From objects to classes: Algorithms for optimal object-oriented design

Karl J. Lieberherr,
Paul Bergstein,
Ignacio Silva-Lepe
Northeastern University, College of Computer Science
Cullinane Hall, 360 Huntington Ave., Boston MA 02115
lieber@corwin.CCS.northeastern.EDU
phone: (617) 437 20 77

July 27, 1993

Texed at 21:10 July 27, 1993

Accepted for publication in "Journal of Software Engineering", a joint publication of IEE and the British Computer Society.

Abstract

The contributions of this paper are two-fold: First we introduce a novel, axiomatically defined, object-oriented data model, called the Demeter kernel model, and second we present abstraction and optimization algorithms and their relationships for designing classes from objects in the kernel model. We analyze several computational problems underlying the class design process which is divided into two phases: a learning phase and an optimization phase. This study focuses on approximation algorithms for the optimization phase and leads to a better understanding and a partial automation of the object-oriented design process. The algorithms and the theory presented in this paper have been implemented in the C++ Demeter System^{TM}, a CASE tool for object-oriented design and programming.

Keywords: Object-oriented programming and design, reverse engineering, formal software engineering techniques, class hierarchy optimization.

Acknowledgements: Preliminary versions of this paper were published in the proceedings of ER '90 [LBSL90b] and SOOLS '90 [LBSL90a].

1 Introduction

In their paper [JF88], Johnson and Foote state that "useful abstractions are usually designed from the bottom up, i.e., they are discovered, not invented." In [Ste89], Steier presents a theory of human algorithm design. He compares this theory with published results from human design activities in other areas (Lisp programming, Fortran programming, software design, mechanical design and architectural design). Steier's theory of algorithm design as well as the other studies about human design, support Steier's following claim:

> Design is driven by evaluation in the context of examples: Designers run their solutions to evaluate them in the context of some input to the program.

This statement gives strong support for object-oriented design to incorporate objects and not only classes[1]. In this paper we propose algorithms which automate the discovery of classes from object examples. We also study the complexity of the underlying computational problems.

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 writing the class definitions by hand. We present algorithms for abstracting class definitions (instance variable types, abstract classes, inheritance) from a representative sample of object examples.

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

1.1 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

[1] Readers not familiar with object-oriented analysis, design and programming are referred to books such as [Cox86, Mey88, CY90, Boo91].

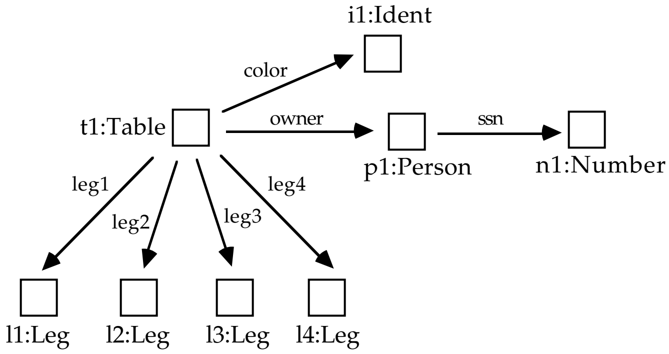


Figure 1: Table object

by parts and a sequence of actions. It is the modelling 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., legs of the table) or attributes or properties (e.g. owner or color). An object example is in Fig. 1 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 four legs have no parts and that the owner is a person object with one part called `ssn`.

1.2 Motivation for class notation

We use a class notation which uses two kinds of classes: construction and alternation classes.[2] 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 Fig. 1 is in Fig. 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

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

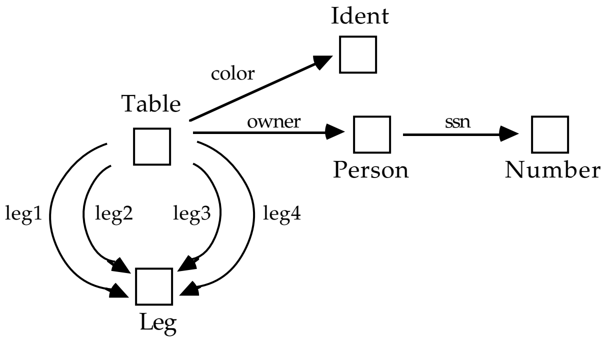


Figure 2: Construction class

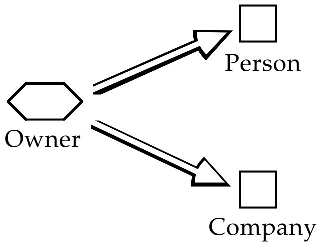


Figure 3: Alternation class

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. 3. 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. 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.

Alternation classes which have common parts are implemented by inheritance. In Fig. 4,

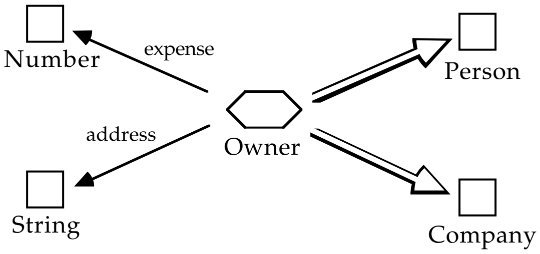


Figure 4: Common parts

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.

1.3 Informal description of the algorithm

We informally describe the abstraction algorithm MCDL-P (for Minimum Class Dictionary Learning) in two steps: A basic learning step and a minimization step.

In the first step, we abstract a potentially non-optimal class dictionary graph from the object examples. The abstracted class dictionary graph defines objects which are "similar" to the given object examples. The object examples are described by a graph which shows the immediate subparts of a given object. From the five object examples in Fig. 5a-e we abstract the non-optimal class dictionary graph in Fig. 5f. Two of the object examples are instances of the same class Undergrad-student which has three parts called ssn, gpa, and major. In the abstracted class dictionary graph, class Area is an alternation class. Notice how this class was discovered: In the examples, part major of Undergrad-student contains objects belonging to two different classes.

In the second step, the class dictionary graph in Fig. 5f is minimized in two phases keeping the set of defined objects invariant. The goal of the minimization phases is to minimize the number of edges in the class dictionary graph to keep the software which will be attached to the classes easy to manage. In the first phase we factor out common parts by introducing new alternation classes if necessary. Fig. 6a shows the factored class dictionary from Fig. 5f. (CNF stands for Common Normal Form and means that all common parts have been optimally factored, that is we have a minimum number of construction edges.)

In the second minimization phase, we optimize the inheritance structure further. Fig. 6b shows the fully optimized class dictionary graph, equivalent to Fig. 5f. We observe a decreased number of alternation edges with respect to Fig. 6a. We also observe that, as a consequence, the number of incoming alternation edges to any construction vertex has been reduced. This is another criterion to determine when a class dictionary graph is optimal and it means that the amount of multiple inheritance is minimal. Minimizng alternation edges is in general NP-hard.[3] We can nevertheless obtain good approximations of an optimal class dictionary graph efficiently and in this paper we present an efficient approximation algorithm.

Our algorithms can be used to analyze existing object-oriented programs regarding the proper use of abstraction. Object examples which are used by existing object-oriented programs can be given as inputs to the abstraction algorithms. The automatically generated class definitions help to evaluate the class definitions in the existing programs. The algorithms are easy to teach and can be applied by hand.

[3] However, when the result is a single inheritance hierarchy, the task can be solved in polynomial time [LBSL90b].

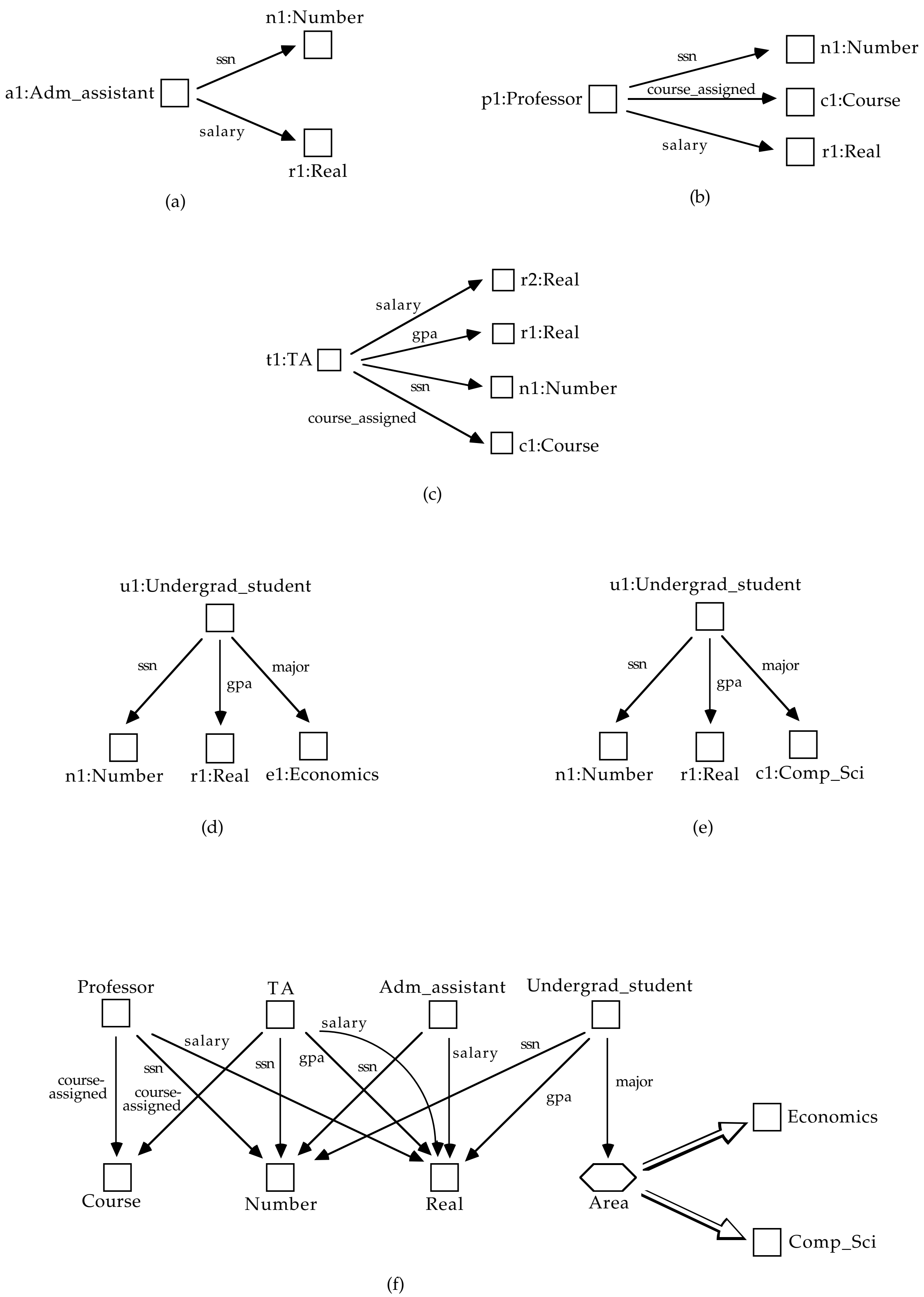
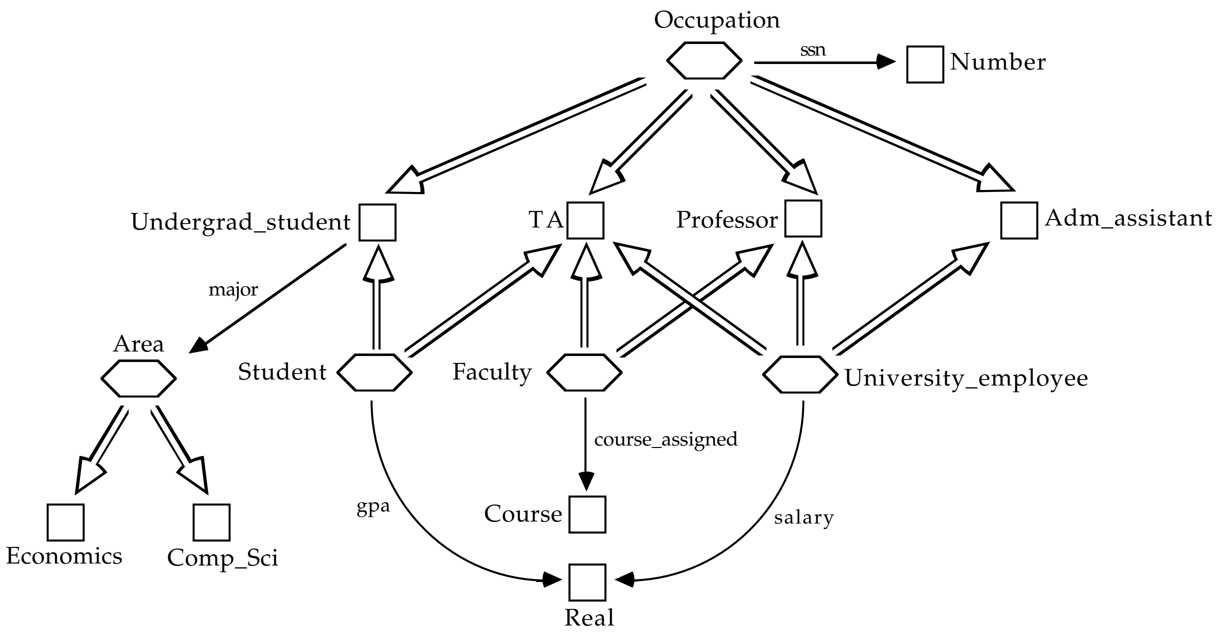
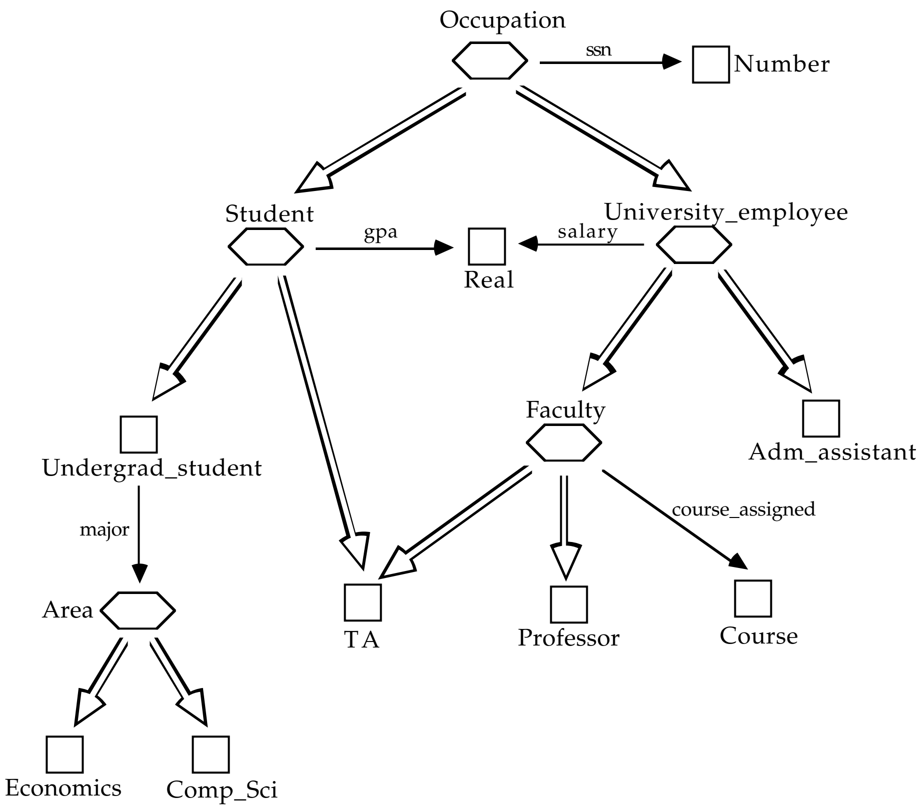


Figure 5: MCDL-P Example



(a)



(b)

Figure 6: MCDL-P Example: optimized

This paper is part of our research program, called the Demeter System^{TM} project (see [WBJ90] for a survey). In section 2 we present the theoretical foundations of the Demeter data model (ignoring its language definition capabilities since we don't need them for describing design algorithms). Section 3 discusses problems and their algorithms for object-oriented design while section 4 discusses the practical relevance of our algorithms from a software engineering view point. In section 5 we compare our work with related work in AI, data bases and object-oriented software engineering. The appendix contains several NP-completeness proofs.

2 Structures for object-oriented design

We introduce mathematical objects which are fundamental to object-oriented design and programming. A mathematical object is described as a tuple consisting of sets and relations defined on the sets. A set of axioms defines minimal properties which the mathematical objects have to satisfy.

For more information on mathematical objects, see the appendix.

The following definitions will be mathematical and perforce abstract. However they are essential for presenting our algorithms and other formal results and they may also help to topple the Tower of Babel that looms over object-oriented models and languages.

There has been considerable debate about the definition of object-oriented systems and there is a lack of agreement on a formal foundation. [ABW+89, Dit90] describe characteristics which such a formal model should possess and we view our formal model, which we call the Demeter kernel model, a contribution towards the goals described in the two papers. The Demeter kernel model has one distinguishing feature compared to other formal models which have appeared: It offers a powerful capability to populate object-oriented databases [Lie88]. We call our model the Demeter kernel model since we developed it during our work with the Demeter System (starting in 1984, [Lie88]) and since it only describes the structural parts of useful object-oriented data models. Next we describe the Demeter kernel model which consists of class dictionary graphs and corresponding object graphs.

2.1 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 paper needs further explanation. A part object does not have to be a physical part; any attribute of an object is a part of it. 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, 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.

Graph		*An object-oriented design*	
vertices	construction	classes	instantiable
	alternation		abstract
edges	construction	class relationships	part-of relationships labels are part names
	alternation		inheritance relationships

Table 1: Standard Interpretation of class dictionary graphs

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].

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 1 *A* **class dictionary graph**[4] 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*

[4] 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.

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. enddefinition

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

begindefinition 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.*

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

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

Example 1 *Fig. 7 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) },
```

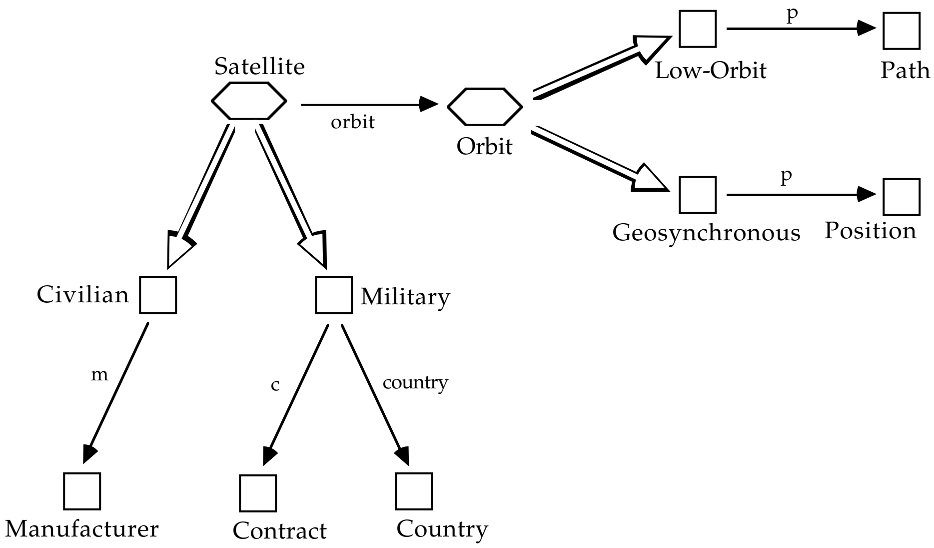


Figure 7: Satellites

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 talk about the components of class dictionary graph ϕ.

For notational convenience, we overload the term *vertex* used for class dictionary graphs. Sometimes we mean by a vertex a labeled vertex of a class dictionary graph, and sometimes we mean the label of a vertex. The context implies which of the two interpretations we mean. As an example, consider the following hypothetical theorem:

Theorem 1 *Let ϕ and ψ be two class dictionary graphs, where $\phi = (VC_\phi, VA_\phi, \Lambda_\phi; EC_\phi, EA_\phi)$, $\psi = (VC_\psi, VA_\psi, \Lambda_\psi; EC_\psi, EA_\psi)$. Class dictionary graphs ϕ and ψ are* **object-equivalent** *if*

$$VC_\phi = VC_\psi$$

and for all $v \in VC_\phi$ the property

$$PartClusters_\phi(v) = PartClusters_\psi(v).$$

holds ($PartClusters(v)$ is defined elsewhere).

In the definition of ϕ and ψ, VC_ϕ and VC_ψ are sets of labeled vertices. In the equation

$$VC_\phi = VC_\psi,$$

we have omitted an explicit label extraction function; in the context of this equation we interpret VC_ϕ and VC_ψ as sets of labels and the equation expresses that the two sets of labels have to be the same.

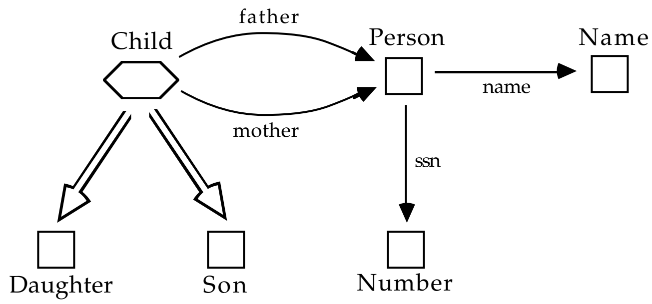


Figure 8: Child

In the expressions "for all $v \in VC_\phi$", v ranges over all labels and in

$$PartClusters_\phi(v) = PartClusters_\psi(v),$$

v stands for a labeled vertex in the appropriate graph.

This overloading saves complexity in our formulas, since without overloading the above equalities would have to be written:

$$Label(VC_\phi) = Label(VC_\psi),$$

$$PartClusters_\phi(Vertex_\phi(v)) = PartClusters_\psi(Vertex_\psi(v)).$$

Label is a function which extracts the label from labeled vertices and $Vertex_\phi$ is a function which expands a label into a labeled vertex in class dictionary graph ϕ. Since there is a bijection between graph vertices and labels, it is justified to overload the meaning of *vertex* in class dictionary graphs for notational convenience.

Later we give a list of axioms 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 unfortunately, it is not as concise as a textual notation. 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. The textual notation is several factors more concise than the graphical notation. 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 Fig. 8

is described by

```
Child :
```

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

// 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.

The construction vertex Person in Fig. 8 is described by:

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

Example 2 *The following text*

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

describes the class dictionary graph in Fig. 9:

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)},
```

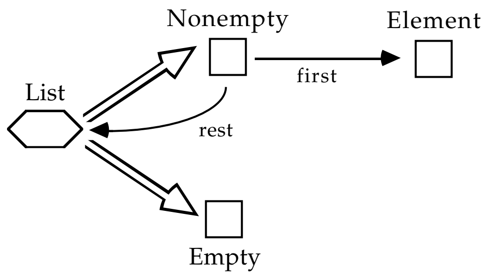


Figure 9: List

Λ = `{first, rest}`.

Definition 2 *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 an* **alternation path of length** *$k \geq 1$, if there exists a path of length $k \geq 1$ from v_0 to v_k such that for $0 \leq j \leq k-1 : (v_j, v_{j+1}) \in EA$. We say that vertex v_k is alternation-reachable from itself. The path consists of the sequence of edges.*

Informally, an alternation path is a path which only contains alternation edges.

Vertex $v_k \in V$ in a class dictionary graph $\phi = (V, \Lambda; EC, EA)$ is said to be **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, $0 \leq j \leq k-1 : (v_j, v_{j+1}) \in EA$ or $\exists w \in V, \exists l \in \Lambda$ such that v_j is alternation-reachable from w and $(w, v_{j+1}, l) \in EC$. The path consists of the sequence of edges.*

We say that vertex v_k is reachable from itself. A class dictionary graph is **cycle-free** *if for all $v \in V$ there is no path from v to v.*

Vertex $v_k \in V$ in a class dictionary graph $\phi = (V, \Lambda; EC, EA)$ is said to be **construction-reachable** *from vertex $v_0 \in V$ via a* **construction path of length** *$k \geq 1$, if there exists a path of length $k \geq 1$ from v_0 to v_k such that for all j, $0 \leq j \leq k-1$ $\exists w \in V, \exists l \in \Lambda$ such that v_j is alternation-reachable from w and $(w, v_{j+1}, l) \in EC$. The path consists of the sequence of edges.*

We say that vertex v_k is construction-reachable from itself.

Informally, a construction path is a path which only contains construction edges.

Note that all paths, alternation paths and construction paths must contain at least one edge.

A vertex w is alternation-reachable from alternation vertex v means in the standard interpretation that w inherits from v or $v = w$.

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

Definition 3 *A class dictionary graph $\phi = (VC, VA, \Lambda;\ EC, EA)$ is* **legal** *if it satisfies the following axioms ($V = VC \cup VA$):*

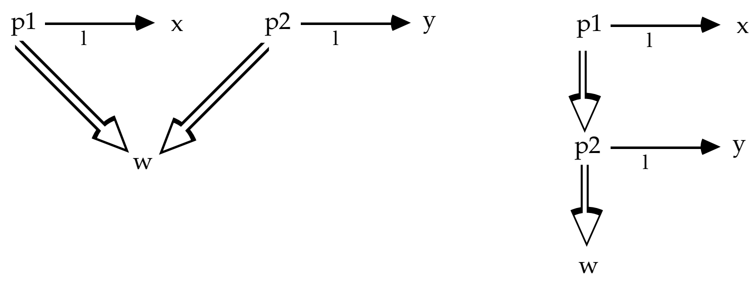


Figure 10: Forbidden subgraphs

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.

2. *Unique labels axiom:*

   $\forall w \in V$ there are no $p_1, p_2 \in V$ s.t. $\exists x, y \in V, \exists l \in \Lambda$ 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.

3. *Cycle-free subgraph axiom:*

   For all $v \in V$ there exists at least one cycle-free partial class dictionary graph anchored at v.

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

2.1.1 Discussion of axioms

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

The Unique Label Axiom guarantees that "inherited" construction edges are uniquely labeled and excludes class dictionary graphs which contain the patterns shown in Fig. 10. 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.

We list some important results of the theory of class dictionary graphs without explaining them in detail. The theory is useful to designers and implementors of CASE tools for object-oriented programming. (In what follows, P (NP) is the class of problems which can be solved in deterministic polynomial (non-deterministic polynomial) time [GJ79].)

The results so far fall into three categories:

- Computing application development plans. Applications are developed in an incremental way, starting out with a minimal or minimum subset of classes and then adding in each step a few more classes until all classes are implemented. Such a "growth plan" is naturally determined by a class dictionary graph. A minimal or minimum subset of classes is formalized by a minimal or minimum class subdictionary graph. A minimal class subdictionary graph is a class dictionary graph which cannot be made smaller while still preserving the class subdictionary graph property (local property). On the other hand, a minimium class subdictionary graph is a class dictionary graph which is smallest (least number of edges) among all class subdictionary graphs (global property).

  The results are that finding a minimum class subdictionary graph is NP-hard [LW89] while finding a minimal class subdictionary graph can be solved in polynomial time.

- Optimizing classes. A set of classes defines a set of objects in a non-unique way since there are many different sets of classes which define the same set of objects. The question is how can we select an optimum set of classes which is "object-equivalent" to a given set of classes. The results are: 1. Minimizing a class dictionary graph is NP-hard [LBSL90c]. 2. Minimizing a class dictionary graph is in P for single-inheritance class hierarchies [LBSL90b]. Our optimization algorithms are useful for improving class organizations so that they are "good" from a software engineering point of view.

- Inductive inference. Objects are simpler than the corresponding classes. We have developed a polynomial algorithm for abstracting a class dictionary graph from example object graphs [LBSL90b]. The algorithm computes an "optimal" inheritance structure. Our inductive inference algorithms are useful for reverse engineering and checking the adequacy of test cases.

2.1.2 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++ [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 Fig. 11. We want to implement a pocket calculator which evaluates the object equivalent of expressions such as 3, (+3(+2 1)), (*3(+2 1)). We implement the calculator in phases by starting with the smallest class subdictionary graph anchored at `Exp` which means that in phase 0 we implement the classes `Exp`, `Simple`, and `Number`. Then we choose the next larger class subdictionary graph which contains those classes, i.e., in phase 1 we implement additionally the classes `Compound`, `Op`, and `Addsym`. In the last phase we add class `Mulsym`. The complete C++ program which has to be written by the user is given in Table 2. The missing parts of the C++ program are generated from the class dictionary graph in Fig. 11 by the Demeter System.

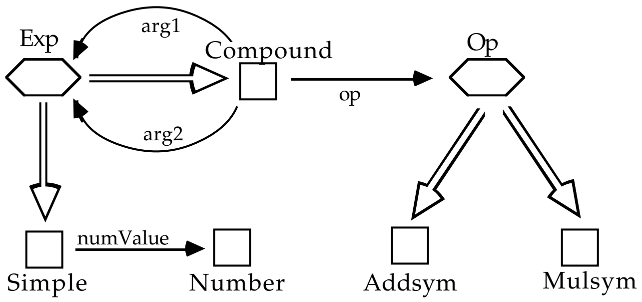


Figure 11: Prefix expression class dictionary graph (user-written)

Growth phases		
phases	*user-written code*	*test inputs*
0	int Exp::eval(){} // virtual	3

	int Simple::eval() {
	return numvalue→eval();
	}	

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

1	int Compound::eval() {
	return op→apply_op(arg1→eval(),arg2→eval());
	}
		(+ 3 (+ 2 1))

	int Op::apply_op(int n1,int n2) {} // virtual	

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

2	int Mulsym::apply_op(int n1,int n2) {
	return n1 * n2;
	}
		(* 3 (+ 2 1))

Table 2: Pocket calculator C++ implementation

```
v' ========> w
|     l
|                    IS PICTURE UPDATED?
|
|
V
v
```

Figure 12: Parts

2.2 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 3.

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

Table 3: Standard interpretation for object graphs

Next we introduce three functions defined for class dictionary graphs. Those functions are used to define the structural semantics of inheritance.

begindefinition Let $\phi = (V, \Lambda;\ EC, EA)$.

For $v \in V$: $Parts(v) = \{(l, w) \mid l \in \Lambda, w \in V, (v', w, l) \in EC$, and v is alternation-reachable from $v'\}$ (see Fig. 12). An element of $Parts(v), v \in V$ is called a **part** of v.

$Parts(\phi) = \bigcup_{v \in VC} Parts(v)$.

$PartLabels(v) = \{l \mid \exists w \in V\ s.t.\ (l, w) \in Parts(v)\}$.

$PartVertices(v) = \{w \mid \exists l \in \Lambda\ s.t.\ (l, w) \in Parts(v)\}$.

enddefinition

$Parts(v)$ is a list of pairs where each pair consists of a label and a class of some part of v, including inherited parts.

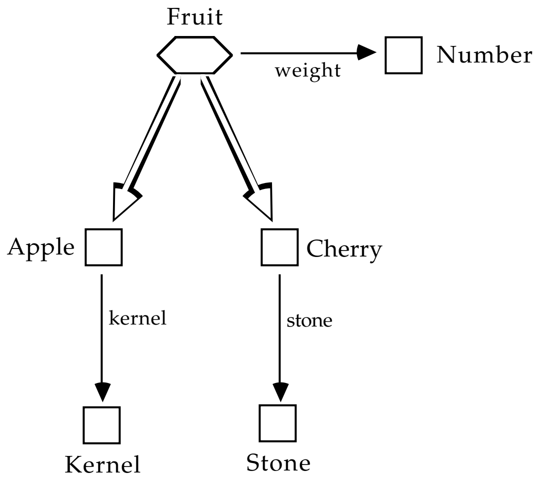


Figure 13: Fruit class dictionary graph

Example 3 *We demonstrate the definitions with the following class dictionary graph (see Fig. 13):*

```
Fruit : Apple | Cherry *common* <weight> Number.
Apple = <kernel> Kernel.
Cherry = <stone> Stone.
Number = .
Kernel = .
Stone = .
```

$Parts(Apple) = \{(kernel, Kernel), (weight, Number)\}$.
$PartLabels(Apple) = \{kernel, weight\}$.
$PartVertices(Cherry) = \{Stone, Number\}$.

The following definition relates 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 which follow ask for even more discipline than C++.

Definition 4 *An* **object graph with respect to a class dictionary graph** $\phi = (V, \Lambda;\ EC, EA)$ *is a graph* $H = (W, S, \Lambda_H;\ E, \lambda)$ *with vertex set* W *and* $S = VC_\phi$*.* E *is a ternary relation on* $W \times W \times \Lambda_H$*.* Λ_H *is a set of edge labels. The function* $\lambda : W \to VC_\phi$ *maps each vertex of* H *to a construction vertex of* ϕ*. If* $(v, w, l) \in E$*, we call* l *the label of the labeled edge (v,w,l).* $\Lambda_H \subseteq \Lambda$*. enddefinition*

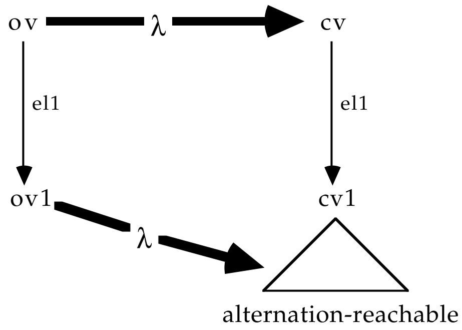


Figure 14: Legality rule

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.

begindefinition An object graph $H = (W, S, \Lambda_H;\ E, \lambda)$ *is* **legal** *with respect to a partial class dictionary graph* $P = (VC_P, VA_P, \Lambda_P;\ EC_P, EA_P, EI_P)$ *anchored at some vertex, where* $S = VC_P$*, if* $\forall \mu \in W\ \exists v \in S$ *s.t.* $\lambda(\mu) = v$ *and*

1. $|PartClusters(v)|=$ *the number of edges outgoing from* μ
2. $\forall (\mu, \nu, l) \in E : \exists (l, A) \in PartClusters(v)$ *s.t.* $\lambda(\nu) \in A$
3. $\forall (l, A) \in PartClusters(v) : \exists \nu \in W$ *s.t.* $\lambda(\nu) \in A$ *and* $(\mu, \nu, l) \in E$.

In database terminology, $Objects(\phi)$ represents all instances of object base schema ϕ.

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.

```
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:

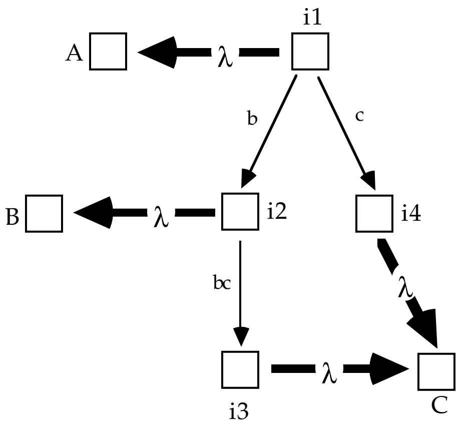


Figure 15: Object graph

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

where inst2 is an object identifier defined elsewhere. Each object identifier has to be defined once.

Example 4 *Consider the class dictionary graph:*

```
A = <b>B <c>C.
B = <bc> C.
C =.
```

The object graph

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

is legal with respect to the class dictionary graph (see Fig. 15). The object graph is given by: $W = \{i1, i2, i3, i4\}$, $E = \{(i1, i2, b), (i1, i4, c), (i2, i3, bc)\}$, $\Lambda_H = \{b, bc, c\}$, $\lambda = \{i1 \rightarrow A, i2 \rightarrow B, i3 \rightarrow C, i4 \rightarrow C\}$.

If vertices are not shared, there is no need for naming them explicitly. Therefore, the above object graph can also be described by:

```
:A(
  <b> :B (
    <bc> :C())
  <c> :C())
```

The vertices are now named implicitly, e.g., by prefixing the class names with i (for *i*nstance) and by numbering them, if needed. The object graph is now:

$W = \{iA, iB, iC1, iC2\}, E = \{(iA, iB, b), ...\}$.

Example 5 *Consider the illegal object graph with respect to the class dictionary graph in Fig. 13:*

```
:Apple(
  <stone> :Stone()
  <weight> :Number)
:Apple(
  <kernel> :Kernel()
  <weight> :Cherry)
```

The first object is illegal since apples don't contain stones and the second, because Cherry *is not alternation-reachable from* Number.

We have defined object graphs and their relationship to class dictionary graphs. We need a second kind of object graph, called an object example graph, which has a life independent of a class dictionary graph. Object example graphs are useful for describing objects from which we want to abstract classes. An object example graph is described in terms of a set S of classes whose relationships will be computed by the learning algorithm.

begindefinition An **object example graph with respect to a set S** is a graph $H = (W, S, \Lambda_H;\ E, \lambda)$. W, Λ_H, E are given as in definition 4. $\lambda : W \rightarrow S$ 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 or more 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 in Λ_H or no outgoing edges at all.

enddefinition

An object graph with respect to a class dictionary graph DG is an object example graph with respect to set VC_{DG}.

2.3 Class dictionary graph 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.

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.

Class dictionary graph learning is formally treated in [LBSL90d]. We repeat here enough of the definitions and the learning algorithm to make this paper self-contained. To formalize the bottom-up design process we need the following definitions:

Definition 5 *An object example graph $H = (W, S, \Lambda_H; E, \lambda)$ is* **similar** *to a set of object example graphs Ω, if the set of object example graphs $\Omega \cup \{H\}$ is* legal *and if for each edge $(u, u', l) \in E$ with $\lambda(u) = w$ and $\lambda(u') = w'$ there is an example graph $\Omega_i \in \Omega$ which contains an edge (v, v', l) where $\lambda(v) = w$ and $\lambda(v') = w'$.*

CDL-P (CLASS DICTIONARY LEARNING)
INSTANCE: A set of legal object example graphs Ω.
SEARCH: Find a class dictionary graph ϕ so that if OG is an arbitrary object example graph with respect to ϕ ($S = VC_\phi$), OG is similar to Ω.

Next we define the size of a class dictionary graph.

Definition 6 *The* **edge-size** *$edge\text{-}size(DG)$ of a class dictionary graph DG is the number of construction edges plus one quarter of the number of alternation edges.*

Note: 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.

2.3.1 Motivation for similarity

The **CLASS DICTIONARY LEARNING** problem is defined in terms of similar objects to avoid the learning of a class dictionary graph which defines a much larger set of objects than the examples intend to describe. The two class dictionary graphs in Fig. 16 have the same edge-size, but the first one defines a much larger set of objects than the second class dictionary graph.

```
Alt : Compound | Numerical | Variable |
      Addsym | Mulsym.
Compound = <op> Alt <arg1> Alt <arg2> Alt.
Numerical = <n> Number.
Variable = <v> Ident.
Addsym = .
Mulsym = .
```

edge-size: $5/4 + 5 = 6\ 1/4$

```
Exp : Compound | Numerical | Variable.
Op : Addsym | Mulsym.
Compound = <op> Op <arg1> Exp <arg2> Exp.
Numerical = <n> Number.
Variable = <v> Ident.
Addsym = .
Mulsym = .
```

edge-size: $5/4 + 5 = 6\ 1/4$

Figure 16: Motivation for similarity

CDL-P can be solved efficiently by the algorithm described in [LBSL90b].

An example explains how the algorithm works and shows how to learn the second class dictionary in Fig. 16 from object examples.

Example 6 *As input we use the following object examples from which the basic abstraction algorithm abstracts:*

```
:Numerical(<n> :Number)

:Variable(<v> :Ident)

:Compound(
    <op> :Mulsym() <arg1> :Numerical <arg2> :Variable)

:Compound(
    <op> :Addsym() <arg1> :Variable <arg2> :Numerical)

:Compound(
    <op> :Addsym() <arg1> :Compound <arg2> :Compound)
```

After the first three examples we have learned the class dictionary graph (asuming that Number *and* Ident *are predefined):*

```
Compound = <op> Mulsym <arg1> Numerical <arg2> Variable.
Mulsym = .
Numerical = <n> Number.
Variable = <v> Ident.
```

After the fourth example we have learned (without repeating classes which have not changed):

```
Compound =
  <op> Compound_op
  <arg1> Compound_arg1 <arg2> Compound_arg2.
Compound_op : Mulsym | Addsym.
Compound_arg1 : Numerical | Variable.
Compound_arg2 : Variable | Numerical.
Addsym = .
```

After the fifth example we have learned:

```
Compound_arg1 : Numerical | Variable | Compound.
Compound_arg2 : Variable | Numerical | Compound.
```

Simplification and renaming gives as result:

```
Compound =
  <op> Op
  <arg1> Exp <arg2> Exp.
Op : Mulsym | Addsym.
Mulsym = .
Addsym = .
Exp : Numerical | Variable | Compound.
Numerical = <n> Number.
Variable = <v> Ident.
```

Note that the first class dictionary graph in Fig. 16 would define too many objects; it would violate the similarity constraint on the learning process.

2.3.2 Object example notation

The object example notation needs to be contrasted with the object graph notation. Although they are syntactically very similar, there are some subtle differences. The goal of the object graph notation is to describe objects which might contain shared subobjects while the goal of the object example notation is to define information from which we want to abstract classes. In the latter case there is no need to express objects with shared subobjects. In object graphs we also want to describe values for terminal classes such as Ident and Number while in object example graphs this information would not be useful.

Therefore, an object example description has the following syntactic form:

```
:v0(
  <successor1> :v1 ( ... )
  <successor2> :v2 ( ... )
  ...
  <successorn> :vn ( ... ))
```

where the (...) may be omitted. This means that an object example description may have all objects only nested to one level.

This capability is useful since it allows shorter object descriptions for reasons to be explained later. It also allows to learn class dictionary graphs which violate the cycle-free subgraph axiom. Consider the following two object graphs:

```
:A(<b> :B)
:B(<b> :A)
```

The abstracted class dictionary graph is illegal:

```
A = <b> B.
B = <a> A.
```

as pointed out by the Cycle-Free Subgraph Axiom.

2.3.3 Practicality of class dictionary graph learning

Our learning algorithm is only practically useful if it does not require too many examples and if the examples are easy to find. Fortunately, the algorithm has both properties. We state an upper bound on the number of examples needed by the basic algorithm in terms of the learned class dictionary:

For each construction vertex v we need at most as many examples as the maximum $N(v)$ over all parts of the number of construction vertices which are alternation-reachable from each part vertex. More precisely, $N(v) = max\{a|a =$ number of construction vertices alternation-reachable from w, where $w \in PartVertices(v)\}$. For a construction class without parts we need one example. The structure of the parts is not specified by the examples and each example only contains as many objects as it has parts.[5] A class A with no parts needs one example: A(). For a further discussion see section 2.5.

Example 7 *For the following class dictionary graph*

```
Compound =
  <op> Op
  <arg1> Exp <arg2> Exp.
Op : Mulsym | Addsym | Subsym.
Exp : Numerical | Variable | Compound.
Mulsym = .
Addsym = .
Subsym = .
```

we need at most 8 examples and a total of 14 objects:

```
Compound : 3 examples, each containing 3 objects
Mulsym   : 1 example, each containing 1 object
Addsym   : 1 example, each containing 1 object
Subsym   : 1 example, each containing 1 object
Numerical: 1 example, each containing 1 object
Variable:  1 example, each containing 1 object
```

This is only an upper bound since we can learn the above class dictionary graph from:

```
:Compound(
  <op> :Subsym()
  <arg1>
    :Compound(
      <op> :Mulsym()
      <arg1> :Numerical(<n> :Number)
      <arg2> :Variable(<v> :Ident))
  <arg2>
    :Compound(
```

[5]This restriction is only theoretical, aimed at obtaining an upper bound. Object example graphs can have as deep a structure as desired.

```
<op> :Addsym()
<arg1> :Variable
<arg2> :Numerical))
```

which contains only 1 example with 12 objects. Object example graphs can have as deep a structure as desired.

It is important for the above upper bound that

```
:A(
  <b> :B ( xxx-something )
  <c> :C())
:A(
  <b> :B ( yyy-something )
  <c> :C())
```

(6 objects) can be abbreviated as (5 objects)

```
:A(
  <b> :B    // B is defined elsewhere
  <c> :C())  // C can have no parts
:B( xxx-something )
:B( yyy-something )
```

We can treat each construction class at the outermost level.

2.3.4 Learning better class dictionary graphs

Next we focus on learning a minimum class dictionary graph from examples.

MCDL-P (MINIMUM CLASS DICTIONARY LEARNING)
INSTANCE: A set ω of legal object examples.
SEARCH: Find a class dictionary graph ϕ so that $Objects(\phi)$ contains only objects similar to objects in ω and for which $edge\text{-}size(\phi)$ is minimum.

MCDL-P is a minimization version of **CDL-P** where the class dictionary graph edge-size is minimized. **MCDL-P** is NP-hard which is proven in the appendix. We use the following combined algorithm for abstracting class definitions from object examples:

1. Apply algorithm CDL-A.
2. Class dictionary graph minimization: Apply algorithm ACP followed by algorithm Consolidate, both defined subsequently.

This algorithm is fast but only provides an approximate solution to **MCDL-P**.

2.4 Class dictionary graph minimization

To formalize the concept of class dictionary graph minimization, we need the following definitions.

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

Object equivalence can be expressed purely in graph-theoretic terms without referring to objects.

In what follows we will describe the main components of class dictionary graph minimization by outlining one of the relationships that came up from our investigation, namely: $CDMS(\phi) = CAS(CNFS(\phi))$.

2.4.1 Common normal form

We introduce a normal form for class dictionary graphs. To this end, we consider the equivalence relation on construction edges given by the parts defined by the set of construction edges of a class dictionary graph.

Definition 8 *Let (u, w_1, l_1), $(v, w_2, l_2) \in EC_\phi$ for some class dictionary graph ϕ. (u, w_1, l_1)* **part-equivalent** *(v, w_2, l_2) iff $\mathcal{A}(w_1) = \mathcal{A}(w_2)$ and $l_1 = l_2$, where $\mathcal{A}(w)$ is the set of construction vertices alternation-reachable from w.*

We say, for a particular equivalence class of size greater than 1 of the part-equivalent relation[6], that all but one of the parts defined by the edges in the equivalence class are redundant. Informally, a class dictionary graph is in common normal form if it does not contain redundant parts.

Definition 9 *A class dictionary graph ϕ is in* **common normal form (CNF)** *if for all e_1, $e_2 \in EC_\phi$, e_1 part-equivalent $e_2 \Rightarrow e_1 = e_2$.*

The following problem formalizes the transformation of a class dictionary graph into an equivalent class dictionary graph in common normal form.

CNF-P (COMMON NORMAL FORM)
INSTANCE: A legal class dictionary graph ϕ.

[6] It is straightforward to verify that part-equivalent is an equivalence relation, and the exercise is left to the reader.

SEARCH: Find a legal class dictionary graph which is object-equivalent to ϕ and which is in common normal form.

A simple procedure for transforming a class dictionary graph to common normal form can be obtained by the following apparatus:

CNF Rule.

1. Select an equivalence class R of part-equivalent such that $|R| > 1$.
2. Choose one construction edge, $(v, w, l) \in R$, and replace every other edge, $(\eta, w', l) \in R$, where $w \neq w'$ with an edge (η, w, l). Note that this replacement preserves object equivalence since $\mathcal{A}(w_i) = \mathcal{A}(w_j)$, for $1 \leq i, j \leq |R|$. Now, every edge in R is of the form (η_i, w, l), for $1 \leq i \leq |R|$.
3. Introduce a new alternation vertex v, and $|R|$ alternation edges $(v, \eta_1), (v, \eta_2), ..., (v, \eta_{|R|})$, where $(\eta_i, w, l) \in R$, for each η_i, if such an alternation vertex does not already exist. Delete all edges (η_i, w, l) for $1 \leq i \leq |R|$, and add edge (v, w, l). Note that for $1 \leq i \leq |R|$, $(l, w) \in Parts(\eta_i)$ still holds since η_i is alternation reachable from v.

Lemma 1 *For all legal class dictionary graphs ϕ there is a sequence σ of applications of CNF rule and there is a class dictionary graph ϕ_1 in $CNFS(\phi)$ such that ϕ_1 can be obtained by applying σ to ϕ.*

Proof:
Let ϕ_1 be an arbitrary class dictionary graph, and σ_1 the sequence of applications of CNF rule that contains exactly one application of CNF rule for each different part in $Parts(\phi_1)$. After the application of CNF rule corresponding to part p_i, there will be only one occurrence of p_i in ϕ_1, therefore p_i will not be redundant. $\Box$

It follows that **CNF-P** can be solved in polynomial time.

Notice that to bring a class dictionary graph to CNF, it might be necessary to introduce new alternation vertices, although class dictionary graph edge-size is effectively decreased since redundant parts are more costly than alternatives. We give a fast algorithm, called "ACP" (for abstraction of common parts), for transforming to common normal form, i.e., for solving **CNF**. This algorithm is not as simple as possible but it behaves well in practice in combination with the consolidation of alternatives algorithm.

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 $(v, w) \in EA$.

Algorithm AR(v) (Alternation Reachable)

1. If v is marked "AR-DONE", return S_v.

2. If $v \in VC$ then $S_v = \{v\}$.

3. Else

   (a) $S_v := \{\ \}$

   (b) For each w, where $(v, w) \in EA$, $S_v := S_v \cup$ ACP-Vertex(w)

4. 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 $(v, w) \in EA$, call ACP-Vertex(w)

3. While there is a label l and a set of construction edges, E, such that $\forall(v, w) \in EA$: $\exists(w, u, l) \in EC$ where the set of construction edges alternation-reachable from u is equal to E (we say the part (l, u) is redundant in every alternative of v), replace one such construction edge, (w, u, l), with the new construction construction edge (v, u, l), 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', u, l) and alternation edge (v, v').

   (c) For each $w \neq v'$ such that $(v, w) \in EA$ and $(w, u', l) \in EC$ where the set of construction vertices alternation-reachable from u is equal to the set alternation-reachable from u', delete edges (w, u', l) and (v, w) and add the alternation edge (v', 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', u, l).
   (b) For each w alternation-reachable from v where $(w, u', l) \in EC$ where the set of construction edges alternation-reachable from u is equal to the set alternation-reachable from u', delete edge (w, u', l) and add the alternation edge (v', w). If w is an alternative of v, then also replace the alternation edge (v, w) with (v, v').
   (c) Call ACP-Vertex(v').
6. Mark v "ACP-DONE" and return.

Example 8 *We demonstrate the normal form transformation by algorithm ACP with the following class dictionary graph:*

```
Element : Coin | Brick | Box.
Coin =
  <weight> Number <position> Vector.
Brick = <color> Ident
  <width> Number <height> Number <length> Number
  <weight> Number <position> Vector.
Box = <contents> ElementList
  <width> Number <height> Number <length> Number
  <weight> Number <position> Vector.
```

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:

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

The resulting class dictionary graph is:

```
Element : Coin | Brick | Box
  *common* <weight> Number <position> Vector.
Coin = .
Brick = <color> Ident
  <width> Number <height> Number <length> Number.
Box = <contents> ElementList
  <width> Number <height> Number <length> Number.
```

which is not yet in common normal form. We introduce a new class:

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

Now the class dictionary graph is in common normal form:

```
Element : Coin | Brick | Box
  *common* <weight> Number <position> Vector.
QuadrangularElement : Brick | Box
  *common* <width> Number <height> Number <length> Number.
Coin = .
Brick = <color> Ident.
Box = <contents> ElementList.
```

2.4.2 Consolidation of alternatives

Following the definition of a class dictionary graph, a second component of class dictionary graph minimization has to be concerned with alternation vertices. It turns out that alternatives defined by alternation vertices can be redundant as well, although in a less clear cut way than parts defined by construction vertices.

CA-P (CONSOLIDATION OF ALTERNATIVES)
INSTANCE: A legal class dictionary graph ϕ.
SEARCH: Find a class dictionary graph ϕ_1 such that the total number of alternation edges of ϕ_1 is minimal and so that the following two conditions hold: 1. ϕ and ϕ_1 are object-equivalent. 2. ϕ_1 has the same number or less construction edges than ϕ.

CA-P is NP-hard which is shown in the appendix.

The algorithm presented in this section is fast but only provides an approximate solution for **CA-P (CONSOLIDATION OF ALTERNATIVES)**.

There are two aspects to consolidating alternatives:

- 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:

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

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

can be abbreviated to

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

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 a 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 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 9 *The following example shows how the algorithm recognizes common triples:*

```
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 $O(p^2)$ alternations 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.

CDM-P (CLASS DICTIONARY MINIMIZATION)
INSTANCE: A class dictionary graph $G = (V, \Lambda; EC, EA)$.
SEARCH: Find a class dictionary graph $G' = (V', \Lambda; EC', EA')$ object-equivalent to G, such that $|EC'| + |EA'|$ is minimum.

CDM-P is NP-hard which is shown in the appendix. A useful approximation algorithm is to first use algorithm ACP followed by algorithm Consolidate Alternatives.

2.5 Object example generation

To get a better understanding of the abstraction algorithms, we also consider their inverse. The abstraction algorithms produce a class dictionary graph from a set of object examples. How can we find a set of object examples for a given class dictionary graph so that the object examples contain all the essential information necessary to (almost perfectly [7]) retrieve the class dictionary graph? This analysis is important to understand that our algorithm needs only a small number of training examples.

OEG-P: (OBJECT EXAMPLE GENERATION)
INSTANCE: A legal class dictionary graph ϕ.
SEARCH: Find a set ω of legal object examples such that ϕ is object-equivalent to all elements in **CDLS**(ω).

OEG-P can be solved in polynomial time by the following algorithm OEG-A. For a given construction vertex γ compute the number, called $EXAMPLES(\gamma)$, as follows: determine

[7] The names of alternation classes will be generated by the algorithm.

the maximum over each part vertex η of γ of the number of construction vertices which are alternation reachable from η. For a construction vertex γ with 0 parts, $EXAMPLES(\gamma) = 1$. For vertex γ we generate $EXAMPLES(\gamma)$ object examples which have the parts determined by $Parts(\gamma)$. In each generated object example all alternatives for the parts must be tried, however the composition of the part classes is left unspecified (only use the class name). For each object o created and for each part s_k, create a part for o with a value from the list of construction classes alternation reachable from s_k. If the list of alternation-reachable construction classes is not long enough, start repeating values from the beginning. If it contains a single element, repeat it every time.

Example 10 *Consider the class dictionary graph ϕ:*

```
A1 : A | B *common* <b1> B1.
B1 : D | E .
A = .
B = <d> D.
D = .
E = .
```

OEG-A(ϕ) is:

```
A( <b1> D ), A( <b1> E )
B(<d> D <b1> D), B(<d> D <b1> E)
D()
E()
```

The following theorem shows that the algorithm OEG-A achieves its goal:

Theorem 2 $\forall$ *class dictionary graphs ϕ : ϕ is object-equivalent to CDL(OEG-A(ϕ)).*

Proof.
By induction on the depth of objects. $\Box$

2.6 Relationships

To find good practical algorithms for the abstraction problems and to study the complexity of them, we investigate relationships between the basic problems.

Theorem 3 *For all legal class dictionary graphs ϕ: $CAS(CNFS(\phi)) = CDMS(\phi)$.*

```
     G1
    /  \
   /    \
  /      \
 /        \
/
G----G2----> minimum
\
 \        /
  \      /
   \    /
    \  /
     Gn

CNF   CA
```

Figure 17: Minimization

Proof:
Consider figure 17. We start with a class dictionary graph G and we transform it to CNF by minimizing the construction edges. The only way to minimize the construction edges in an object-equivalent way is to move them up along existing or new alternation edges. There are many class dictionary graphs which are in CNF and object-equivalent to the original. Each contains "some consolidation of alternatives".

Once we have minimized the construction edges, we minimize the alternation edges, keeping the construction edges at the minimum. Usually several non-isomorphic class dictionary graphs with the same minimum number of edges can be obtained. It is conceivable that by adding construction edges, we could get a lower minimum. But to keep the class dictionary graph object-equivalent we could only add construction edges by introducing redundant parts. It is always cheaper to factor the redundant parts because

```
A : B | C *common* <d> D.
B = .
C = .
```

(edge-size: 1.5)

is smaller in edge-size than

```
B = <d> D.
C = <d> D.
```

(edge-size: 2)

Therefore there is no trade-off between construction and alternation edges (i.e., we can not add extra construction edges to get a better consolidation of alternatives). $\Box$

Theorem 4 *For all legal object examples* $\omega : MCDLS(\omega) = CDMS(CDLS(\omega))$

Proof:
Follows from theorem 3 and the definitions.$\Box$

Theorem 5 *For all legal class dictionary graphs* $\phi : CDMS(\phi) = MCDLS(OEGS(\phi))$.

Proof:
Follows directly from the definitions.$\Box$

3 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 vertices 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 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.

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

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

3.1 Minimizing construction edges

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 following class dictionary graph which is not in CNF:

```
Coin = <radius> Number <weight> Number.
Brick = <width> Number <length> Number <height> Number <weight> Number.
```

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:

```
Coin = <radius> Number .
Brick = <width> Number <length> Number <height> Number .
Weight_related : Coin | Brick *common* <weight> Number.
```

Now we implement the print function for Coin:

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

After the change of the weight composition, we get

```
Weight_related : Coin | Brick *common* <mass> Number <gravity> Number.
```

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 which are instantiated in f.

3.2 Minimizing alternation edges

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.

3.3 Generalization to methods

So far we have considered parts of classes to be classes themselves. Another useful way of looking at parts is as methods attached to classes. Particularly if we consider applying class dictionary minimization to class dictionaries other than those supplied by the basic abstraction algorithm. Now we can no longer assume that the classes in the class dictionary to be minimized come from the abstraction of object examples which in turn had no methods attached to them.

Therefore we must find a suitable representation for methods in a class dictionary graph. We can represent methods in a class dictionary graph in three ways:

- By their names and signatures. That is, specifying the name of the method and the class names of the argument types and return type.
- By their names, signatures and the dependencies they define. That is, we can add to the signature specification a list of class names referred to by the method.
- By their implementation. This includes signature, dependencies and code.

Let us consider the three options. Specifying implementation involves too much unnecessary detail and, unless a formal notation is invented, also dependent on some programming language syntax. Inventing a formal notation is out of the scope of this work and it is not clear whether it would shed much light to the issues we are addressing, namely, those of abstracting commonalities.

Specifying dependencies could give more insight in discerning whether or not two method specifications are the same and hence subject to abstraction. However, specifying dependencies forces us to consider issues such as what kinds of dependencies can a method validly specify, what classes are visible to a specific method, issues not immediately related to the problem of method abstraction. An interaction of our work with theories that explore the issue of specification of dependencies, such as the Law of Demeter [LHR88], is definitely important in its own right and is left for future research.

With this in mind, we represent methods by their names and signatures. We consider two types of methods, those that can be redefined in subclasses of an alternation class and those that cannot. Even at this early point we realize that, although we want to minimize class dictionaries with methods, we do not want to eliminate method edges in the process. With this constraint the concepts of recognizing commonalities and minimizing class dictionaries get a new meaning. We want to identify polymorphism as given by identical method definitions in unrelated classes. We want to group these classes in inheritance hierarchies that capture the polymorphic method as an abstract concept. We are actively pursuing the formalization of these ideas at the present time.

4 Related work

The fast algorithm which we present here has not been reported in the literature. 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 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 modelling. 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.

What is unique of Casais' model and algorithms is the capability to model undesirable class hierarchies and to optimise them while minimizing the changes needed.

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 paper 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 tranformation 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" normalised class hierarchy. The filter would be constructed as an expert system allowing users to input rules and constraints which might, for example, specify the priorty 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].

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.

The axiomatic Demeter kernel 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 can be adapted to abstract feature structures from examples (this was actually the starting point of our investigation).

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

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 11 *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, converting to CNF we learn 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 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.

Wegner describes informally the idea of a class dictionary transformation in his subsection 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 in this paper a mathematical treatment of a calculus of class transformations which was also informally described in [Lie88].

5 Conclusions

We have studied how to optimize class definitions and how to learn them from object examples. There are several applications of our algorithms: 1. They serve as tools for the object-oriented designer. 2. They serve as tools for the software engineer who moves from record structures to class structures with inheritance. 3. They lead to a better understanding of the object-oriented design process.

The algorithms have been developed as a part of our ongoing work on the Demeter CASE tool [Lie88], [LR89].

Acknowledgements: We would like to thank the anonymous reviewers, Kenneth Baclawski, Mitchell Wand, and Keith Wright for their contributions. Many thanks to the other members of the current Demeter team, including Ian Holland, Walter Hürsch, Christos Stamelos, Paul Steckler and Cun Xiao for their encouragement and support with implementations, paper feedback and reorganization proposals. Cun Xiao was influential in simplifying and refining the axioms for class dictionary graphs and object graphs.

6 Appendix

6.1 Mathematical objects

begindefinition A **mathematical object** is a tuple

$$(S; R_1, ..., R_k; f_1, ..., f_l)$$

where S is a finite nonempty set and $R_i, i = 1, ..., k$, are relations on S, i.e., for some natural number m_i,

$$R_i : S^{m_i} \to \{true, \quad false\}$$

and $f_i, i = 1, ..., l$ are functions defined on direct products of S, i.e., for some natural numbers m_{di}, m_{ri}

$$f_i : S^{m_{di}} \to S^{m_{ri}}.$$

(R_i is called an m_i-ary relation.)

enddefinition

begindefinition A **mathematical object** is **legal** if it satisfies all axioms of some axiom system Σ. The axioms in Σ are formulas in some logical language. (Legal objects are also called **models** of the axiom system.) enddefinition

Sometimes we partition the set S into several subsets which we name explicitly in the object definition. For further information about structures see, e.g., [BBFS74, 508] or [JLSS90, 325].

```
Cover : C1 | C2 | ... | Cm.
C1 : some alternatives from {A1 ... An}
C2 : some alternatives from {A1 ... An}
....
Cm = some alternatives from {A1 ... An}
A1 = .
...
An = .
D = <c1> C1 <c2> C2 ... <cm> Cm <cover> Cover.
```

Figure 18: Reduction

6.2 Proofs

We prove several complexity results:

Lemma 2 **CA-P** *and* **CDM-P** *are polynomially related.*

Proof:
We know from theorem 3 that for all legal class dictionaries ϕ: $CAS(CDCNFS(\phi)) = CDMS(\phi)$. Since **CDCNF-P** can be solved efficiently, the result follows.□

Lemma 3 **CDM-P** *and* **MCDL-P** *are polynomially related.*

Proof:
We know that for all legal class dictionaries ϕ: $CDMS(\phi) = MCDLS(OEGS(\phi))$. **OEG-P** can be solved in polynomial time. Since an object example which is not an output of **OEG-P** can be transformed into an equivalent object example using the basic abstraction algorithm and an algorithm for **OEG-P**, the result follows.□

Theorem 6 **CA-P (CONSOLIDATION OF ALTERNATIVES)** *is NP-hard.*

Proof:
Suppose we have a routine to solve **CA-P**,i.e., we use a Turing reduction. We can then solve the **MINIMUM COVER** problem as follows.

Let $S = \{A_1, A_2, \ldots A_n\}, C = \{C_1, C_2, \ldots C_m\}$ be an instance of minimum cover, where $C_i = \{A_{i_1}, A_{i_2}, ..., A_{i_{m_i}}\}$ is a subset of S. Build a class dictionary graph $G(VC, VA, \Lambda; EC, EA)$ as follows (see Fig. 18):

- VA has $m+1$ vertices, labeled $Cover$, $C_1, C_2, \ldots C_m$. Each vertex in VA is called a subset vertex.
- VC has n vertices, labeled $A_1, \ldots A_n$ and D
- EC contains the construction edges (D, C_i, c_i) for all i $(1 \leq i \leq m)$.
- EA has and edge (C_i, A_j), where $C_i \in VA$ and $A_j \in VC$, if subset C_i contains element A_j in the **MINIMUM COVER** instance. EA also has an edge $(Cover, A_k)$, for $k = 1, \ldots n$.

Let $G' = (VA', VC, EC, EA')$ be a solution to **CA-P** with instance G. We claim that we can get back a solution for **MINIMUM COVER** from G'. There are two possibilities:

1. VA' does not have any new vertices. Then the successors of $Cover$ are subset vertices which form a minimum cover. This is because $|EA'|$ is minimal.
2. VA' has new vertices. Then the successors of $Cover$ may include some of the new vertices. But the set of vertices reachable from any of these new vertices is a subset of the set of vertices reachable from some $C_i \in VA$. To get back a solution for **MINIMUM COVER**, we inspect all the immediate successors of $Cover$. If an immediate successor of $Cover$ is a subset vertex, we include the corresponding subset in the minimum cover. If an immediate successor of $Cover$ is a new vertex h, then we consider some subset vertex C_x which is a predecessor of h. We then include the subset corresponding to C_x in the minimum cover.

□

Example 12 *We give examples of each of the two possibilities.*

1. **MINIMUM COVER** *instance:* $S = \{X_1, X_2, X_3\}$, $C_1 = \{X_1\}$, $C_2 = \{X_2\}$, $C_3 = \{X_3\}$, $C_4 = \{X_1, X_2\}$.

   Corresponding **CA-P** *instance:*

```
Cover : X1 | X2 | X3.
C1    : X1.
C2    :      X2.
C3    :           X3.
C4    : X1 | X2.
D = <c1> C1 <c2> C2 <c3> C3 <c4> C4 <cover> Cover.
```

   After minimization:

```
Cover : C3 | C4.
C1    : X1.
C2    :      X2.
C3    :           X3.
C4    : X1 | X2.
D = <c1> C1 <c2> C2 <c3> C3 <c4> C4 <cover> Cover.
```

Notice that the immediate successors of $Cover$ are the solution to **MINIMUM COVER**.

2. **MINIMUM COVER** *instance:* $S = \{X_1, X_2, X_3, X_4\}$, $C_1 = \{X_1, X_2, X_3\}$, $C_2 = \{X_1, X_2, X_4\}$, $C_3 = \{X_3, X_4\}$.

   Corresponding **CA-P** *instance:*

```
Cover : X1 | X2 | X3 | X4.
C1    : X1 | X2 | X3.
C2    : X1 | X2 |      X4.
C3    :           X3 | X4.
D = <c1> C1 <c2> C2 <c3> C3 <cover> Cover.
```

   After minimization:

```
Cover : h | C3.
C1    : h |      X3.
C2    : h |           X4.
C3    :          X3 | X4.
H     : X1 | X2.
D = <c1> C1 <c2> C2 <c3> C3 <cover> Cover.
```

   Notice that H is a new vertex and a successor of $Cover$. Since C_1 is a predecessor of H and a subset vertex, we select it for the solution of **MINIMUM COVER**. *C_3 is also in the solution we select for* **MINIMUM COVER**. *Notice that we could have chosen C_2 instead of C_1.*

Corollary 1 **CDM-P** *and* **MCDL-P** *are NP-hard.*

Proof:
CDM-P is NP-hard since CA-P is NP-hard and CDM-P and CA-P are polynomially related. MCDL-P is NP-hard since CDM-P is NP-hard and MCDL-P and CDM-P are polynomially related. □

References

[ABW+89] M. Atkinson, F. Bancilhorn, D. De Witt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proceedings of International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, 1989.

[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.

[BBFS74] H. Behnke, F. Bachmann, K. Fladt, and W. Süss, editors. *Fundamentals of Mathematics*. The MIT Press, 1974.

[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.

[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.

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

[Cox86] Brad J. Cox. *Object-Oriented Programming, An evolutionary approach*. Addison Wesley, 1986.

[CY90] Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Yourdon Press, 1990. second edition.

[Dit90] Klaus R. Dittrich. Object-oriented database systems: The next miles of the marathon. *Information Systems*, 15(1):161–167, 1990.

[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.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability*. Freeman, 1979.

[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.

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

[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.

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

[JLSS90] Gerhard Jäger, Hans Läuchli, Bruno Scarpellini, and Volker Strassen, editors. *Ernst Specker Selecta*. Birkhäuser Verlag, Basel, Boston, Berlin, 1990.

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

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

[LBSL90a] Karl Lieberherr, Paul Bergstein, and Ignacio Silva-Lepe. Optimal and efficient abstraction of single inheritance hierarchies. In *Symposium on Object-Oriented Languages and Systems*, pages 1–24, Twente University (Inter-Actief and Computer Science), 1990. Inter-Actief, P.O. Box 217, 7500 Enschede, Netherlands.

[LBSL90b] 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.

[LBSL90c] Karl J. Lieberherr, Paul Bergstein, and Ignacio Silva-Lepe. From objects to classes: Algorithms for object-oriented design. Technical Report Demeter-3, Northeastern University, January 1990.

[LBSL90d] Karl J. Lieberherr, Paul Bergstein, and Ignacio Silva-Lepe. Optimal and efficient abstraction of classes from objects. Technical Report NU-CCS-90-6, revised Jan. 91, Northeastern University, January 1990.

[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.

[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.

[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.

[LR89] Karl J. Lieberherr and Arthur J. Riel. Contributions to teaching object-oriented design and programming. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 11–22, October 1989.

[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.

[LW89] Karl Lieberherr and Carl Woolf. Grammar-based planning for object-oriented applications. Technical Report NU-CCS-89-11, Northeastern University, Feb. 1989.

[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.

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

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

[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.

[Ste89] David Steier. Automating algorithm design within a general architecture for intelligence. Technical Report CS-89-128, Carnegie Mellon University, April 1989.

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

[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.

[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.

Contents