Incremental Class Dictionary Learning and Optimization

Paul L. Bergstein and Karl J. Lieberherr
Northeastern University, College of Computer Science
Cullinane Hall, 360 Huntington Ave., Boston MA 02115
(pberg or lieber)@corwin.CCS.northeastern.EDU

Abstract

We have previously shown how the discovery of classes from objects can be automated, and how the resulting class organization can be efficiently optimized in the case where the optimum is a single inheritance class hierarchy. This paper extends our previous work by showing how an optimal class dictionary can be learned incrementally. The ability to expand a class organization incrementally as new object examples are presented is an important consideration in software engineering.

Keywords: Object-oriented programming and design, reverse engineering, class library organization, class abstraction algorithms.

1 Introduction

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.

We have previously shown ([LBSL90], [LBSL91]) how the discovery of classes from objects can be automated, and how the resulting class organization can be efficiently optimized in the case where the optimum is a single inheritance class hierarchy. This paper extends our previous work in an important way: We show how an optimal class organization can be learned **incrementally**.

The algorithms discussed in this paper are a part of our research results in reverse engineering of programs from examples. In one line of research, we start with object examples and apply an abstraction algorithm described in this paper to get a set of class definitions. Then we apply a legalization algorithm to the class definitions to ensure that each recursive class definition is well behaved. Next, an optimization algorithm summarized in this paper makes the class definitions as small as possible while preserving the same set of objects. Then we apply an LL(1)-correction algorithm which adds some concrete syntax to the class definitions to make the object description language LL(1) for easy readability and learnability. The object description language allows very succinct object descriptions and the LL(1)-property guarantees that there is a one-to-one correspondence between sentences and objects. Finally we apply a C++ code generation algorithm to the class definitions which produces a tailored class library for manipulating the application objects (e.g., reading, printing, traversing, comparing, copying etc.).

This sequence of algorithms allows us to produce a tailored C++ library just from object examples. After the specific object implementations are injected into this library, we have the complete application code. The creative steps in this method of software development are 1) to find the right objects, 2) to find good replacements for the names which are generated by the abstraction programs, 3) to fine tune the object syntax and 4) to write the specific object implementations. However, it is much easier to start with a custom generated C++ class library than to proceed manually from the object examples. For further information on our research program in object-oriented software engineering, we refer the reader to the survey in [WBJ90].

In section 2 the basic learning algorithm is formally presented. An informal presentation has been given in [LBSL90]. 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 given in section 3. The ability to expand a class dictionary incrementally as new object examples are presented is an important consideration in software engineering. In section 4 the algorithm is extended to incrementally learn an optimal class dictionary graph when the optimum is a single inheritance class dictionary.

Our algorithms are programming language independent and are therefore useful to programmers who use object-oriented languages such as C++ [Str86], Smalltalk [GR83], CLOS [BDG+88] or Eiffel [Mey88]. We have implemented the abstraction algorithms as part of our C++ CASE tool, called the C++ Demeter System^{TM} [Lie88], [LR88]. The input to the abstraction algorithms is a list of object examples, and the output is a programming language independent set of class definitions. They can be improved by the user and then translated into C++ by the CASE tool.

We first describe our class definition and object example notations (the key concepts behind the algorithms we present in this paper), since they are not common in the object-oriented literature.

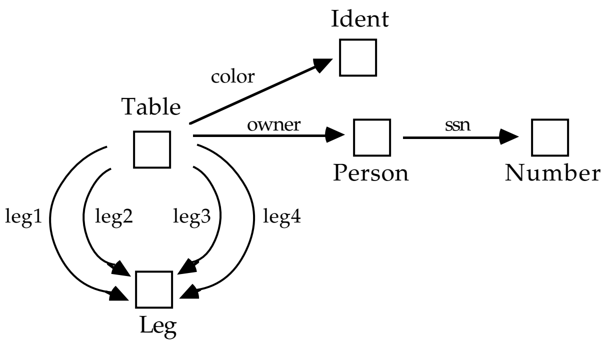


Figure 1: Construction class

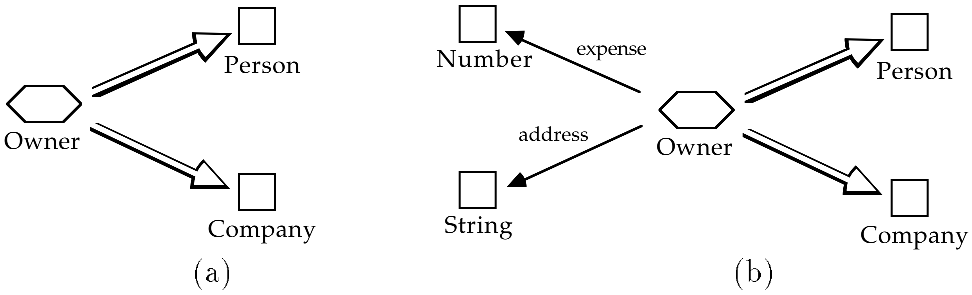


Figure 2: Alternation classes

1.1 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. Examples of construction classes are in Figure 1 for: Table, Leg, etc.

Each construction class 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 owner of a table can be either a person or a company. So the objects we want to store in the owner part of the table are either person or company objects. We use alternation classes to define such union classes. An example of an alternation class is in Fig. 2a.

`Person` and `Company` are called alternatives of the alternation class. Often the alternatives have some common parts. For example, each owner had an expense to acquire the object. We use the notation in Fig. 2b 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

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

to use than variant records.

Alternation classes which have common parts are implemented by inheritance. In Fig. 2b, `Person` and `Company` inherit from `Owner`. Class `Owner` has methods and/or instance variables to implement the parts `expense` and `address`.

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.

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

Next we partition the set of vertices into two subclasses, called the construction and alternation vertices.

Definition 2 *We define*

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

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

Definition 3 *Vertex $v_k \in V$ in a class dictionary graph, $\phi = (V, \Lambda;\ EC, EA)$, is said to be* **alternation reachable** *from vertex $v_0 \in V$ via a* **path of length** *$k \geq 1$, if there exist $k-1$ vertices $v_1, v_2, ..., v_{k-1}$ such that for all j, $0 \leq j < k$, $(v_j, v_{j+1}) \in EA$. The path consists of the sequence of alternation edges. We say that every vertex is alternation-reachable from itself.*

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

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

1. *Cycle-free alternation axiom:*

   There are no cyclic alternation paths, i.e., $\forall v \in VA$ there is no alternation path from v to v.

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

2. *Unique labels axiom:*

   $\forall w \in V$ there are no $p_1, p_2 \in V$ $s.t.$ $\exists x, y \in V, l \in \Lambda$ $s.t.$ $e_1 = (p_1, x, l) \in EC$ *and* $e_2 = (p_2, y, l) \in EC$, $e_1 \neq e_2$ and w is alternation reachable from p_1 and p_2.

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

In the rest of this paper, when we refer to a class dictionary graph we mean a legal class dictionary graph.

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 double arrows for alternation edges (see Figures 1 and 2).

1.2 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 use 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., legs of the table) or attributes or properties (e.g., owner or color). An object example is in Fig. 3 which defines a table object with 6 parts: 4 physical parts (legs) and two attributes: color and owner. The object example also indicates that the

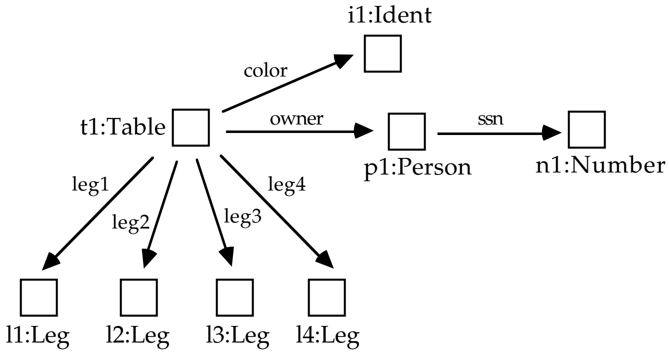


Figure 3: Table object

four legs have no parts and that the owner is a Person object with one part called `ssn` which is a Number.

Definition 5 *An* **object example graph with respect to a set of classes, S,** *is a graph $H = (W, S, \Lambda_H;\ E, \lambda)$ with vertex set W. Λ_H is a set of edge labels. E is a ternary relation on $W \times W \times \Lambda_H$. If $(v, w, l) \in E$, we call l the label of the labeled edge (v, w, l), from v to w. The function $\lambda : W \rightarrow S$ labels each vertex of H with an element of S. The following axioms must hold for H:*

(1) No vertex of H may have two outgoing edges with the same label. (2) All vertices which have the same element $s \in S$ as label (under λ) must have either outgoing edges with the same labels or no outgoing edges at all.

Definition 6 *An* **object graph with respect to a class dictionary graph,** ϕ *is an object example graph, $H = (W, S, \Lambda_H;\ E, \lambda)$ with respect to set S, where $S = VC_\phi$ and $\Lambda_H \subseteq \Lambda_\phi$.*

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. For a formal definition of legality see [LBSL91].

The set of all legal object graphs defined by a class dictionary graph ϕ is called $Objects(\phi)$.

When we optimize a class dictionary graph, we must insure that the optimized version defines the same set of objects. The following definition formalizes the concept that two sets of class definitions define the same set of objects.

Definition 7 *A class dictionary graph $G1$ is* **object-equivalent** *to a class dictionary graph $G2$ if $Objects(G1) = Objects(G2)$.*

We use a textual notation for describing object graphs using an adjacency representation which also shows the mapping of object graph vertices to class dictionary graph vertices. The example of Fig. 3 has the following textual representation:

```
t1:Table(
  <leg1> l1:Leg()
  <leg2> l2:Leg()
  <leg3> l3:Leg()
  <leg4> l4:Leg()
  <color> i1:Ident()
  <owner> p1:Person(
    <ssn> n1:Number()))
```

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.

1.3 A simple example of incremental class dictionary learning

Example 1 *Consider the two object graphs which represent a basket containing two apples and a basket with an orange:*

```
b1:Basket(
   <contents> o1:OneOrMore(
                    <one> a1:Apple( <weight> n1:Number())
                    <more> o2:OneOrMore(
                                 <one> a2:Apple( <weight> n2:Number())
                                 <more> no1:None())))

b1:Basket(
   <contents> o1:OneOrMore(
                    <one> or1:Orange( <weight> n1:Number())
                    <more> no1:None()))
```

After seeing the first object example graph, the learning algorithm generates the class dictionary graph in Fig. 4a. Now when the second object example is presented, the algorithm will learn the class dictionary graph in Fig. 4b.

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.

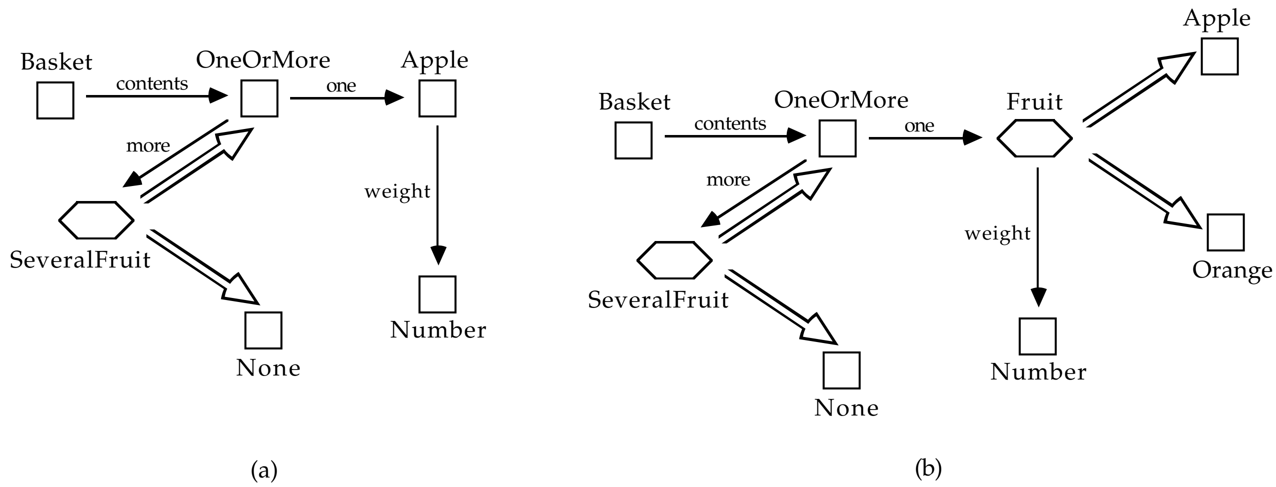


Figure 4: Fruit basket class dictionary graphs

```
// 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; }
```

2 Basic Learning

Given a list of object example graphs, the basic learning algorithm 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 list of object example graphs, $\Omega_1, \Omega_2, ..., \Omega_n$, we learn a legal class dictionary graph, ϕ, such that $Objects(\phi) \supseteq \{\Omega_1, \Omega_2, ..., \Omega_n\}$, and for all legal class dictionary graphs, ϕ' where $Objects(\phi') \supseteq \{\Omega_1, \Omega_2, ..., \Omega_n\} : Objects(\phi) \subseteq Objects(\phi')$.

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 8 *A list of object example graphs $\Omega_1, ..., \Omega_n$ is* **legal** *if all vertices which have the same element $s \in S$ as label (under λ_{Ω_i} for some $i, 1 \leq i \leq n$) have either outgoing edges with the same labels (under E for Ω_i) or no outgoing edges at all.*

A legal list of object example graphs $\Omega_1, ..., \Omega_n$ of the form $\Omega = (W_\Omega, S_\Omega, \Lambda_\Omega; E_\Omega, \lambda_\Omega)$ is translated into a class dictionary graph $\phi = (V, \Lambda;\ EC, EA)$ as follows:

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

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

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

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

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

   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.

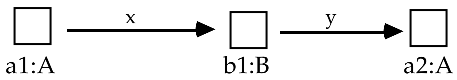
6. $EA = \{((r,l),s)|(r,l) \in VA, s \in V, \exists i, v, w : (v,w,l) \in E_{\Omega_i}, \lambda_{\Omega_i}(v) = r, \lambda_{\Omega_i}(w) = s\}$

   Finally, we create a 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:

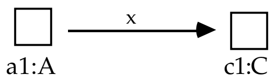
Example 2 .
Ω_1: *a1:A(⟨x⟩ b1:B(⟨y⟩ a2:A))*

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



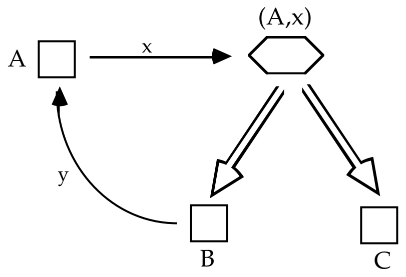
Ω_2: *a1:A(⟨x⟩ c1:C())*

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



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 Incremental Learning

Given a class dictionary graph, ϕ, and an object example graph, Ω, the incremental learning algorithm will learn a class dictionary graph, ϕ', such that the set of objects defined by ϕ' includes Ω 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 Ω 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 Ω.

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

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

Definition 9 *An object example graph Ω is* **incrementally legal** *with respect to a class dictionary graph ϕ if there exists a legal class dictionary ϕ' such that $Objects(\phi') \supseteq Objects(\phi) \cup \Omega$.*

If a list of object example graphs $\Omega_1, ..., \Omega_n$ is legal, then each Ω_i in the list must be incrementally legal with respect to the class dictionary graph learned from $\Omega_1, ..., \Omega_{i-1}$. Therefore a class dictionary graph can be learned incrementally from a legal list of object example graphs.

Denote the intermediate class dictionary learned from $\Omega_1, \Omega_2, ..., \Omega_m$ by ϕ_m, and let $\phi_0 = (\emptyset, \emptyset; \emptyset, \emptyset)$. Then ϕ_m is learned from ϕ_{m-1} and Ω_m, where $1 \leq m \leq n$, as follows:

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

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

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

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

   The first term represents the alternation vertices already learned in ϕ_{m-1}. The second term adds the alternations we learn from Ω_m alone (this is the same term as in the Basic Algorithm, where $\Omega_i = \Omega_j = \Omega_m$). The last term adds alternations that are learned in the Basic Algorithm when $\Omega_i \neq \Omega_j$. In the case of incremental learning we rely on the fact that the edges of Ω_1, ..., Ω_{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. $EC = (EC_{\phi_{m-1}} - \{(r,s,l) \mid (r,l) \in (VA - VA_{\phi_{m-1}})\})$
   $\cup \{(r,(r,l),l) \mid (r,l) \in (VA - VA_{\phi_{m-1}})\}$
   $\cup \{(r,s,l) \mid r, s \in V, \exists v, w \in W_{\Omega_m} : \lambda_W^{\Omega_m}(v) = r, \lambda_W^{\Omega_m}(w) = s, (v, w, l) \in E_{\Omega_m}, (r,l) \notin VA\}$

   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 Ω_m.

6. $EA = EA_{\phi_{m-1}}$
   $\cup \{((r,l),s)(r,l) \in VA, s \in V, \exists v, w \in W_{\Omega_m} : \lambda_W^{\Omega_m}(v) = r, \lambda_W^{\Omega_m}(w) = s, (v, w, l) \in E_{\Omega_m}\}$
   $\cup \{((r,l),s)(r,l) \in VA, s \in V, (r,l,s) \in EC_{\phi_{m-1}}\}$

   Here we start with the alternation edges from the previous class dictionary graph and add edges learned from Ω_m alone, and from Ω_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 1 *A class dictionary graph learned incrementally is identical to the class dictionary graph learned using the basic learning algorithm.*

4 Incremental Optimization

In this section, we develop an algorithm for incrementally learning minimum single-inheritance class dictionary graphs. We measure class dictionary graphs by counting the number of edges, except that we consider construction edges to be at least twice as expensive as alternation edges. Consideration of this 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 a vertex, v, in a class dictionary graph has two incoming construction edges with the same label, l, the part (l, v) is redundant.

We observe that we can always avoid redundant parts by introducing 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 always produces the smaller class dictionary, since construction edges are at least twice as expensive as alternation edges.

In [LBSL91] an efficient algorithm is presented for abstracting minimum 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 useless alternation vertices, and with a single-inheritance hierarchy is guaranteed minimal. An alternation vertex is "useless" if it does not have at least two outgoing alternation edges.

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

Instance:
A minimum single-inheritance class dictionary graph, ϕ, and an object example graph, Ω, where Ω is incrementally legal with respect to ϕ.

Problem:
Find a minimum single-inheritance class dictionary graph, ϕ', such that
$Objects(\phi') \supseteq Objects(\phi) \cup \Omega$.

In order to maintain the desired conditions in the intermediate class dictionary graphs, each new object example graph must meet two criteria:

1. If we learn from an object example graph, $H = (W, S, \Lambda_H;\ E, \lambda)$, that a class occurring in H (under λ) has a part in common with some other class, C, in the class dictionary it must have all the parts inherited by C.

2. If an object has a class with parts in common with two or more classes in the class dictionary, all of the classes with which it has parts in common must lie on a single alternation path.

It is easy to see how the incremental learning algorithm presented in section 3 can be extended to produce minimum single-inheritance class dictionaries.

5 Practical Relevance

In this paper we propose a metric (minimizing the number of edges) for measuring class hierarchies. We propose to minimize the number of construction and alternation edges of a class dictionary graph while keeping the set of objects invariant. Our technique is as good as the input which it gets: 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 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.

Our current metric is quite rough: we just minimize the number of edges. We also minimize the amount of multiple inheritance (since this is consistent with minimizing edge size), but ignore other criteria such as the amount of repeated inheritance. This is left for future research.

We motivate now why our metric produces class hierarchies which are good from a software engineering point of view.

5.1 Minimizing the number of construction edges: CNF

We minimize the number of construction edges by eliminating redundant parts. We say a class dictionary with no redundant parts is in class dictionary common normal form (CNF).

Even simple functions cannot be implemented properly if a class dictionary is not in CNF. By properly we mean with resilience to change. Consider the class dictionary in Figure 5 which is not in CNF. Suppose we implement a print function for Coin and Brick. Now assume that several hundred years have passed and 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 in Figure 6. Now we implement the print function for Coin:

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

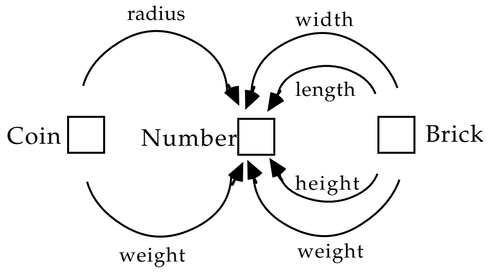


Figure 5: A class dictionary not in CNF

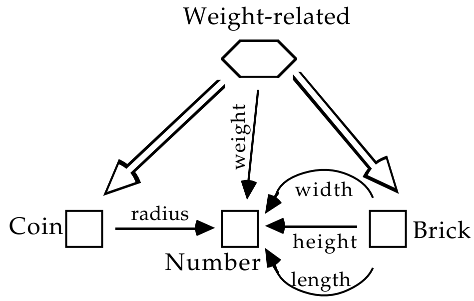


Figure 6: After transforming to CNF

After the change of the weight composition, we get the class dictionary in Figure 7. 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 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 which are instantiated in f.

Transformation to CNF can be more complicated, but not less beneficial, than the above example suggests.

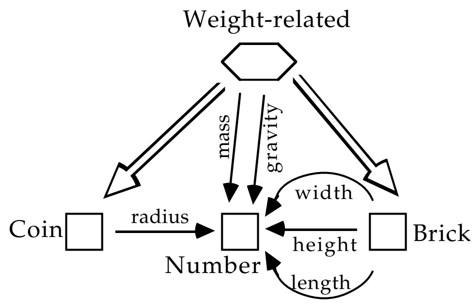


Figure 7: After change of weight composition

5.2 Minimizing the number of alternation edges

Consider the non-minimal class dictionary in Figure 8. By changing the class definitions for Occupation and Univ-employee we get the class dictionary in Figure 9. We have now reduced the number of alternation edges by 5 and have also reduced the amount of multiple inheritance, which we propose as another metric to produce "good" schemas from the software engineering point of view.

Another indication that our class dictionary optimization algorithm is useful is that it succeeds in finding single-inheritance solutions. We can prove the following statement: If we give a class dictionary which is object-equivalent to a single-inheritance class dictionary to the optimization algorithm, it will return such a single-inheritance class dictionary. 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.

6 Related work

Our work is a continuation of earlier work on inductive inference [CF82, Chapter XIV: Learning and inductive inference], [AS83]. Our contribution is an efficient algorithm for inductive inference of high-level class descriptions from examples. Related work has been done in the area of learning context-free grammars from examples and syntax trees [AS83]. The key difference to our work is that our approach learns grammars with a richer structure, namely the class dictionary graphs we learn, define both classes and languages.

In [Cas89, Cas90] and in his upcoming dissertation, Eduardo Casais introduces 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 two axioms needed for data modeling. For example, we distinguish between abstract and concrete classes.
- A step in Casais' incremental algorithm consists of adding a new subclass with potentially rejected attributes. In our work, an incremental step is adding a new object to a class hierarchy and to restructure the hierarchy so that it is optimal and at the same time describes the newly added object.
- The goal of Casais' algorithms is to restructure class hierarchies to avoid explicit rejection of inherited properties. In our work we avoid rejected properties.
- Casais' algorithms deal with operation signatures. In our work we have not added operation signatures yet. However, an operation of a class can be easily represented as a part by encoding the signature into the part's class name.

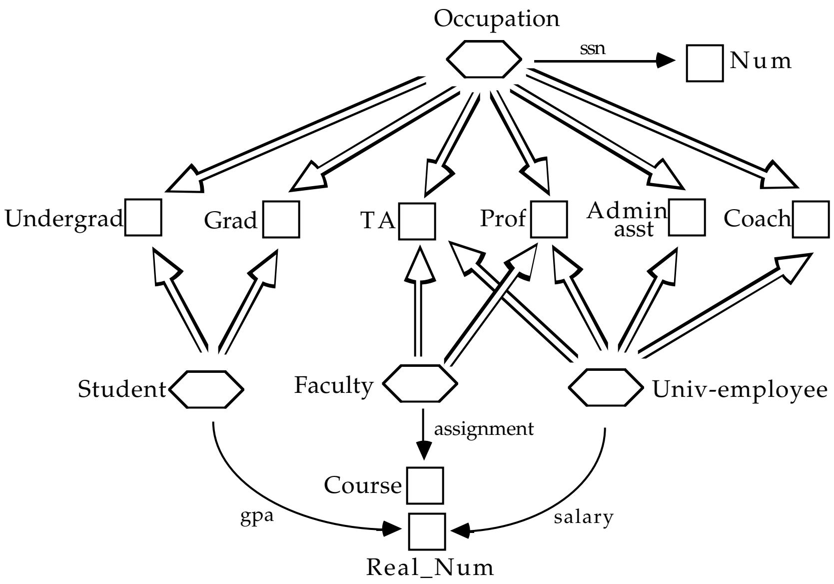


Figure 8: Before minimizing alternation edges

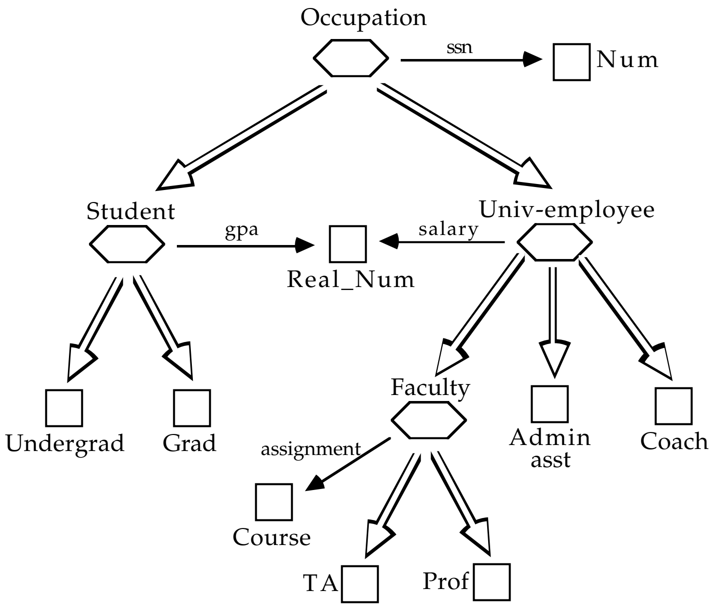


Figure 9: After alternation edge minimization

In [LM91] several ways in which conceptual database evolution can occur through learning are discussed. One of these, the generalization of types to form supertypes, is a special case of our abstraction of common parts, where there are only two objects from which the common parts are abstracted. Another, 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.

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.

7 Conclusion

We have presented novel algorithms for incremental learning and optimization of class dictionaries. Earlier we have studied global class reorganization algorithms [LBSL90], [LBSL91]. Incremental algorithms are useful for at least two reasons:

First, incremental algorithms are much more efficient than global reorganization algorithms. If we have a class library with several hundred classes, we don't want to globally restructure all those classes if there is a small evolution of some class definition.

Second, incremental algorithms give further insight into the design process. They serve as a tool to understand change propagation when there is a change in the class structure.

Our algorithms are a useful ingredient to a tool suite for object-oriented design and programming and we have implemented them in the Demeter System.

Acknowledgments: We would like to thank Ignacio Silva-Lepe for the university example given in section 6.2.

References

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

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

[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.

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

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

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

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

[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.

[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.

[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.

[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.

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

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

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

[WBJ90] Rebecca J. Wirfs-Brock and Ralph E. Johnson. A survey of current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, September 1990. The description of the Demeter project starts on page 120.

Contents