, to w, e_v, we add a replacement edge, e'_v, from v to u such that $PartOrder(v, e'_v) = PartOrder(v, e_v) - 1$. In other words, we remove the first part, p, from every w object, and insert the part p just before the w part of every object that *contains* a w object.

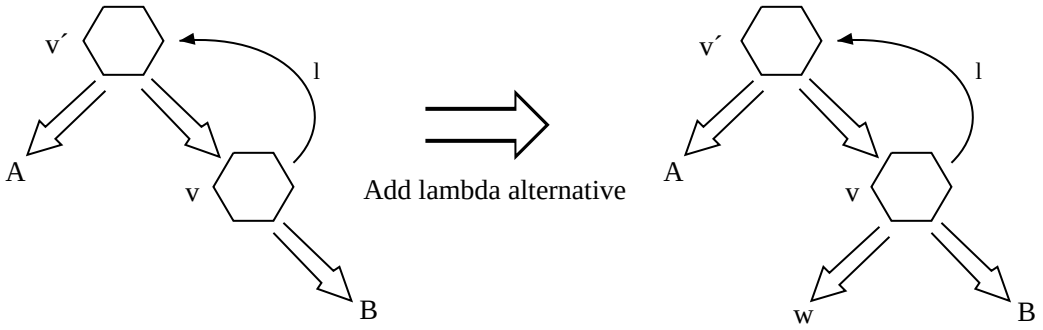


Figure 50: Addition of lambda alternative

This is the inverse of *nesting of parts*.

8.3.5 Addition of lambda parts

From any vertex, $v \in V$, an outgoing construction edge, $(v \xrightarrow{l} w)$ to a construction vertex, $w \in VC$, may be added if w has no outgoing edges. An outgoing syntax edge, $(v \rightarrow w)$ to a syntax vertex, w, may be added if w is the "empty string".

8.3.6 Deletion of lambda parts

A construction edge whose target is a construction vertex with no outgoing edges, or a syntax edge whose target is the "empty string" may be deleted. This is the inverse of *addition of lambda parts*.

8.3.7 Addition of lambda alternative

An alternation edge, $(v \Longrightarrow w)$, may be added from an alternation vertex, $v \in VA$ to a construction vertex, $w \in VC$ if w has no outgoing edges, v has only one outgoing construction edge, $(v \xrightarrow{l} v')$, and the target, v', of that edge has an outgoing alternation edge, $(v' \Longrightarrow v)$, back to v. See, for example, Figure 50.

8.3.8 Deletion of lambda alternative

An alternation edge, $(v \Longrightarrow w)$, from a vertex, $v \in VA$ to a construction vertex, $w \in VC$, may be deleted if w has no outgoing edges, v has only one outgoing construction edge, $(v \xrightarrow{l} v')$, and the target, v', of that edge has an outgoing alternation edge, $(v' \Longrightarrow v)$, back to v. This is the inverse of *addition of lambda alternative*.

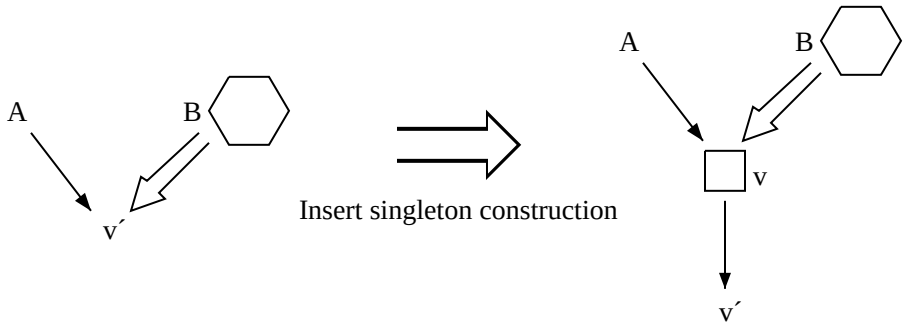


Figure 51: Insertion of singleton construction

8.3.9 Insertion of singleton construction

A new construction vertex, v, with a single outgoing construction edge to a vertex, $v' \in V$, may be added to a class dictionary graph, and any incoming construction edges at v' may be rerouted to v. Incoming alternation edges at v' may also by rerouted to v if the rerouting does not result in the inheritane of additional parts (syntax or construction edges) at v. See, for example, Figure 51.

8.3.10 Deletion of singleton construction

If a class dictionary graph contains a construction vertex, v, with no inherited parts and a single outgoing edge to a vertex, $v' \in (V \cup VS)$, then v may be deleted if all incoming edges at v are rerouted to v'. This is the inverse of *insertion of singleton construction.*

8.3.11 Attribute to subclass

If a class dictionary graph contains a construction vertex, $v \in VC$, with an outgoing construction edge, $(v \xrightarrow{l} w)$, to an alternation vertex, $w \in VA$, then we may delete the construction edge from v to w and for each vertex, w', such that $(w \Longrightarrow w') \in EA$, we add a new construction vertex, v', with an incoming alternation edge from v, $(v \Longrightarrow v')$, and an outgoing construction edge, $(v' \xrightarrow{l} w')$ to w'. Each of the new construction edges is mapped to the same number (under *Ord*) as was the deleted construction edge. Since v now has outgoing alternation edges it becomes (by definition) an alternation vertex. See, for example, Figure 52.

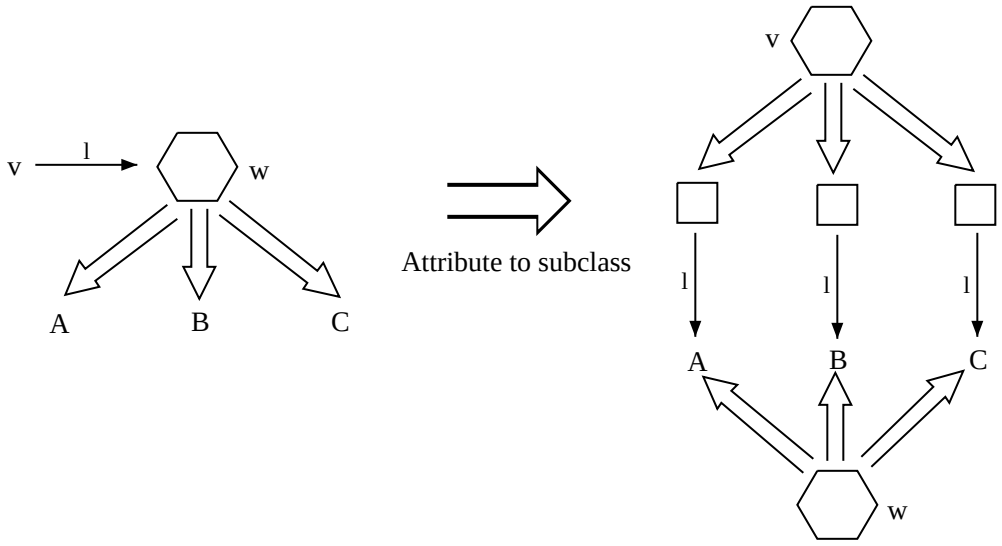


Figure 52: Attribute to subclass

8.3.12 Subclass to attribute

If a class dictionary graph contains alternation vertices, $v, w \in VA$, such that there is a one to one correspondence between the vertices, v', where $(v \Longrightarrow v') \in EA$ and the vertices, w' where $(w \Longrightarrow w') \in EA$, such that for each w' the corresponding v' is a construction vertex with a single incoming edge, $(v \Longrightarrow v')$, and a single outgoing edge, $(v' \stackrel{l}{\longrightarrow} w')$, then we may delete each such v' along with its incoming and outgoing edges and add a new construction edge, $(v \stackrel{l}{\longrightarrow} w)$, from v to w. Since v no longer has any outgoing alternation edges it becomes (by definition) a construction vertex. This is the inverse of *attribute to subclass*.

8.4 Justification for the primitive transformations

One justification for the selection of the chosen primitives is that they make it possible to express commonly occurring language-preserving transformations as a sequence of primitives. Examination of the literature and personal experience with the evolution of the Demeter system indicate that the primitive transformations defined in this chapter are powerful enough to express most, if not all, of the transformations that could be considered language-preserving. Rather than argue the subjective "practical usefulness" of the transformations, however, we demonstrate that the primitive language-preserving transformations defined

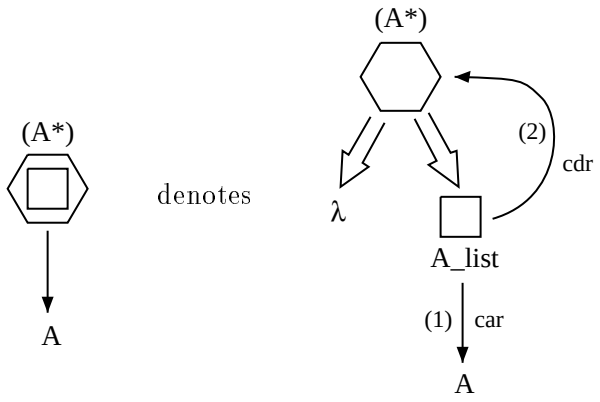


Figure 53: Closure operator

here form the basis of a *complete* transformation system for a subset of class dictionary graphs powerful enough to express the regular languages.

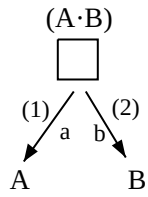
8.4.1 Regular class dictionary graphs

The *regular* class dictionary graphs are defined so that there is a bijection between them and the regular expressions. The alphabet of a regular class dictionary graph consists of the *terminal construction vertices* (those with a single outgoing syntax edge and no outgoing construction edges). Non-terminal construction vertices are concatenation ($\cdot$) operators and alternation vertices are union ($+$) operators. The closure ($*$) operator can be expressed using a cycle combining alternation and construction vertices. For convenience, we introduce a symbol (see Figure 53) for the $*$ operator and the symbol, λ, for the terminal construction vertex with the "empty string" as the target of its syntax edge.

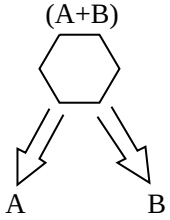
Definition 8.2. *Let VS be the set of all syntax vertices. Then the* **regular** *class dictionary graphs and their bijection with the regular expressions are defined recursively as follows.*

1. 

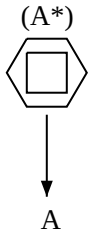
for all $A \in VS$

2. 

for regular subgraphs A and B

3. 

for regular subgraphs A and B

4. 

for regular subgraph A

Since there is a bijection between the regular expressions and the regular class dictionary graphs, we sometimes denote a regular class dictionary graph by its corresponding regular expression in the following discussion.

8.4.2 Axiom systems for regular expressions

It is well known that using substitution as the only rule of inference, there is no finite set of equations over the regular expressions that allows the derivation of a regular expression, α, from a regular expression α' iff α and α' define the same language. That is, there is no finite complete set of axioms in the algebra of regular expressions [Sal69]. However, the following system, $\mathcal{F}$, due to Salomaa [Sal69] is complete if an additional rule of inference is allowed:

$$(\alpha + (\beta + \gamma)) = ((\alpha + \beta) + \gamma)$$

$$(\alpha \cdot (\beta \cdot \gamma)) = ((\alpha \cdot \beta) \cdot \gamma)$$

$$(\alpha + \beta) = (\beta + \alpha)$$

$$(\alpha \cdot (\beta + \gamma)) = ((\alpha \cdot \beta) + (\alpha \cdot \gamma))$$

$$((\alpha + \beta) \cdot \gamma) = ((\alpha \cdot \gamma) + (\beta \cdot \gamma))$$

$$(\alpha + \alpha) = \alpha$$

$$(\alpha \cdot \lambda) = \alpha$$

$$(\alpha \cdot \phi) = \phi$$

$$(\alpha + \phi) = \alpha$$

$$(\alpha*) = (\lambda + (\alpha \cdot (\alpha*)))$$

$$(\alpha*) = ((\lambda + \alpha)*)$$

The additional rule of inference is *solution of equations*: If β does not possess the "empty word property" then the equation $\alpha = (\gamma \cdot (\beta*))$ may be inferred from the equation $\alpha = ((\alpha \cdot \beta) + \gamma)$.

Definition 8.3. *A regular expression, α, (or its corresponding regular class dictionary graph) has the* **empty word property** *(e.w.p.) iff one of the following holds:*

1. $\alpha = \lambda$
2. $\alpha = (\beta*)$ *for any* β
3. $\alpha = (\beta + \gamma)$ *where* β *or* γ *has the e.w.p.*
4. $\alpha = (\beta \cdot \gamma)$ *where* β *and* γ *have the e.w.p.*

8.4.3 Completeness proof

In this section, we show that for each equation over the regular expressions which is a substitution instance of an axiom in the complete system $\mathcal{F}$, there is a sequence of primitive transformations to transform the regular class dictionary graph corresponding to the left hand side of the equation to the class dictionary graph corresponding to the right hand side. Since every primitive transformation has an inverse, it is also possible to transform the right hand side to the left hand side. In other words, if it is possible to substitute a regular expression, α, for a regular expression α' in $\mathcal{F}$, then it is possible to transform the regular class dictionary graph, α' to α with the primitive transformations. Any regular expression, α, can be derived from any language-equivalent regular expression, α', in $\mathcal{F}$, so it must be possible to transform any regular class dictionary graph to any language-equivalent

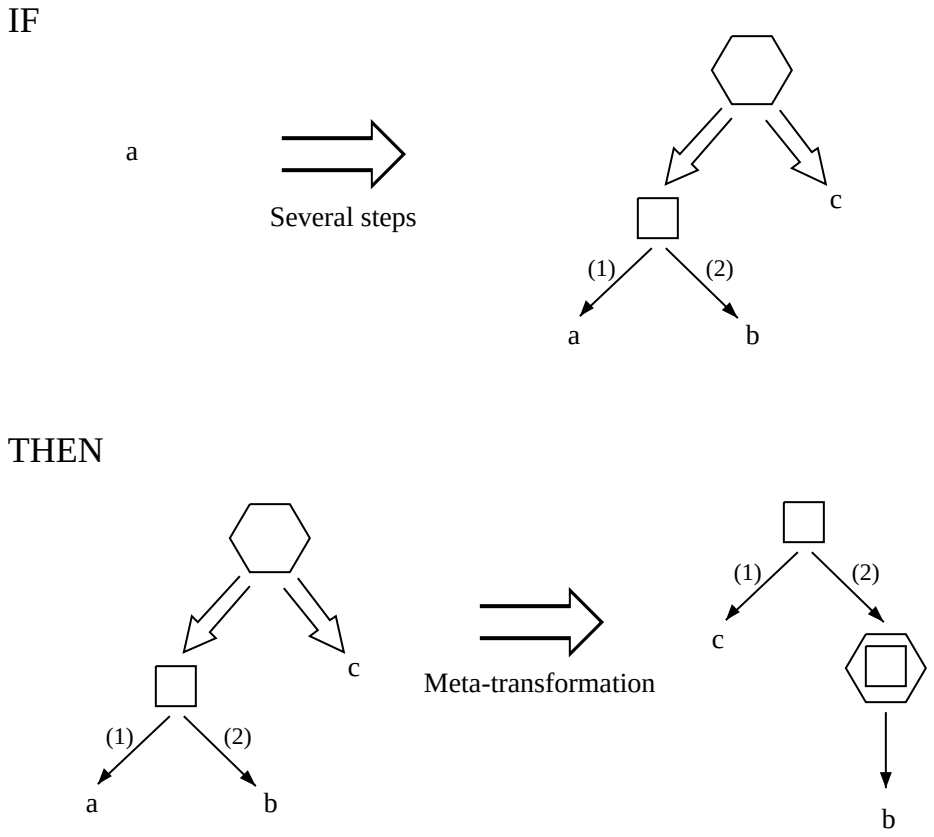


Figure 54: Solution of equations

regular class dictionary graph by a sequence of primitive transformations if the following meta-transformation (Figure 54), corresponding to the extra rule of inference, solution of equations, is allowed:

- If a regular class dictionary graph, ϕ, contains a subgraph, $((\alpha \cdot \beta) + \gamma)$, which was obtained by a sequence of primitive transformations from α, and β does not possess the empty word property, then $((\alpha \cdot \beta) + \gamma)$ may be replaced with the subgraph $(\gamma \cdot (\beta*))$.

The meta-transformation may prove difficult to apply in practice and is not considered one of the primitives. It is included here only to demonstrate that the primitive transformations themselves form a system as complete as any known system for regular expressions.

Figures 55 - 63 show sequences of primitive transformations that correspond to each of the equations in the axiom system, $\mathcal{F}$, except for the equations $(\alpha \cdot \phi) = \phi$ and $(\alpha + \phi) = \alpha$ which are not applicable since the regular class dictionary graphs as defined here do not

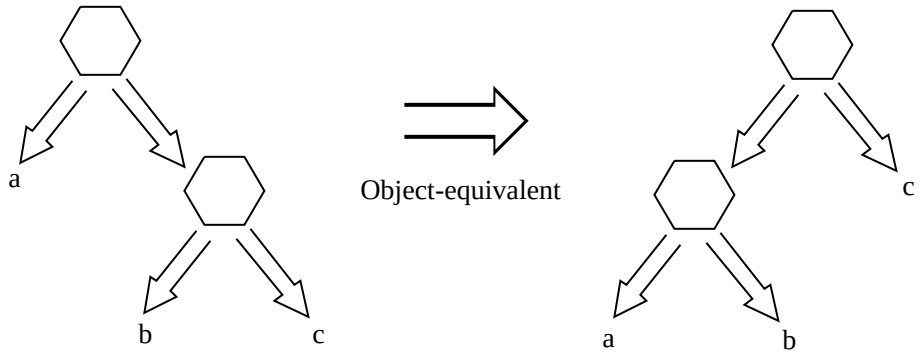


Figure 55: $(a + (b + c)) = ((a + b) + c)$

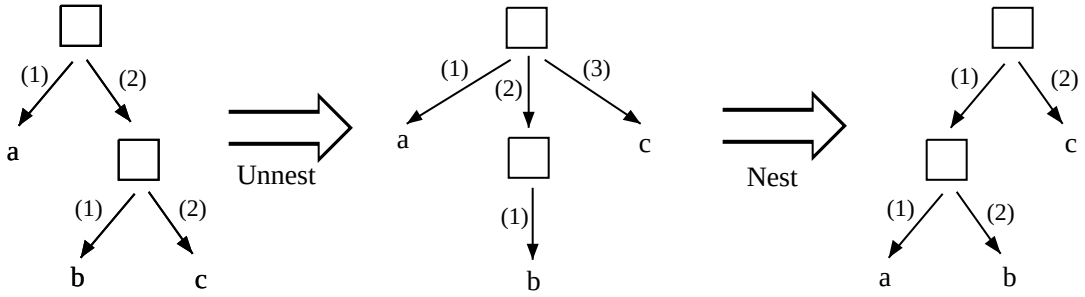


Figure 56: $(a \cdot (b \cdot c)) = ((a \cdot b) \cdot c)$

contain the empty language, ϕ.

8.5 CDG Program Transformations

In this section, code update rules for programs written in the CDG language are given for each of the primitive language preserving transformations. The rules are relatively straight forward since CDG is untyped, and since the objects in a CDG program always form a tree at runtime. Some of the code transformations require that an object have access to

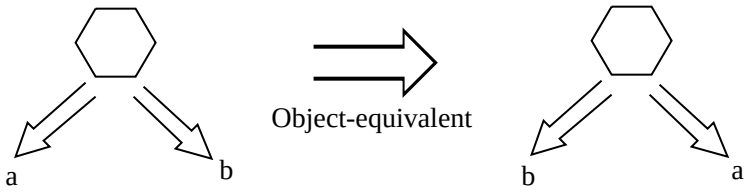


Figure 57: $(a + b) = (b + a)$

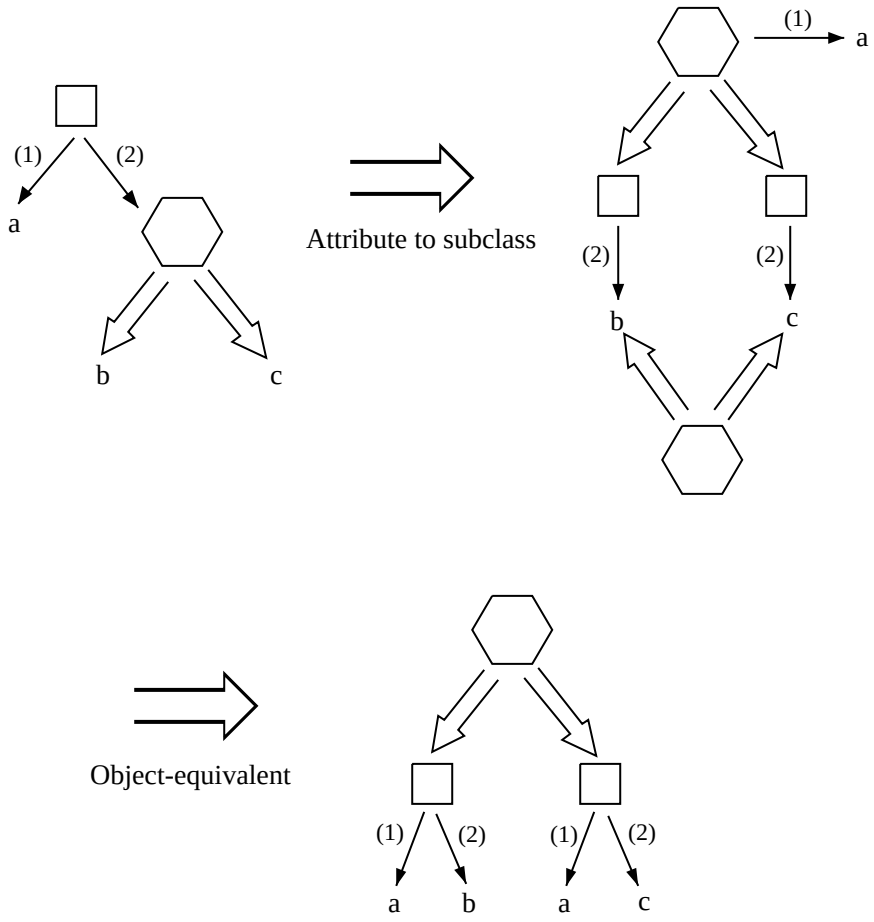


Figure 58: $(a \cdot (b + c)) = ((a \cdot b) + (a \cdot c))$

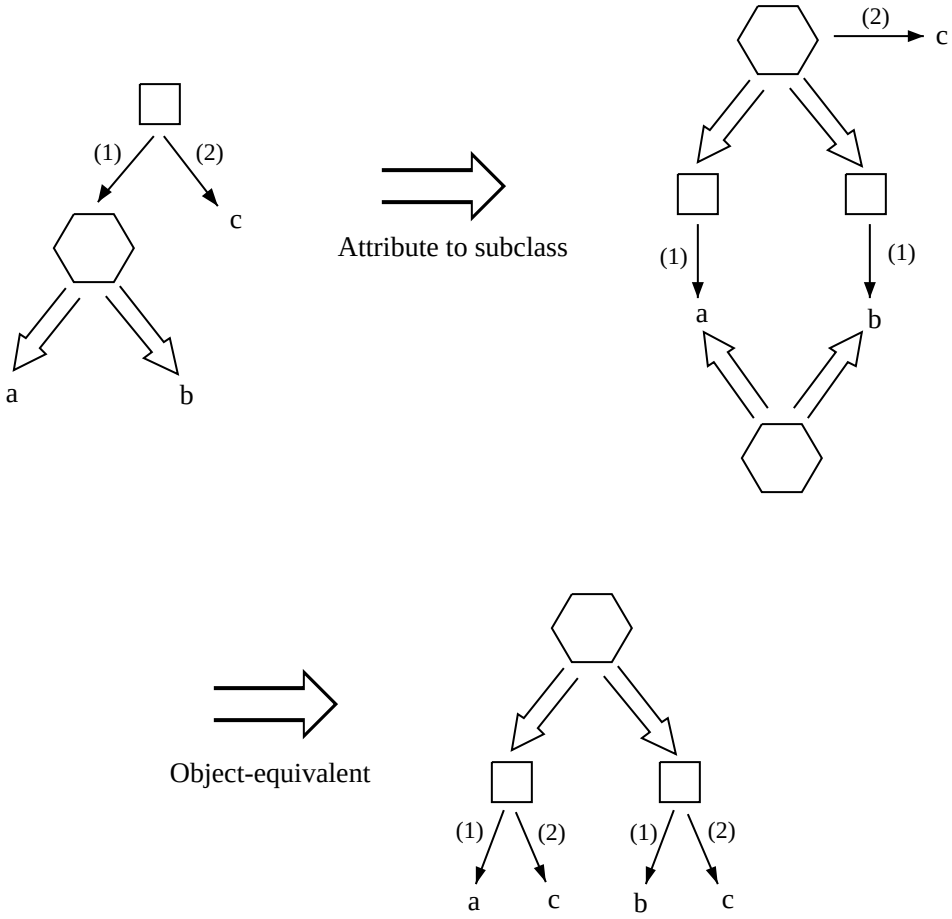


Figure 59: $((a + b) \cdot c) = ((a \cdot c) + (b \cdot c))$

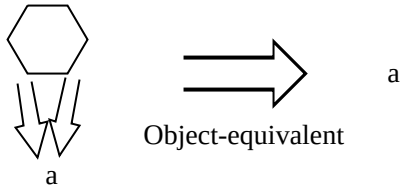


Figure 60: $(a + a) = a$

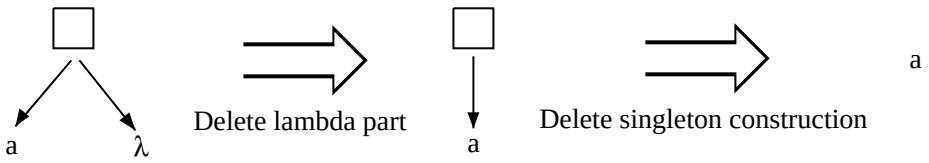


Figure 61: $(a \cdot \lambda) = a$

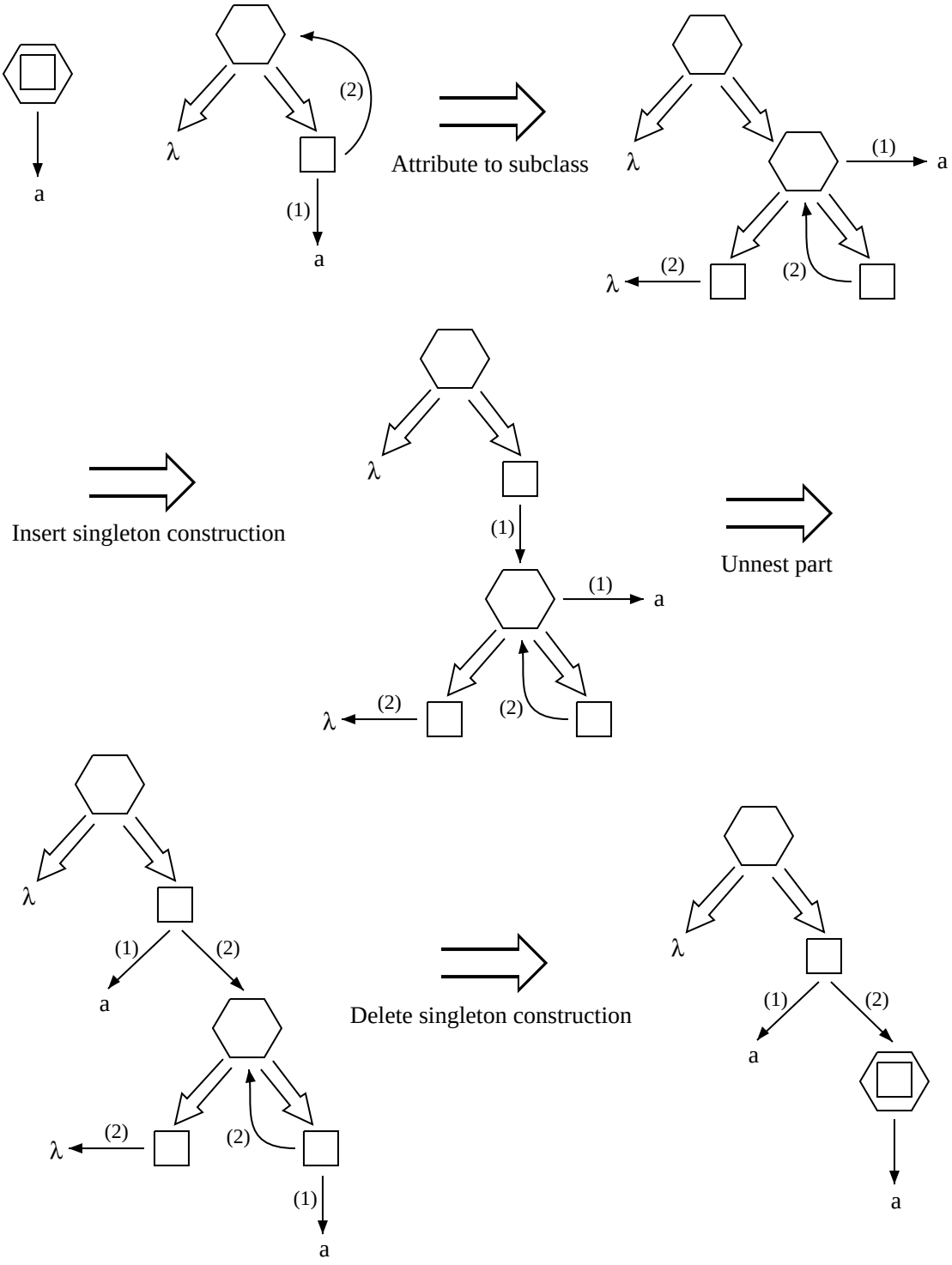


Figure 62: $(a*) = (\lambda + (a \cdot (a*)))$

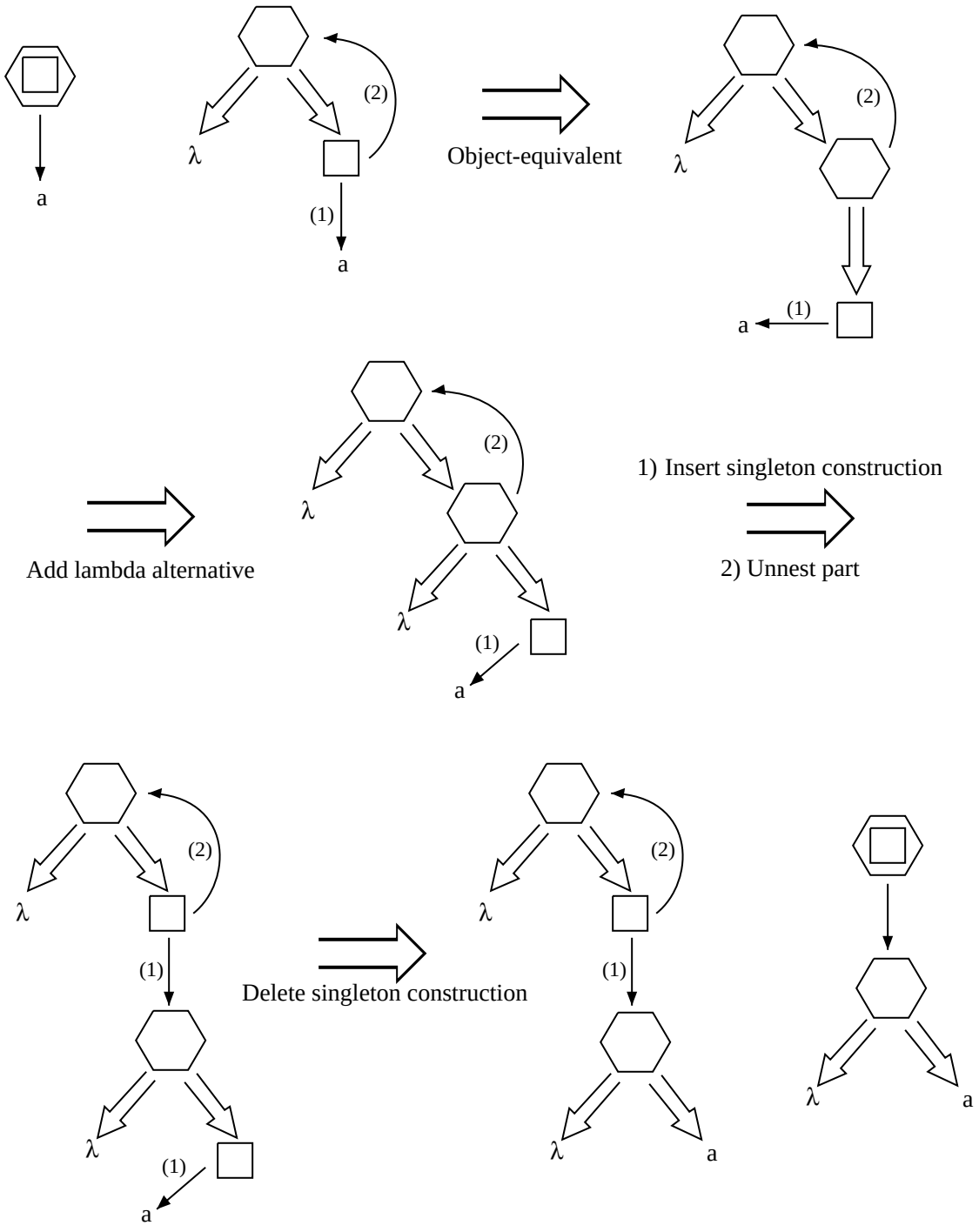


Figure 63: $(a*) = ((\lambda + a)*)$

its "container" (parent) object and may involve adding code to the container class. For example, when a part is unnested (moved up the part-of hierarchy) instances of the class which originally had the part must access it indirectly through their containers. In CDG there is a built-in method called `container` that provides the required access. In "real" languages, an instance variable may be added where necessary to provide a link to an object's container. Still, there are additional complications when the objects don't form a tree. When a class dictionary graph is used to define the class structure for a program written in a typical object-oriented language, a construction edge from some vertex, A, to another vertex, B, implies that every A object has a B object as a part, but not the converse; every B object is not necessarily a part of an A object. In general, there may be no suitable container class for a code transformation, and if there is a suitable class it cannot in general be identified without examining the existing code. The code transformations presented below, while correct for CDG programs, are intended to be used with human guidance in the general case. The container classes are specified manually, and code must be transformed manually if a container class is required by a code transformation rule and none is available. For strongly typed languages there are the complications discussed in Chapter 6.

8.5.1 The object-preserving transformations

The code updates required for the object-preserving transformations are defined in Chapter 6 (untyped language model).

8.5.2 Renaming of vertices and edges

No code updates are required when vertices are renamed. In the standard interpretation, renaming an edge corresponds to changing the identifier for an instance variable. When an edge, $(v \xrightarrow{l} w)$, is renamed to $(v \xrightarrow{l'} w)$, the identifier l' is substituted for the identifier, l, in the methods of class v. Since CDG provides strong data encapsulation, there is no need to make substitutions in methods of any other classes.

8.5.3 Nesting of parts

If a class, A, has outgoing construction edges, $(A \xrightarrow{b} B)$ and $(A \xrightarrow{c} C)$, and the edge $(A \xrightarrow{c} C)$ is replaced by an edge $(B \xrightarrow{c} C)$ (the c part of A is nested under its b part),

then in methods attached to class A, the c part must be accessed indirectly through its b part. An accessor method to return the c part is added to class B, and the identifier, `c`, is replaced by `b <- c()` in methods of class A. If methods of class C access A objects through the `container` operator, the access must now be indirect through the intermediate B object. We add an accessor method to class B for its container, and in methods of class C the expression `container` is replaced by the expression `container <- container()`.

8.5.4 Unnesting of parts

If a class, A, has an outgoing construction edge, $(A \xrightarrow{b} B)$ to a class B which has an outgoing construction edge, $(B \xrightarrow{c} C)$, to class C, and the edge $(B \xrightarrow{c} C)$ is replaced by the edge, $(A \xrightarrow{c} C)$, (the c part is unnested from under the b part of class A) then in methods attached to class B, the c part must be accessed indirectly through its parent A object. Similarly, in methods of class C that access B objects through the `container` operator, the access must now be indirect through its parent A object. Accessor methods are added to class A to return the b and c parts. In methods of class B, the expression `c` is replaced by the expression `container <- c()` and in methods of class C the expression `container` is replaced by `container <- b()`.

8.5.5 Addition of lambda parts

Adding a part does not require any change in the code.

8.5.6 Deletion of lambda part

If a class, A, has a construction edge, $(A \xrightarrow{b} B)$, to a "lambda" class B and the edge is deleted, class A must supply any functionality that was formerly delegated to class B. Note that since B is a lambda class, none of its methods have access to any objects other than `self` and `container`. Each method of class B is copied unchanged to class A except that the expression `container` is replaced with `self` in the copied methods. In class A's original methods, the expression `b` is replaced with the expression `self`.

8.5.7 Addition of lambda alternative

When an alternation edge, $(A \Longrightarrow B)$, is added from a class, A to a lambda class, B, by addition of lambda alternative, B objects may be interspersed in a list of A objects.

Any message received by such a B object should be passed to the A object that has been displaced. For each method attached to any subclass of A a corresponding method is generated for class B which simply delegates to the next object in the list. We also add to each of the original A classes a method called `this` which returns `self`. To class B we add a `this` method that returns `container <- this()`. In each of the original A methods the expression `container` is replaced by `container <- this()` so that these expressions will have the same objects as their values as if the B objects were not present in the list.

8.5.8 Deletion of lambda alternative

Deletion of a lambda alternative does not require any change in the code.

8.5.9 Insertion of singleton construction

When a new construction class, A, with a construction edge to a class, B, is added by insertion of singleton construction, a method which simply delegates to its B part is added to class A for each method in class B. An accessor is added to A for its container, and in the methods of class B the expression `container` is replaced by `container <- container()`.

8.5.10 Deletion of singleton construction

When a construction class, A, with a construction edge, $(A \xrightarrow{b} B)$, to class B is deleted by deletion of singleton construction, each method in class A is copied to class B with substitution of the expression `self` for `b`. In the original methods of class B the expression `container` is replaced by `self`.

8.5.11 Attribute to subclass

When a construction class, is transformed into an alternation class by attribute to subclass, its methods are simply copied to each of its new subclasses.

8.5.12 Subclass to attribute

When an alternation class, A, gains an attribute, l by subclass to attribute, each of its subclasses, B, must be a singleton construction whose only outgoing edge, $(B \xrightarrow{l} C)$, has label l. The elimination of the subclasses is handled as for deletion of singleton construction (above), except that an additional argument is added to each of the methods copied from

class B to C, and the copied methods are modified to access any parts inherited in B (from A) indirectly through the extra argument. Accessor methods are added to class A for each of its parts, and for each method copied from B to C, a corresponding method is added to class A which simply delegates to its l part, passing along whatever arguments it received plus `self` for the actual value of the extra argument.

8.6 Examples of CDG program transformations

Example 8.1. *Figures 64-66 show how yet another version of the prefix expression evaluator (Figure 64) is transformed when its class structure evolves first by transforming the* `op` *attribute of class* `Compound` *to subclasses followed by deletion of the useless alternation vertex* `Op` *(Figure 65) and then by deleting the singleton construction vertices* `MulExp` *and* `AddExp` *followed by renaming of the classes* `MulOp` *and* `AddOp` *to* `MulExp` *and* `AddExp`*, respectively (Figure 66).*

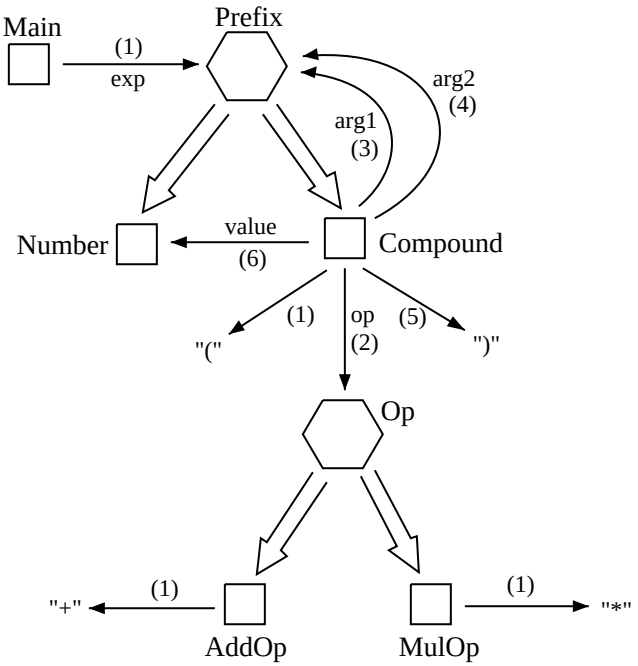
Example 8.2. *Figure 67 shows a CDG program to calculate the total weight of all the bricks in a pile. If the class dictionary graph evolves by addition of lambda alternative to allow balloons to be interspersed with the bricks the code to calculate the total weight of the bricks is updated as shown in Figure 68.*

8.7 Search algorithms

If the primitive language-preserving transformations are used to restructure the class organization of a CDG program, the code may be automatically updated following the rules defined in Section 8.5. More generally, given an arbitrary CDG program and a new language-equivalent class dictionary graph, we must be able to find a sequence of primitives that produces the given transformation in order to apply the code transformation rules.

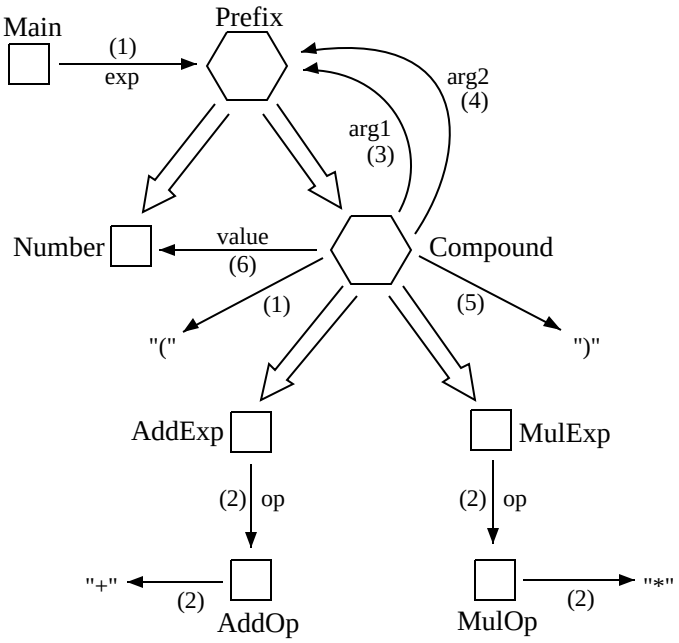
8.7.1 Regular languages

Since the primitive transformations are not complete for regular class dictionary graphs without the addition of a meta-transformation, it is not always possible to reduce an arbitrary language-preserving transformation over the regular class dictionary graphs to a sequence of primitives. However, there is an algorithm to perform the reduction to a sequence



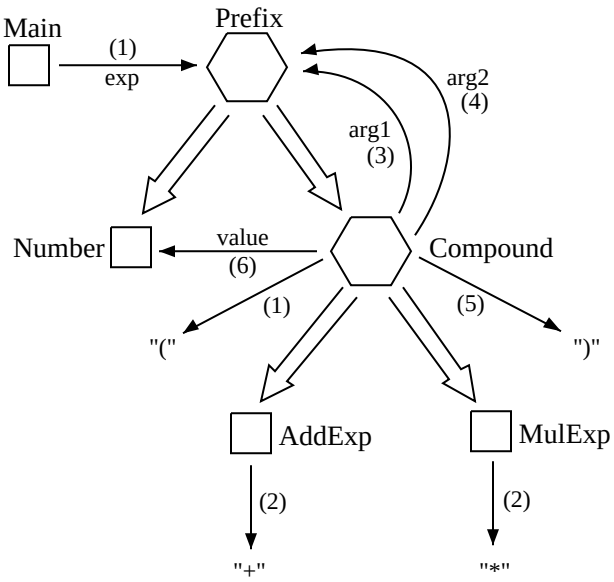

```
Main : main ()
{ exp <- eval() <- print() }

Number : eval ()
{ self }

Compound : eval ()
{ op <- apply(arg1, arg2, value) }

AddOp : apply (a1, a2, v)
{ v <- assign(a1 <- eval()) <- add(a2 <- eval()) }

MulOp : apply (a1, a2, v)
{ v <- assign(a1 <- eval()) <- mul(a2 <- eval()) }
```

Figure 64: Program C



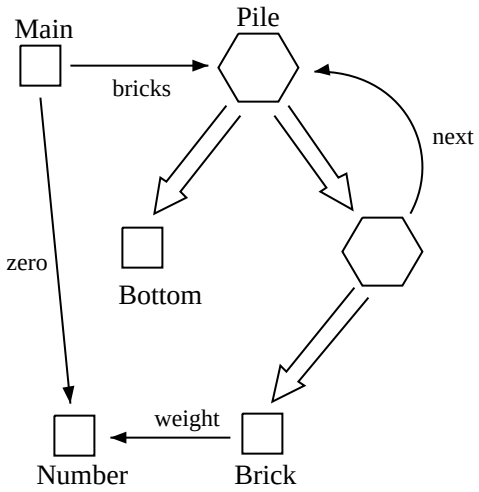

```
Main : main ()
{ exp <- eval() <- print() }

Number : eval ()
{ self }

MulExp : eval ()
{ op <- apply(arg1, arg2, value) }

AddExp : eval ()
{ op <- apply(arg1, arg2, value) }

AddOp : apply (a1, a2, v)
{ v <- assign(a1 <- eval()) <- add(a2 <- eval()) }

MulOp : apply (a1, a2, v)
{ v <- assign(a1 <- eval()) <- mul(a2 <- eval()) }
```

Figure 65: Program D



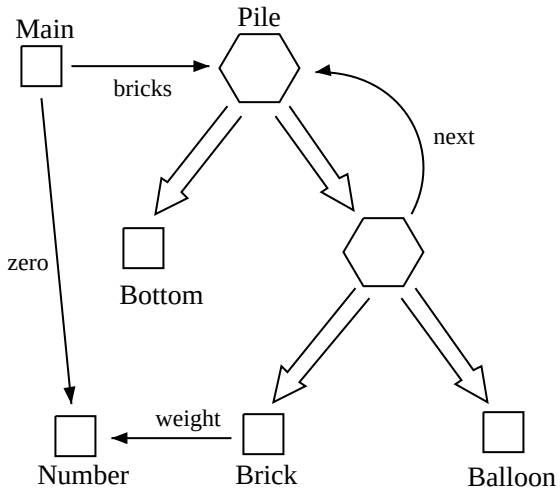

```
Main : main ()
{ exp <- eval() <- print() }

Number : eval ()
{ self }

MulExp : eval ()
{ self <- apply(arg1, arg2, value) }

AddExp : eval ()
{ self <- apply(arg1, arg2, value) }

AddExp : apply (a1, a2, v)
{ v <- assign(a1 <- eval()) <- add(a2 <- eval()) }

MulExp : apply (a1, a2, v)
{ v <- assign(a1 <- eval()) <- mul(a2 <- eval()) }
```

Figure 66: Program E




```
Main : main ()
{ bricks <- last() <- weight() <- print() }

Brick : last ()
{ next <- last() }

Bottom : last ()
{ container }

Brick : weight ()
{ container <- weight() <- add(weight) }

Main : weight ()
{ zero }
```

Figure 67: Program to calculate weight of brick pile




```
Main : main ()
{ bricks <- last() <- weight() <- print() }

Main : weight ()
{ zero }

Brick : last ()
{ next <- last() }

Brick : weight ()
{ container <- this() <- weight() <- add(weight) }

Brick : this ()
{ self }

Bottom : last ()
{ container <- this() }

Balloon : this ()
{ container <- this() }

Balloon : last ()
{ next <- last() }

Balloon : weight ()
{ next <- weight() }
```

Figure 68: After adding balloons to the pile

of primitives and meta-transformations. Manual code updates must then be performed only for the meta-transformations.

The proof that Salomaa's axiom system for the regular expressions is complete [Sal69] is constructive in the sense that for any valid equation $X = Y$, over the regular expressions it gives a method to constuct its proof. To reduce an arbitrary language-preserving transformation over the regular class dictionary graphs to a sequence of primitives we first construct the proof that the corresponding regular expressions are equivalent. Each substitution in the proof is mapped to a sequence of primitives as defined in Section 8.4.3. Each solution of equations is mapped to the meta-transformation defined in Section 8.4.3.

8.7.2 Context free languages

There can be no algorithm guaranteed to reduce an arbitrary language-preserving transformation over the extended class dictionary graphs to a sequence of primitives. Even if the set of primitive transformations were complete, such an algorithm would be impossible since equivalence of context-free languages is undecidable, and the extended class dictionary graphs define context-free languages[2]. Nevertheless, it is reasonable to expect that searches for sequences of primitives will often terminate successfully since the primitives are designed to represent the kinds of transformations that are likely to arise in practice.

The search problem may be viewed in terms of the classic state-space search paradigm as defined in the literature of artificial intelligence. Given an initial state, S, a set of operators on states, O, and a set of goal states, G, the state-space is defined as a directed graph where each node represents a state and each arc represents an operation. The problem is to find a path from the initial state to a goal state. Normally, the graph is not made explicit except for the solution path.

In our case, the initial state is a class dictionary graph, the operators are the primitive language-preserving transformations, and the only goal state is a language-equivalent class dictionary graph. Alternatively, we may consider the set of goal states to be the set of all class dictionary graphs which are object-equivalent to a given language-equivalent class dictionary graph since we already have efficient algorithms for checking object-equivalence and reducing an object-preserving transformation to a sequence of primitives.

State-space search has been heavily investigated in AI, and sophisticated systems have

[2]See Table 3. Also, consider extending the textual notation for class dictionary graphs presented in Chapter 2 with concrete syntax or terminals.

have been developed for evaluating states and choosing the next operation to apply in various domains. A detailed algorithm of this sort is beyond the scope of the current work and is left for future research. However, a simple search strategy might proceed as follows: The state space is searched in depth-first order (with backtracking) and operators are applied to vertices in the class dictionary graph in breadth-first order starting with the `Main` class. The `Main` vertex of the initial class dictionary graph is brought into congruence with the `Main` vertex of the goal state by applying operators (primitive transformations) until the vertices have the same types and numbers of outgoing edges, outgoing construction edges have the same labels, and outgoing syntax edges have the same targets. Next, the target of each construction and alternation edge is brought into congruence with the corresponding class in the goal state in the same manner, and the process continues until the target is reached or the depth in the state space exceeds some specified value.

An important heuristic which can are used to improve the search performance is to use the names of classes and labels of edges to guide the search. If, for example, a vertex must have an outgoing construction edge with label l to a vertex labeled V, we first check to see if there is already an outgoing edge with label l. If two analogous classes have parts with the same names, we guess that the parts are also analogous. Otherwise, we check if some other vertex has an outgoing edge with label l and target V that can be brought into the proper position by nesting and unnesting of parts. If neither condition is met we look for a vertex labeled V and finally for an edge with label l.

This strategy is useful if the class dictionary graph changes gradually during the evolutionary process, since most classes and parts will retain their original names. It is also useful if the designers use names consistently when reorganizing the class structure. Finally, if a class dictionary graph has changed dramatically it may be easy for a human designer familiar with the application domain to supply a mapping between classes with analogous roles by manually renaming parts and classes before starting the search. For the human designer, giving a partial analogy by renaming the parts and classes is the "easy" part – elaborating the analogy by finding the primitive transformations and then updating all of the code is the "hard" part. For the machine, the reverse is true; thus, the machine compliments the abilities of the human designer when this strategy is employed.

The concrete syntax may be used as a further guide of the search or to prune nodes in the state space if we note that it is not possible to find a solution by bringing two vertices into congruence that have different sets of reachable syntax vertices.

8.8 Related Work

8.8.1 Structure Mapping Theory

In structure mapping theory [Gen83, GT86] knowledge is represented as propositional networks comprising object nodes and predicates (attributes and relations). An analogy maps object nodes from the base domain to object nodes in the target domain. Generally, there is a 1-1 mapping between nodes in the base and target domains. Each pair of corresponding object nodes in the mapping is part of the analogy: "the target is like the base". The analogy is applied by using mapping rules, based on the principle of *systematicity*, to determine which predicates should be brought from the base domain to the target domain. The selected predicates are carried over using the node substitutions indicated in the object mapping.

A sequence of primitive transformations where each primitive only renames a vertex in a class dictionary graph graph would be equivalent to an analogy as defined by structure mapping theory. However, the primitive transformations can be more expressive since they may include changes in certain *structural relations* (e.g. part-of, kind-of) as part of the analogy. For example, in the base domain a class `Human` might have an attribute (part) called `Gender`, with possible values (kinds) `Male` or `Female`. In the target domain an analogous structure might have a class `Person` with subclasses (kinds) `Man` and `Woman`. The simple mapping ($Human \rightarrow Person, Male \rightarrow Man, Female \rightarrow Woman$) does not properly express the analogy. Instead, the relationship between `Person` and `Human` must be qualified, as in: "a `Person` is like a `Human` where the attribute `Gender` is expressed by subclassing". Gentner's structure mapping theory is not powerful enough to express such a qualified structural analogy, but this can be expressed by a primitive transformation, say "attribute-to-subclass".

In analogical program synthesis, application of the analogy involves bringing relations, in the form of program code, from the base domain over to the target domain. As in structure mapping theory, the rules depend only on syntactic properties and not on an understanding of the contents of the domains. Therefore, the code is brought over with little modification.

Structure mapping theory says nothing about how an analogy, "the *Target* is like the *Base*", is broken down into a mapping of nodes in the base to nodes in the target. In analogical program synthesis, a search is performed to find a sequence of primitive transformations that would convert the base structure to the target structure.

8.8.2 Graph Transformations in Analogical Reasoning

Pötschke [Pöt87] has used analogical reasoning to construct programs to control robots in assembly processes, and for programming chemical synthesis reaction sequences. In each case, scene descriptions are given in the form of digraphs where vertices are interpreted as objects and edges represent relations between the objects. A program consists of a sequence of instructions for transforming the graph from a representation of the initial scene to a representation of the final scene.

When a new program is required, it may be constructed automatically if the overall transformation desired is analogous to the transformation accomplished by the old program. The generated program must meet certain criteria. For example, in robotic assembly the solution must be collision-free. Sequences of substitutions are sought to convert the original initial scene to the new initial scene, and to convert the original final scene to the new new final scene. These substitutions are applied to the original program to produce a new program that will transform the new initial scene to the desired final scene.

An important similarity in Pötschke's work is that he generates sequences of graph transformations from initial state to goal state as an important step in the process of analogical program synthesis. He says nothing, however, about how the transformation sequences are generated and apparently does not work from a set of primitives. Another difference is that the programs themselves are always in the domain of graph transformation so that the generated sequences can be used to transform the original program in a trivial manner.

8.8.3 Analogical Program Synthesis Guided by Correctness Proofs

Ulrich and Moll [UM77] have used correctness proofs to guide the formation of analogies and the construction of analogous programs. Each line in the proof of a program written for the base domain is mapped into a statement in the target domain. Terms and relationships in the target domain are substituted for terms and relationships in the original proof. As the process is carried out, the original program is modified by the same substitutions. This process produces a new program and its correctness proof at the same time.

Dershowitz and Manna [DM77] used a similar approach to automatically modify programs. They formulate an analogy as a set of substitutions that yield a specification of the

desired program when applied to the specification of an analogous program. The specifications, including input specifications, are given for both programs in a high-level assertion language. In our case, the known CDG program contains its own input specification in the form of an extended class dictionary graph.

An important aspect of a program specification is the inclusion of *invariant assertions* which are utilized in correctness proofs. Transformations are applied to all assertions as well as to program code. The transformed assertions can be used to obtain verification conditions for the new program. In our case, it is the language defined by the extended class dictionary graphs that remains invariant. Correctness is guaranteed by the correctness of the primitive transformations.

Bibliography

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

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

[AKN86] H. Ait-Kaci and R. Nasr. Login: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[AS83] Dana Angluin and Carl Smith. Inductive inference: Theory. *ACM Computing Surveys*, 15(3):237–269, September 1983.

[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.

[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.

[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.

[Ber93] Paul L. Bergstein. Object-preserving class transformations: Applications to software design and maintenance. In Preparation, 1993.

[BH93] Paul L. Bergstein and Walter L. Hürsch. Maintaining behavioral consistency during schema evolution. In S. Nishio and A. Yonezawa, editors, *International Symposium on Object Technologies for Advanced Software*, pages 176–193, Kanazawa, Japan, November 1993. JSSST, Springer Verlag, Lecture Notes in Computer Science. Also available as Northeastern University, College of Computer Science technical report number NU-CCS-93-04.

[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.

[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.

[Boo91] Grady Booch. *Object-Oriented Design With Applications.* Benjamin/Cummings Publishing Company, Inc., 1991.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In Gilles Kahn, David MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types*, pages 51–67. Springer Verlag, 1984.

[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.

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

[CF82] Paul R. Cohen and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence*, volume 3. William Kaufmann, Inc., 1982.

[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.

[DJ88] V. Dhar and M. Jarke. Dependency directed reasoning and learning in systems maintenance support. *IEEE Transactions on Software Engineering*, 14(2):211–227, February 1988.

[DM77] Nachum Dershowitz and Zohar Manna. The evolution of programs: Automatic program modification. *IEEE Transactions on Software Engineering*, SE-3(6):377–385, November 1977.

[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.

[Gen83] Dedre Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7:155–170, 1983.

[GPG90] Marc Gyssens, Jan Paradaens, and Dirk Van Gucht. A graph-oriented object model for database end-user interfaces. In Hector Garcia-Molina and H.V. Jagadish, editors, *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 24–33, Atlantic City, 1990. ACM Press.

[GT86] Dedre Gentner and Cecile Toupin. Systematicity and surface similarity in the development of analogy. *Cognitive Science*, 10:277–300, 1986.

[Hal89] Rogers P. Hall. Computational approaches to analogical reasoning: A comparative analysis. *Artificial Intelligence*, 39:39–120, 1989.

[HY84] R.B. Hull and C.K. Yap. The format model: A theory of data organization. *Journal of the Association for Computing Machinery*, 31(3):518–537, July 1984.

[IUT87] Takeshi Imanaka, Kuniaki Uehara, and Junichi Toyoda. Analogical program synthesis from program components. In *Logic Programming 1987. Proceedings of the Sixth Conference*, pages 69–79, Tokyo, Japan, June 22-24 1987. Springer-Verlag 1988.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[KK88] J. Karimi and B.R. Konsynski. An automated software design assistant. *IEEE Transactions on Software Engineering*, 14(2):194–210, Feb. 1988.

[Kli71] Robert E. Kling. A paradigm for reasoning by analogy. *Artificial Intelligence*, 2:147–178, 1971.

[KV84] G.M. Kuper and M.Y. Vardi. The logical data model. In *Principles of Database Systems*, pages 86–96. ACM, 1984.

[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.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Electrical Engineering and Computer Science Series. MIT Press, McGraw-Hill Book Company, 1986.

[LH89] Karl J. Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[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.

[LHR88] Karl J. Lieberherr, Ian Holland, and Arthur J. Riel. Object-oriented programming: An objective sense of style. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, number 11, pages 323–334, San Diego, CA., September 1988. A short version of this paper appears in IEEE Computer, June 88, Open Channel section, pages 78-79.

[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.

[LHX94] Karl J. Lieberherr, Walter L. Hürsch, and Cun Xiao. Object-extending class transformations. *Formal Aspects of Computing, the International Journal of Formal Methods*, 6(4), 1994.

[Lie85] Y.E. Lien. Relational database design. In S. Bing Yao, editor, *Principles of Database Design*, pages 211–254. Prentice Hall, 1985.

[Lie88] Karl J. Lieberherr. Object-oriented programming with class dictionaries. *Journal on Lisp and Symbolic Computation*, 1(2):185–212, 1988.

[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.

[LR88] Karl J. Lieberherr and Arthur J. Riel. Demeter: A CASE study of software growth through parameterized classes. *Journal of Object-Oriented Programming*, 1(3):8–22, August, September 1988. A shorter version of this paper was presented at the *10th International Conference on Software Engineering, Singapore, April 1988, IEEE Press*, pages 254-264.

[LRV90] Christophe Lecluse, Philipe Richard, and Fernando Velez. O2, an object-oriented data model. In Zdonik and Maier, editors, *Readings in Object-Oriented Database Systems*, pages 227–236. Morgan Kaufmann Publishers, 1990.

[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.

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

[MMP88] Ole Lehrmann Madsen and Birger Møller-Pedersen. What object-oriented programming may be - and what it does not have to be. In S.Gjessing and K. Nygaard, editors, *European Conference on Object-Oriented Programming*, pages 1–20, Oslo, Norway, 1988. Springer Verlag.

[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.

[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.

[Pöt87] Dieter Pötschke. Synthesis of programs for intelligent robots by analogy algorithms. In I. Plander, editor, *Artificial Intelligence and Information-Control Systems of Robots*, pages 411–415. Elsevier Science Publishers, North-Holland, 1987.

[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.

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

[Sal69] Arto Salomaa. *Theory of Automata*. International series of monographs in pure and applied mathematics, v. 100. Pergamon Press, 1969.

[SM86] R.E. Stepp and R.S. Michalski. Conceptual clustering: Inventing goal-oriented classification of structured objects. In R.S. Michalski et al., editor, *Machine Learning: An Artificial Intelligence Approach, Vol. II*, pages 471–498. Morgan-Kaufman Publishers, 1986.

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

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

[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.

[TL82] Dennis Tsichritzis and Frederick Lochovsky. *Data Models*. Software Series. Prentice-Hall, 1982.

[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.

[UM77] John Wade Ulrich and Robert Moll. Program synthesis by analogy. *SIGPLAN Notices*, (64):22–28, August 1977.

[Weg90] Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, Aug. 1990.

[Win70] P.H. Winston. Learning structural descriptions from examples. Technical Report 76, MIT, 1970. Project MAC.