

Formal Foundations for Object-Oriented Data Modeling

Karl Lieberherr and Cun Xiao
Northeastern University, College of Computer Science
Cullinane Hall, 360 Huntington Ave., Boston MA 02115
lieber@corwin.CCS.northeastern.EDU
cunxiao@corwin.CCS.northeastern.EDU
phone: (617) 437 20 77, fax: (617) 437 51 21

January 26, 1993

Texed at 14:37 January 26, 1993
Copyright ©1991 Karl Lieberherr

Abstract

We give an axiomatic definition of the basic structure, called a class dictionary graph, used by object-oriented designers and programmers during the software development process. The contributions of this paper are twofold: An axiomatic foundation for object-oriented data modeling and efficient algorithms for checking whether a given data model satisfies the axioms.

The presented data model is the foundation of a number of techniques for developing object-oriented systems including adaptive software, schema abstraction from object examples, schema optimization, planning techniques for system development, the Law of Demeter, etc.

Keywords: Object-oriented data modeling and programming, algorithms and tools for object-oriented design, axioms for data models, Demeter Method.

1 Introduction

In object-oriented data modeling and programming, each entity of the problem domain is represented by a set of objects with relations and operations. Each object is composed of part objects/subobjects which express relations between objects. Operations, called methods or

member functions, may call operations of other objects or change the state of some objects. At the object level in Fig. 1, the object called John has two part objects which are his mother and father, called Mary and Joe. These two parts represent the mother and father relations.

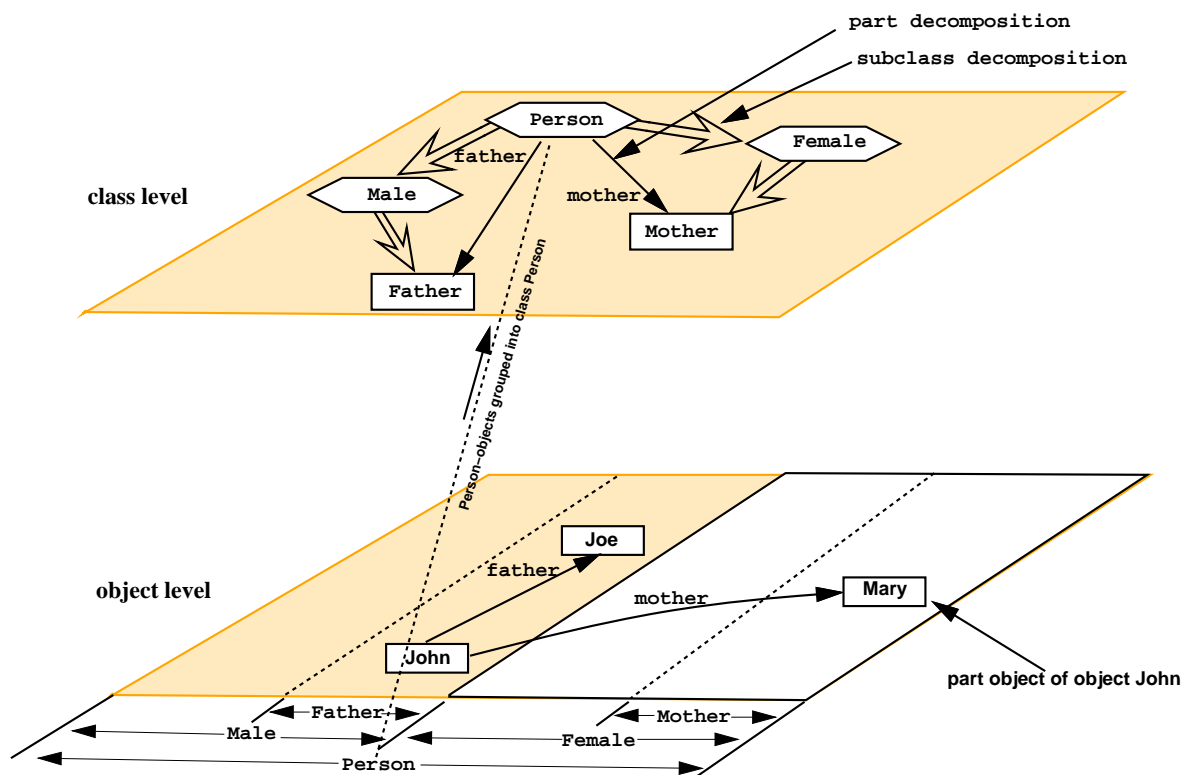


Figure 1: Classes and objects

In many programming languages (e.g., [9], [31], [34]) which support object-oriented programming, objects are grouped into classes and the operations are attached to classes. To express the structure of objects, we have two kinds of class decompositions: 1. Each class of a system may be decomposed into subclasses. 2. Each class may be refined into part classes on a lower level of partitioning. **Person**-objects are grouped into class **Person**. A **Person**-object can be a **Male**-object or a **Female**-object. A **Male**-object can be a **Father**-object or does not have any child (not shown in Fig. 1). And similarly for a **Female**-object.

The class **Person** is called an abstract class or the super class of classes **Male**, **Female**, **Father** and **Mother**. Class **Father** is a subclass of **Person** and **Male**. Class **Father** is also a concrete class. Abstract classes are uninstantiabile and have subclasses. Concrete classes are instantiabile and have no subclasses.

The part class decomposition describes the part-of relations with other objects. A **Person**-object must have two relations, a mother relation and a father relation, with two other objects. Subclass decomposition describes inheritance relations between classes. Class **Male** will inherit all the part-of relations class **Person** owns. In other words, A **Male**-object must have the father and mother relations.

The decomposition of classes is naturally represented as a graph, called a *class dictionary graph*. We use the graphical notation based on [36] to draw class dictionary graphs.

Classes become vertices and decomposition relationships are represented by edges in the graph (see Fig. 1). There are two kinds of edges used in the graph: edges for the subclass decomposition and edges for the part decomposition. Subclass decomposition edges are called *alternation edges* (drawn as \implies) and part decomposition edges are called *construction edges* (drawn as \longrightarrow). Construction edges are part-of relationships; alternation edges are kind-of/is-a relationships.

There are two kinds of vertices in the graph: the vertices for instantiable classes and the vertices for uninstantiable classes. The vertices for instantiable classes are called *construction vertices* (drawn as \square), like **Father** and **Mother**. The classes they represent are also called *concrete classes*. The vertices for uninstantiable classes are called *alternation vertices* (drawn as \diamond), like **Person**, **Male** and **Female**. The classes they represent are also called *abstract classes*.

During a programming process, alternation classes serve to define interfaces (i.e., they serve the role of types) and construction classes serve to provide implementations for the interfaces. In standard object-oriented terminology we describe here the accepted programming rule: “Inherit only from abstract classes” [15]. This rule can be exploited to derive an analogy between class dictionary graphs and grammars.

Not all class dictionary graphs are meaningful. For example, every person has a mother and a father and the father and mother also have their own fathers and mothers. This leads to objects of infinite size, unless the objects are circular. But circular objects are not meaningful in this case. Circular objects, often used in practice, are meaningful in other cases. So we don’t want to exclude them. However, we want to avoid the situation that *all* finite objects of some class must be circular (we will give an example in Section 3). Therefore we give an axiomatic characterization of the meaningful class decompositions.

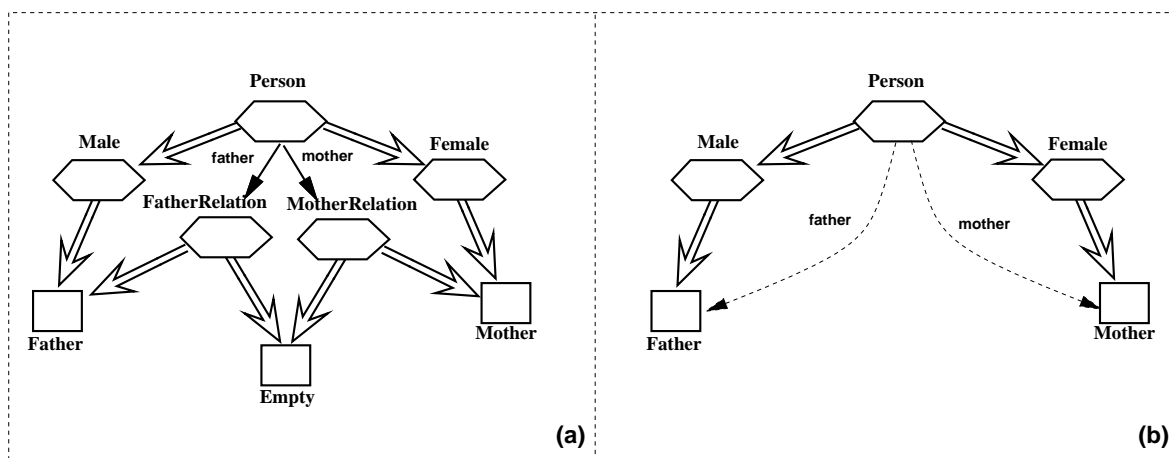


Figure 2: The relaxed graph

Part-of relations between objects may be optional. Suppose the relations between one person and his or her father and mother are optional. In Fig. 2(a), we expressed the optional relations

by adding two alternation classes, which are `MotherRelation` and `FatherRelation`, and one construction class `Empty`. If the part-object for the father relation is an `Empty`-object, that means we do not specify who is the father. In this case, we can have finite objects. In practice, we use a third kind of part decompositions, called *optional construction edges* (drawn as \dashrightarrow), to express optional part-of relations. Fig. 2(b) gives an equivalent class dictionary graph with two optional construction edges.

Part-of relations between objects may be repeated. Suppose we want to design a class dictionary graph for enrollments in a college. The requirements are that every student must take at least one course, and that certain courses may not be taken by students. We give the class dictionary graph in Fig. 3. Each instance of vertex `AtLeastOneCourse` is a list of one or more `Course`-objects. Construction edge `AtLeastOneCourse` \xrightarrow{first} `Course` represents the first course in a list. Construction edge `AtLeastOneCourse` \xrightarrow{second} `Courses` represents the rest of courses in a list. Construction edge `Student` \xrightarrow{takes} `AtLeastOneCourse` defines that each student has to select at least one course.

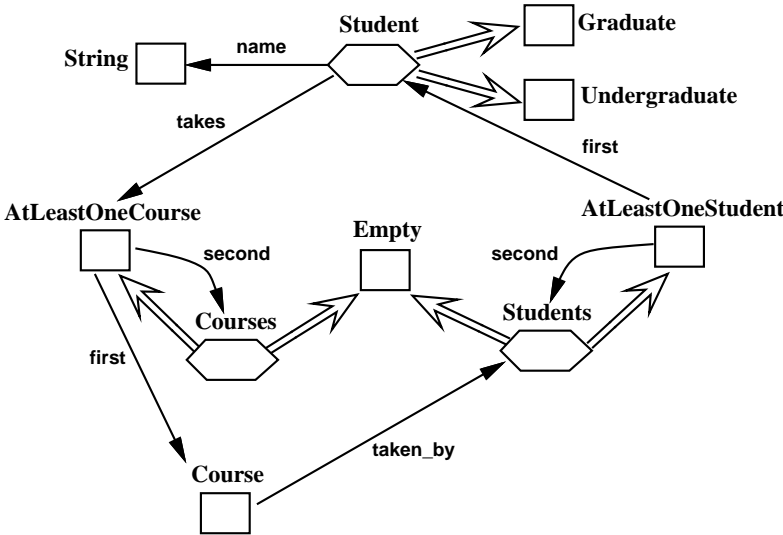


Figure 3: Students and Courses

Fig. 3 is not an elegant way to express repeated part-of relations, such as the repeated part-of relation between `AtLeastOneCourse` and `Course`. In practice, we use repetition vertices (drawn as \diamond) to represent them.

Fig. 4 gives an equivalent class dictionary graph by using repetition classes. The repetition edge, which is drawn as \dashrightarrow , represents zero or more repeated part-of relations. The repetition edge, which is drawn as \rightarrow , represents one or more repeated part-of relations.

To make the analysis simpler, in this paper we only consider construction vertices, alternation vertices, construction edges and alternation edges. And when we talk about construction edges, we really mean that they are required. When a construction edge outgoing from a class is required, its corresponding part-object has to be present in every object of the class. Required

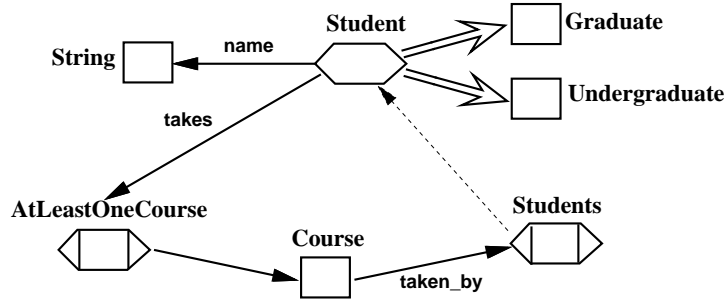


Figure 4: Equivalent class dictionary graph with repetition vertices

construction edges eliminate conditional statements, since we know that those parts always exist.

There has been considerable debate about the definition of object-oriented systems and there is a lack of agreement on a formal foundation. [1, 8] describe characteristics which such a formal model should possess. We view our formal model, which we call the Demeter kernel model, as 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 [20]. We call our model the Demeter kernel model since we developed it during our work with the Demeter System (starting in 1984, [20]) and since it only describes the structural parts of useful object-oriented data models. Class dictionary graphs are a useful design abstraction which can be debugged independently. Class dictionary graphs are complemented by propagation patterns [19, 17] to define operations in succinct form. A propagation pattern describes a collection of algorithms from which we can select one by giving a class dictionary graph.

This research is part of a long-range research project started at GTE Laboratories in 1984 and continued at Northeastern since 1985. The pattern of our research is to identify abstractions in the object-oriented software development process, to formalize the abstractions and to provide tools to take advantage of the abstractions [20], [26], [23], [22], [10], [25], [24], [27], [21], [14].

Dozens of students have been involved in the research and in tool development. Hundreds of users in the U.S. and Europe have used our tools and have learned the theory on which the tools are based. The tools are part of the Demeter System which consists of the Method and supporting Tools. The Method describes a mechanism for object-oriented software development based on the class dictionary graph abstraction. It is this abstraction which we describe in this paper and we show how to enforce it algorithmically. The Demeter Tools generate or provide generically software for working with the objects defined by a class dictionary, a generalization of a class dictionary graph. The available object operations include: constructing, destructing, parsing, printing, drawing, traversing, updating, comparing, copying etc. Class dictionary graphs are also useful without the tools as a high-level object-oriented design notation.

The rest of the paper is organized as follows: In section 2 we formally define structures for classes and objects graphs. We introduce the Demeter kernel model restricted by two axioms, the

Cycle-Free Alternation Axiom and the *Unique Label Axiom*. In section 3 we formally introduce the *Inductiveness Axiom*. We give an example of programming with a class dictionary graph in section 4. The first two axioms are easily checked, but for the Inductiveness Axiom we present simple, efficient checking algorithms. In section 5, we first simplify the axiom checking problem to flat class dictionary graphs and in section 6 we present algorithms for checking the axiom for flat class dictionary graphs. The paper concludes with section 7 which discusses related work.

2 Structures for object-oriented design

Part-of relations and *kind-of/is-a* relations are used to describe relationships between classes. In [22, 4], *class dictionary graphs* are introduced based on these two kinds of relations, and kind-of relations are expressed by alternation edges. Later we found that kind-of relations are not enough for describing the properties of subtyping and inheritance, in terms of defining objects and describing programs. We split a kind-of relation into an alternation and inheritance relation. This results in a generalized structure, called *semi-class dictionary graph*, which is used as abstraction for defining behavior and for defining object sets.

We will formally introduce *semi-class dictionary graphs*, *partial class dictionary graphs*, *class dictionary graphs* and *object graphs*. Semi-class dictionary graphs serve as generalized structures for class dictionary graphs and partial class dictionary graphs. Partial class dictionary graphs and class dictionary graphs are used for defining object sets. Partial class dictionary graphs are also used to check the inductiveness of class dictionary graphs, which will be discussed in this paper.

A class dictionary graph is an abstraction of the class definitions in some object-oriented language rather than an ad hoc programming language for the following reasons:

- A class dictionary graph focuses only on is-a and part-of relationships between classes. This level of abstraction is useful for several tasks, e.g., debugging of class structures and planning an implementation or querying the objects defined by the class dictionary graph. Class dictionary graphs also serve as abstractions for application-specific class libraries which only contain generic functionality for manipulating objects, such as copying, printing, comparing, accessing, constructing etc.
- Class dictionary graphs serve as customizers for adaptive programs [19, 17].
- A class dictionary graph contains sufficient information to define legal objects. Each part of an object has a type. (This also holds for the class definitions of strongly typed languages, such as C++. However, C++ does not enforce at compile-time that all objects must be legal in our sense.)
- Class dictionary graphs are concise and better structured. There are several kinds of class definitions used for structuring the classes.

- A class dictionary graph is at a higher level of abstraction than class definitions in many object-oriented languages. Recursive class definitions are free of pointers and the class definitions are given in a programming language independent notation.
- Class dictionary graphs may contain parameterized class definitions[26]. Parameterized class definitions are not allowed in many object-oriented languages.
- A class dictionary graph, if extended with concrete syntax, also defines a language for describing the objects[27].

In data base terminology, object graphs are instances of classes defined in an object base schema, the class dictionary graph. 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 [36]. More recently, graphs have been used to model object-oriented data bases in [16, 12, 3].

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., [29]). The call relationships between classes describe important design information, e.g., for checking the Law of Demeter [25].

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 is-a relationships. For more motivation from the “adaptive” software point of view, see [17].

2.1 Structures for defining classes

We have introduced three kinds of vertices, which are construction, alternation and repetition vertices. We also introduced six kinds of edges, which are required/optional construction, alternation, inheritance and zero-more/one-more repetition edges. Since repetition vertices and optional construction edges can be represented by alternation and construction concepts, we only deal with construction, alternation vertices, (required) construction edges, alternation edges and inheritance in the rest of the paper.

Vertices and edges are used in many different ways. A vertex may represent a class or a set of objects. An edge may represent an instance variable, a function call or a selection of subobjects. It will be obvious from the context which interpretation we refer to.

Following [17], we introduce a sequence of graph structures which are important for developing adaptive software with propagation pattern. The first graph structure, called semi-class dictionary graph, is important for defining traversal algorithms for objects. A second application of

semi-class dictionary graphs is to specialize them to partial class dictionary graphs which are important for defining sets of objects. For the purpose of this paper, we could have defined partial class dictionary graphs directly, but since semi-class dictionary graphs are fundamental to adaptive software, the detour is well justified.

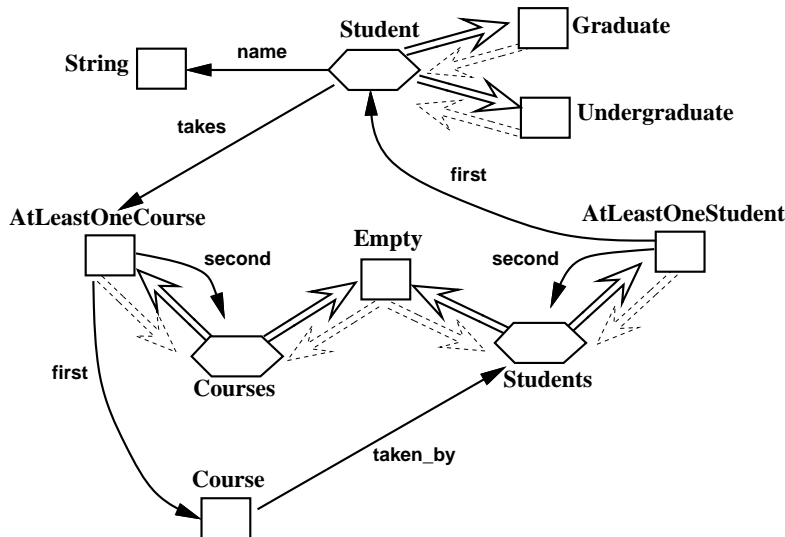


Figure 5: A semi-class dictionary graph

Fig. 5 shows a semi-class dictionary graph. Edges drawn as \dashrightarrow are *inheritance edges*. Undergraduate and Graduate are subclasses of Student, since there are alternation edges $\text{Student} \Rightarrow \text{Undergraduate}$ and $\text{Student} \Rightarrow \text{Graduate}$. Therefore Undergraduate-objects and Graduate-objects are Student-objects. Undergraduate and Graduate inherit from Student, since there are inheritance edges $\text{Undergraduate} \dashrightarrow \text{Student}$ and $\text{Graduate} \dashrightarrow \text{Student}$. Therefore Undergraduate and Graduate inherit the part-of relationship with String from Student. Formally,

Definition 1 A *semi-class dictionary graph* is a tuple $\Phi = (VC, VA, \Lambda; EC, EA, EI)$ with finitely many labeled vertices in the disjoint sets VC and VA . We define $V = VC \cup VA$. VC is a set of vertices called construction vertices. VA is a set of vertices called alternation vertices. EC is a ternary relation on $V \times V \times \Lambda$, called construction edges. Λ is a finite set of construction edge labels. EA is a binary relation on $VA \times V$, called alternation edges. EI is a binary relation on $V \times VA$, called inheritance edges. ■

We also use $V(\Phi)$ to represent all the vertices in Φ .

There are three path concepts for semi-class dictionary graphs: *alternation paths*, *inheritance paths* and *containment paths*.

Definition 2 In a semi-class dictionary graph $\Phi = (VC, VA, \Lambda; EC, EA, EI)$:

- An *alternation path* is a path satisfying regular expression EA^* , i.e. containing only alternation edges.

Vertex w is *alternation-reachable* from vertex v if there is an alternation path from v to w or $v = w$.

We write $v \xRightarrow{*} w$ if w is alternation-reachable from v . We write $v \xRightarrow{+} w$ if w is alternation-reachable from v via more than one alternation edge.

- An *inheritance path* is a path satisfying regular expression EI^* , i.e. containing only inheritance edges.

Vertex w is *inherits* from vertex v if there is an inheritance path from v to w or $v = w$.

We write $v \xRightarrow{*} w$ if v inherits from w . We write $v \xRightarrow{+} w$ if v inherits from w via more than one inheritance edge.

- A *containment path* is a path satisfying regular expression $(EA \mid (EI)^*EC)^*$, i.e., a containment path is a sequence of construction, alternation and inheritance edges such that an inheritance edge can only be followed by a construction edge or an inheritance edge and the path cannot end with an inheritance edge.

■

The *length* of path p is the number of edges in p . We say that from vertex v back to vertex v there is always an alternation path of length zero, an inheritance path of length zero and a containment path of length zero. A *cycle* in a semi-class dictionary graph is a path of length more than 0 from some vertex v back to v .

If there is an alternation path from vertex v to vertex w , then vertex w is a subclass of vertex v and an object of w is also an object of v . If there is an inheritance path from vertex v to vertex w , then vertex v inherits from vertex w . The relation *inherits* is the reflexive transitive closure of the relation EI . The relation *alternation-reachable* is the reflexive transitive closure of the relation EA .

Further explanation is needed for containment paths. The motivation behind the containment path concept is to define the set of classes which are needed to build objects of a given class. The set of vertices which are reachable (by a containment path) from a given vertex v , defines the set of classes whose instances may appear as (nested) part-objects of v -objects. A v -object is an object of class v .

Consider containment path $\text{Course} \xrightarrow{\text{taken_by}} \text{Students} \implies \text{AtLeastOneStudent} \xrightarrow{\text{first}} \text{Student} \implies \text{Graduate} \xRightarrow{\text{name}} \text{String}$. Following the edges on the containment path, a `Students`-object is a part-object of a `Course`-object because of construction edge $\text{Course} \xrightarrow{\text{taken_by}} \text{Students}$. An `AtLeastOneStudent`-object is a part-of object of the `Course`-object because of alternation edge $\text{Students} \implies \text{AtLeastOneStudent}$. The `AtLeastOneStudent`-object represents one or more students who take the course. A `Student`-object is a nested part-object of the `Course`-object, which is the first student in the list who takes the course. The `Student`-object actually is a `Graduate`-object because of alternation edge $\text{Student} \implies \text{Graduate}$. From `Student`,

Graduate inherits part-of relationship name with String, i.e., a Graduate-object has a part-object which is a String-object.

Path $\text{Graduate} \dashrightarrow \text{Student} \xrightarrow{\text{name}} \text{String}$ is also a containment path, since it satisfies regular expression $(EA \mid (EI)^*EC)^*$. This containment path tells that a Graduate-object has a String-object as its immediate part-object through the inheritance relationship from Graduate to Student.

But path $\text{Graduate} \dashrightarrow \text{Student} \Rightarrow \text{Undergraduate}$ is not a containment path. Graduate represents a set of Graduate-objects which is a subset of the set of Student-objects. Similarly, Undergraduate represents a set of Undergraduate-objects which is a subset of the set of Student-objects. The set of Graduate-objects and the set of Undergraduate-objects are disjoint. Therefore it does not make sense if we start from Graduate and follow the inheritance edge and choose alternation edge $\text{Student} \Rightarrow \text{Undergraduate}$. By choosing alternation edge $\text{Student} \Rightarrow \text{Undergraduate}$, we mean to choose the set of Undergraduate-objects. Regular expression $(EA \mid (EI)^*EC)^*$ excludes such a pattern.

Alternation paths are a special kind of containment paths. Inheritance paths are not. When we refer to a path p , we mean a containment path, unless we explicitly mention that p is an inheritance or alternation path.

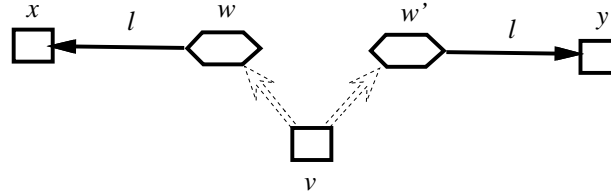


Figure 6: Forbidden graph

The semi-class dictionary graphs which appear in applications should always satisfy two independent axioms. We say a semi-class dictionary graph is *legal* if it satisfies two axioms called the *Cycle-Free Alternation and Inheritance Axiom* and the *Unique Label Axiom*.

Definition 3 A *legal* semi-class dictionary graph $\Phi = (VC, VA, \Lambda; EC, EA, EI)$ satisfies $(V = VC \cap VA)$:

Cycle-Free Alternation and Inheritance Axiom

There are no alternation cycles and no inheritance cycles, i.e., $\nexists v \in V : v \xrightarrow{+} v$ or $v \dashrightarrow^+ v$;

Unique Label Axiom (see Fig. 6)

$\forall v, w, w', x, y \in V, l \in \Lambda$: if $v \dashrightarrow^* w$ or $w \xrightarrow{*} v$, and $v \dashrightarrow^* w'$ or $w' \xrightarrow{*} v$, then

$w \xrightarrow{l} x, w' \xrightarrow{l} y \in EC$ implies $w \xrightarrow{l} x = w' \xrightarrow{l} y$ (i.e., $w = w'$ and $x = y$).

■

The following two motivations for the axioms are from [17].

The Cycle-Free Alternation Axiom is natural and has been proposed by other researchers, e.g., [32, page 396], [33, page 109: Class names may not depend on themselves in a circular fashion involving only (alternation) class productions]. The axiom implies that a class may not inherit from itself via at least one inheritance edge.

The Unique Label Axiom guarantees that “inherited” construction edges are uniquely labeled and excludes class dictionary graphs which contain the pattern shown in Fig. 6. Other mechanisms for uniquely naming the construction edges could be used, e.g., the renaming mechanism of Eiffel and the overriding of part classes [31].

From now on, when we refer to a semi-class dictionary graph, we mean a legal semi-class dictionary graph.

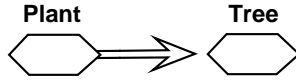


Figure 7: A semi-class dictionary graph which cannot define objects

Some semi-class dictionary graphs are not meaningful for defining objects. The semi-class dictionary graph in Fig. 7 is an example. There are two alternation vertices in this example, but there is no construction vertex. Therefore this semi-class dictionary graph cannot define objects, because alternation vertices are uninstantiable. Furthermore, the two vertices are in an alternation relationship, but not in an inheritance relationship. This is not meaningful to define objects, since for object definition, alternation edge plays a double role: alternation and inheritance. Partial class dictionary graphs are a minimal structure sufficient for defining objects.

Definition 4 A *partial class dictionary graph* $P = (VC_P, VA_P, \Lambda_P; EC_P, EA_P, EI_P)$ anchored at vertex v_0 is a semi-class dictionary graph with the following four properties ($V_P = VC_P \cup VA_P$):

1. $v_0 \in V_P$ and $\forall w \in V_P \exists v \in V_P : v$ is reachable from v_0 via a containment path and w is reachable from v via an inheritance path.
2. $\forall v \Rightarrow w \in EA_P : w \dashrightarrow v \in EI_P$.
In other words, alternation edges imply inheritance edges.
3. $\forall v \in VA_P \exists w \in V_P : w \dashrightarrow v \in EI_P$.
In other words, there is no alternation vertex without an incoming inheritance edge.
4. $\forall v \in VA_P : v = v_0$ or $\exists w \Rightarrow v \in EA_P$ or $\exists w \xrightarrow{l} v \in EC_P$ implies $\exists v' \in V_P$ s.t. $v \Rightarrow v' \in EA_P$.
In other words, if an alternation vertex is “used” (i.e., it is v_0 or has some incoming construction or alternation edge in P), then this vertex must have at least one outgoing alternation edge in P .

The vertices incident with the edges are also in P . ■

The semi-class dictionary graph in Fig. 8 is a partial class dictionary graph.

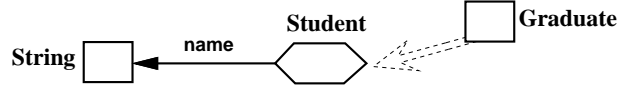


Figure 8: A partial class dictionary graph

The intuition of the first property is that we want to use P to define only objects which are instances of vertex v_0 . The reason for the second property is that the inheritance ancestors have the common structures and behaviors of their inheritance descendants. The reason for the third property is that every alternation vertex must play a role in defining objects. The reason for the fourth property is that an alternation vertex, corresponding to an uninstantiable class, has to have at least one immediate alternation descendant if vertex v is used as construction or alternation descendant or if v is the vertex we want to build objects of. We use “anchored” instead of “rooted”, because usually a partial class dictionary graph does not have tree structure.

The partial class dictionary graph in Fig. 8 cannot define objects for **Student**, since **Student** represents an abstract class and there is not alternation edge outgoing from **Student**. Class dictionary graphs are a specialization of partial class dictionary graphs. A *class dictionary graph* defines objects for each vertex in the class dictionary graph. Formally,

Definition 5 A *class dictionary graph* $\Phi = (VC, VA, \Lambda; EC, EA, EI)$ is a union of partial class dictionary graphs such that

$$v \implies w \in EA \text{ iff } w \dashrightarrow v \in EI.$$

■

This definition is equivalent to the class dictionary graph definition given in earlier papers [22, 4, etc.]. Those papers also contain further motivations. The semi-class dictionary graph in Fig. 5 is a class dictionary graph.

Since in class dictionary graphs the inheritance edge, say $v \dashrightarrow w$, occurs whenever the alternation edge $w \implies v$ occurs and vice versa, we usually do not draw inheritance edges in class dictionary graphs as shown in Fig 3.

A class dictionary graph Φ defines objects for each vertex in Φ . Sometimes when we want to analyze or test a system [24, 18], we need to find all the required vertices and associated edges to build objects for a certain vertex. Partial class dictionary graphs of a given class dictionary graph provide such functionality. Informally, a partial class dictionary graph anchored at class C , contains enough classes to build some C -object.

Definition 6 For a class dictionary graph $\Phi = (VC_\Phi, VA_\Phi, \Lambda_\Phi; EC_\Phi, EA_\Phi, EI_\Phi)$, a *partial class dictionary graph* $P = (VC_P, VA_P, \Lambda_P; EC_P, EA_P, EI_P)$ of Φ anchored at vertex v_0 is a partial class dictionary graph which has the following four properties:

1. $VC_P \subseteq VC_\Phi, VA_P \subseteq VA_\Phi, EC_P \subseteq EC_\Phi, EA_P \subseteq EA_\Phi, EI_P \subseteq EI_\Phi$ and $\Lambda_P \subseteq \Lambda_\Phi$
2. $\forall v \in VC_P \cup VA_P \forall v \dashrightarrow w \in EI_\Phi : v \dashrightarrow w \in EI_P$.
In other words, if a (construction or alternation) vertex v is contained in V_P then all inheritance edges outgoing from v in Φ are in P .
3. $\forall v \in VC_P \cup VA_P \forall v \xrightarrow{l} w \in EC_\Phi : v \xrightarrow{l} w \in EC_P$.
In other words, if a (construction or alternation) vertex v is contained in V_P then all construction edges outgoing from v in Φ are in P .

The vertices incident with the edges are also in P . ■

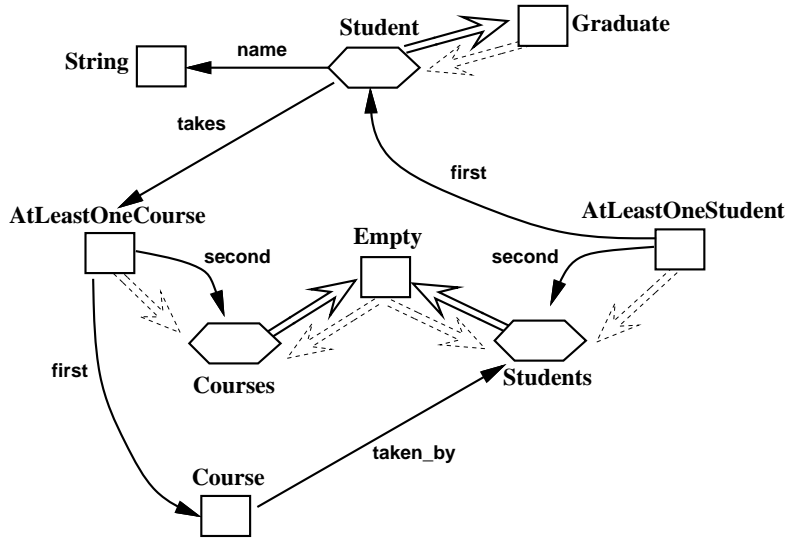


Figure 9: A partial class dictionary graph

For a given vertex v in a partial class dictionary graph P of a class dictionary graph Φ , all the superclasses of v in Φ have to be present in P according to the second property. The reason for the third property is that part-of relations between classes are required. The partial class dictionary graph in Fig. 8 is not a partial class dictionary graph of the class dictionary graph in Fig. 5, because construction edge $\text{Student} \xrightarrow{\text{takes}} \text{AtLeastOneCourse}$ is not present in the partial class dictionary graph. The partial class dictionary graph in Fig. 9 is a partial class dictionary graph of the class dictionary graph in Fig. 5 anchored at **Graduate**.

Fig. 10 shows the relations between the concepts we introduced so far, and is reproduced from [17].

2.2 Object graphs

In this section we formally define a set of objects defined by a partial class dictionary graph and therefore also by a class dictionary graph. We first define three technical concepts: *associated*,

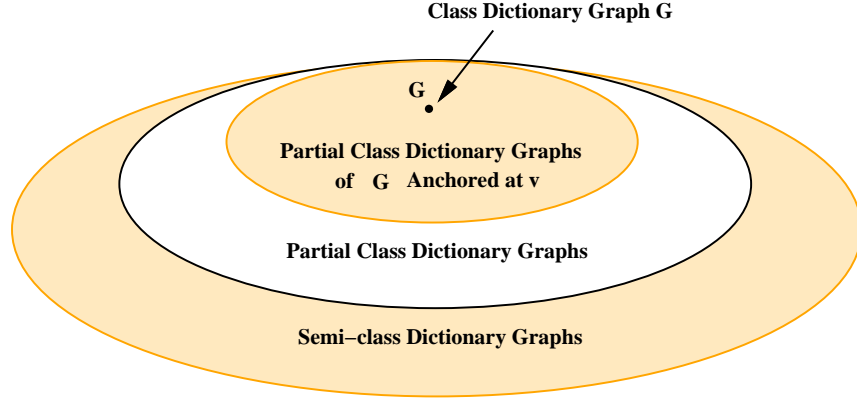


Figure 10: The relations between concepts

Parts and *PartClusters* before we define object graphs. An *associated* set of a class defines the set of instantiable subclasses of the class. *Parts* defines the set of parts of a class with their names and types. *PartClusters* is a generalization of *Parts* where the part types are given by a set of instantiable classes, using the definition of *associated*. The object graph definition is split into two parts: first we define the structure of object graphs without reference to a partial class dictionary graph and then we introduce the legality of an object-graph with respect to a partial class dictionary graph.

All the objects in this model are instantiated from construction vertices. For any vertex v in a partial class dictionary graph Φ , if we know the set S of all the construction vertices which are alternation-reachable from v , we will know all the possible objects of vertex v . The set S is called the *associated* set of vertex v .

Definition 7 Let P be the partial class dictionary graph $(VC, VA, \Lambda; EC, EA, EI)$ anchored at some vertex. The *associated* set of a vertex $v \in VC \cup VA$ is

$$\mathcal{A}(v) = \{v' \mid v' \in VC \text{ and } v \xrightarrow{*} v'\}.$$

■

Next we introduce the *Parts* function. The construction edge $v \xrightarrow{l} w$ describes a part-of relationship of vertex v with vertex w . The relation is called l . Such relationships can be inherited by inheritance descendants. It is convenient to have the pair (l, w) , called a *part*, which means the relation with vertex w , called l . Therefore each vertex has the parts of its own or inherited from its inheritance ancestors.

Consider the class dictionary graph in Fig. 9. Vertex **Graduate** has two parts, which are **(name, String)** and **(takes, AtLeastOneCourse)**. The two parts are inherited from vertex **Student**.

Definition 8 Let P be the partial class dictionary graph $(VC, VA, \Lambda; EC, EA, EI)$ anchored at some vertex. For any $v \in V$,

$$Parts(v) = \{(l, w) \mid \exists v' : v \overset{*}{\dashrightarrow} v' \text{ and } v' \xrightarrow{l} w \in EC\}.$$

■

The *PartClusters* function is a generalization of the *Parts* function. For an object of a given vertex, each immediate part-object corresponds to an object of a part class. The function *Parts* is not sufficient to define what kind of objects can be in each part. Therefore we replace w with $\mathcal{A}(w)$ in the previous definition to obtain the definition of part clusters. The result indicates the classes which may be instantiated for each part.

Definition 9 Let P be the partial class dictionary graph $(VC, VA, \Lambda; EC, EA, EI)$ anchored at some vertex. For any $v \in V$,

$$PartClusters(v) = \{(l, \mathcal{A}(w)) \mid \exists v' : v \overset{*}{\dashrightarrow} v' \text{ and } v' \xrightarrow{l} w \in EC\}.$$

■

The *PartClusters* definition uses both relationships: alternation and inheritance. Alternation is used to express a set of objects which may be in a part and inheritance is used to inherit parts. Consider the class dictionary graph in Fig. 9.

$$PartClusters(\text{AtLeastOneStudent}) = \{ (\text{first}, \{\text{Graduate}\}), \\ (\text{rest}, \{\text{Empty}\}) \}.$$

Fig. 11 shows an object of vertex **Graduate**, usually called a **Graduate-object**. The graph is called an *object graph*. Each vertex in the object graph corresponds to an instantiation of a construction vertex. Each edge is an instance of a part-of relation. We use $i1 \overset{\text{name}}{\dashrightarrow} i2$ to represent the edge from vertex $i1$ to vertex $i2$ with label **name**. In the picture, $i1$ is the object identifier of the **Graduate-object**. And similarly for $i2, i3$ and etc.

Since we simulate optional and repeated part-of relations by alternation and construction relations, all the part-objects of an object have to exist. Therefore when we talk about an "object", this "object" can be a group of objects because all its immediate and nested part-objects are forced to be included.

An *object graph* describes the structure of a group of objects mathematically. Each vertex in the object graph corresponds to an element in the group, called an instance/object of some vertex in a partial class dictionary graph.

We formulate the concept of an object graph independently of a class dictionary graph. We use a set S to represent the vertices of some class dictionary graph. In definition 12 we specify when an object graph is legal with respect to a class dictionary graph.

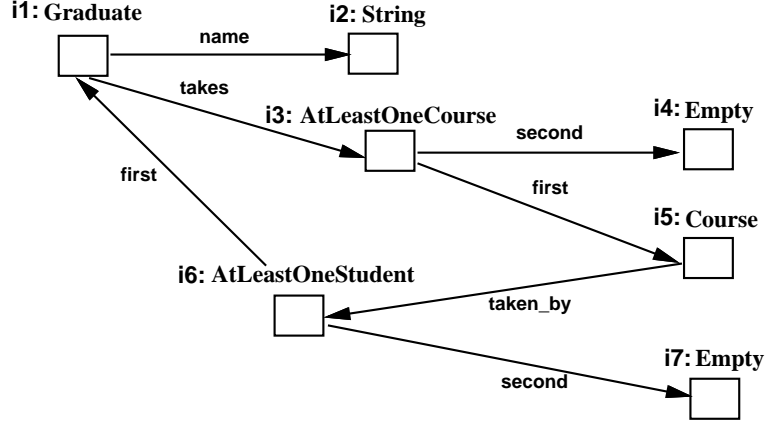


Figure 11: An object of vertex Graduate

Definition 10 An *object graph* is a graph $H = (W, S, \Lambda_H; E, \lambda)$ with vertex sets W, S and edge label set Λ_H satisfying the following properties:

1. The function $\lambda : W \rightarrow S$ maps each vertex of H to a vertex in S .
2. E is a ternary relation on $W \times W \times \Lambda_H$.
3. $\forall v, w, w' \in W \forall l \in \Lambda_H :$

$$v \xrightarrow{l} w, v \xrightarrow{l} w' \in E \text{ implies } v \xrightarrow{l} w = v \xrightarrow{l} w' \text{ (i.e., } w = w').$$

■

All the elements in W are object identifier. All the elements in S are the types of the vertices in the object graph. Condition 3 tells that there are no two edges with the same label outgoing from a vertex in an object graph.

In an object graph $H = (W, S, \Lambda_H; E, \lambda)$, a *path* P is a sequence of edges from E such that the end vertex of each edge in P is the start vertex of the next edge in P if there is one. The length of path P is the total number of edges in P .

If there is a path of length n from vertex v to vertex w , we write $v \xrightarrow{n} w$. For any vertex v , it is always true that $v \xrightarrow{0} v$. We write $v \xrightarrow{*} w$ and say that vertex w is *reachable* from vertex v when n can be zero; we write $v \xrightarrow{+} w$ when n is larger than zero.

Definition 11 In an object graph $H = (W, S, \Lambda_H; E, \lambda)$, if there exists a vertex w_0 in W such that every vertex in W is reachable from w_0 then we call object graph H *anchored* at w_0 . ■

Since only construction vertices can be instantiated, each vertex in the object graph is mapped to some *construction* vertex in the partial class dictionary. The edges between vertices

in the object graph stand for the part-of relations between the objects in the group. These relations are derived from the construction edges in the partial class dictionary graph. You can check object graphs with the *PartClusters* function of the partial class dictionary graph. For each object of a construction vertex v , *PartClusters*(v) tells you how many part-objects this object *must* have, and what kind of objects each part may contain.

Not all object graphs with respect to a partial class dictionary graph are legal. Intuitively, the object structure has to be consistent with the class definitions and all the classes in S have to be construction classes.

Definition 12 An object graph $H = (W, S, \Lambda_H; E, \lambda)$ anchored at w_0 is a *legal* v_0 -object with respect to a partial class dictionary graph P anchored at v_0 where $P = (VC, VA, \Lambda; EC, EA, EI)$, if H satisfies the following ($\exists!$ means “there exists exactly one”):

1. $S \subseteq VC$ and $\Lambda_H \subseteq \Lambda$ and
2. $\exists v \in VC$: v is alternation-reachable from v_0 and $v = \lambda(w_0)$,
3. $\forall \mu \in W \exists! v \in S$ s.t. $\lambda(\mu) = v$ and
 - (a) $\forall (\mu, \nu, l) \in E : \exists! (l, A) \in PartClusters(v)$ s.t. $\lambda(\nu) \in A$
 - (b) $\forall (l, A) \in PartClusters(v) : \exists! \nu \in W$ s.t. $\lambda(\nu) \in A$ and $(\mu, \nu, l) \in E$.

■

The definition enforces three properties: 1. All the vertices in S have to be construction vertices in P . Therefore every vertex in W must be an instance of some construction vertex in P . 2. The anchor of the object graph must be an instance of a vertex in P which is alternation reachable from v_0 . 3. If a vertex μ in W is an instance of a construction vertex v in P , then every part-object of the instance must conform to some part of vertex v , and every part of vertex v must have an instance as a part-object of instance μ . The third property is an application of the Unique Label Axiom. If this axiom is violated, we can not identify the element (l, A) from *PartClusters*(v), simply by the label l .

The object graph anchored at vertex **i1** in Fig. 12 is not a legal **Graduate**-object of the partial class dictionary graph in Fig. 9, since vertex **Graduate** has two parts. But the object graph is a legal **Graduate**-object of the partial class dictionary graph in Fig. 8.

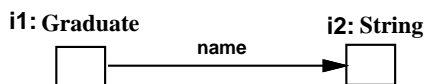


Figure 12: Illegal object

From now on, we use legal object graphs. Next we formally define all the object graphs of a class dictionary graph Φ . In database terminology, *Objects*(Φ) represents all instances of object

base schema Φ . A \mathcal{C} -object, where \mathcal{C} is a construction vertex in the class dictionary graph, is an instance of vertex \mathcal{C} . For any alternation ancestor \mathcal{A} of vertex \mathcal{C} , we also say a \mathcal{C} -object is an \mathcal{A} -object.

Definition 13 Let class dictionary graph Φ be $(VC, VA, \Lambda; EC, EA, EI)$.

- An η -object graph anchored at μ , where $\eta \in VC$, is an object graph anchored at μ with $\lambda(\mu) = \eta$.
- An η -object graph anchored at μ , where $\eta \in VA$, is a γ -object graph for some $\gamma \in VC$ s.t. $\eta \xrightarrow{*} \gamma$.
- $\forall \eta \in VC, Objects(\eta) = \{o | o \text{ is an } \eta\text{-object graph}\}$.
- $\forall \eta \in VA, Objects(\eta) = \bigcup_{u \in \mathcal{A}(\eta)} Objects(u)$.
- $Objects(\Phi) = \bigcup_{u \in VC} Objects(u)$.

■

3 Inductiveness of class dictionary graphs

To reduce the complexity of building objects from class dictionary graphs and the software associated with them, we introduce the *Inductiveness Axiom* for class dictionary graphs.

If a class dictionary graph does not contain any cycle, we can build complex objects from simple objects inductively. The reason is obvious. We can topologically sort any acyclic directed graph, and the topological order tells how to build objects inductively. If a class dictionary graph becomes more and more complex, which means there may be more and more cycles, we can still build objects inductively and incrementally as long as every cycle must have "a way out of cycles". Otherwise we have to build finite cyclic objects for any vertex on those cycles. We argue that non-inductive class dictionary graphs should be avoided most of the time.

Consider the class dictionary graph in Fig. 13a. When we construct a partial class dictionary graph anchored at vertex **Nonempty**, vertex **Nonempty** forces all the outgoing construction and inheritance edges. Vertex **List** must have the only outgoing alternation edge **List** \implies **Nonempty**, because it has an incoming construction edge. Fig. 13b shows the only partial class dictionary graph anchored at vertex **Nonempty**.

Consider the class dictionary graph in Fig. 13c. Fig. 13d shows one of the partial class dictionary graphs anchored at vertex **Nonempty**. The difference from the above case is that we can select alternation edge **List** \implies **Empty** instead of taking alternation edge **List** \implies **Nonempty**.

In the class dictionary graph of Fig. 13a, a **Nonempty**-object must contain an **Element**-object and a **List**-object. A **List**-object is always be a **Nonempty**-object — an infinite recursion. In

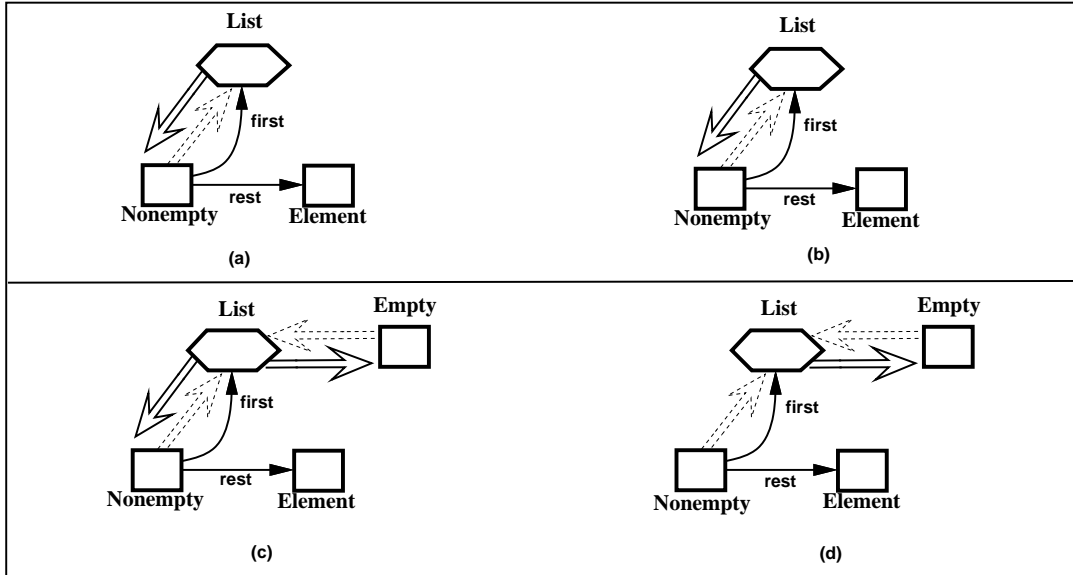


Figure 13: Illustration of partial class dictionary graphs

Fig. 13b, this infinite recursion is expressed by the cycle formed by $\text{Nonempty} \xrightarrow{\text{rest}} \text{List}$ and $\text{List} \Rightarrow \text{Nonempty}$. This cycle is forced to be included.

In the class dictionary graph of Fig. 13c, a **Nonempty**-object must contain an **Element**-object and a **List**-object. But a **List**-object can be an **Empty**-object. In this case, we don't have an infinite recursion. We can have a **Nonempty**-object which is a list containing only one element, an **Element**-object. The **Empty**-object is used here for the end of the list.

Comparing the two class dictionary graphs in Fig. 13a and 13c, we can only build cyclic **Nonempty**-objects from the first class dictionary graph in Fig. 13a; while we can build acyclic **Nonempty**-objects of any size based on the **Nonempty**-objects of smaller size for the second class dictionary graph. We call the second class dictionary graph, an *inductive* class dictionary graph. The first class dictionary graph is not inductive.

Definition 14 (*Inductiveness Axiom*)

A class dictionary graph is *inductive* if for all vertices v there exists at least one cycle-free partial class dictionary graph anchored at v . ■

If a class dictionary graph Φ is not inductive, we call each vertex v in Φ a *noninductive vertex* if there is no cycle-free partial class dictionary graph anchored at v .

The Inductiveness Axiom is fundamental and not obvious. It basically says that each recursion appearing in the class dictionary graph needs to be terminating, or equivalently, that each inductive object definition needs to have a base case.

The purpose of the Inductiveness Axiom is (verbatim quote from [17]):

1. to guarantee that the inductive definitions of the objects which are associated with the vertices of the class dictionary graph, have a base case. Informally, the axiom disallows classes which have only circular objects.
2. to exclude certain useless symbols [13, page 88] from the grammar corresponding to a class dictionary graph. For further information regarding the grammar extension of a class dictionary graph, we refer the reader to [26]. There are two kinds of useless symbols: the ones which cannot be reached from the start symbol and the ones which are involved in an infinite recursion. The Inductiveness Axiom excludes useless symbols of the infinite recursion kind.

The Inductiveness Axiom appears in an unrelated area: automated deduction. Consider a class dictionary graph as a grammar which is written in Horn clause form. Each recursion should “end” in a fact [6].

What is the purpose of an inductive class dictionary graph? An inductive class dictionary graph Φ has to serve as an inductive definition of a non-empty set of objects $InductiveObjects(\Phi)$. If this inductive definition property is violated, we cannot use the powerful mathematical technique, called structural induction, for reasoning about objects and for formally proving properties of them. Structural induction is widely used to prove the correctness of programs and also serves as one motivation for the Law of Demeter [23, 25]. The objects in $InductiveObjects(\Phi)$ are defined in two steps: The base case of an inductive object definition establishes as legal objects the objects of classes which don't have part classes. The inductive object definition then uses the class definitions to define the legal composite objects.

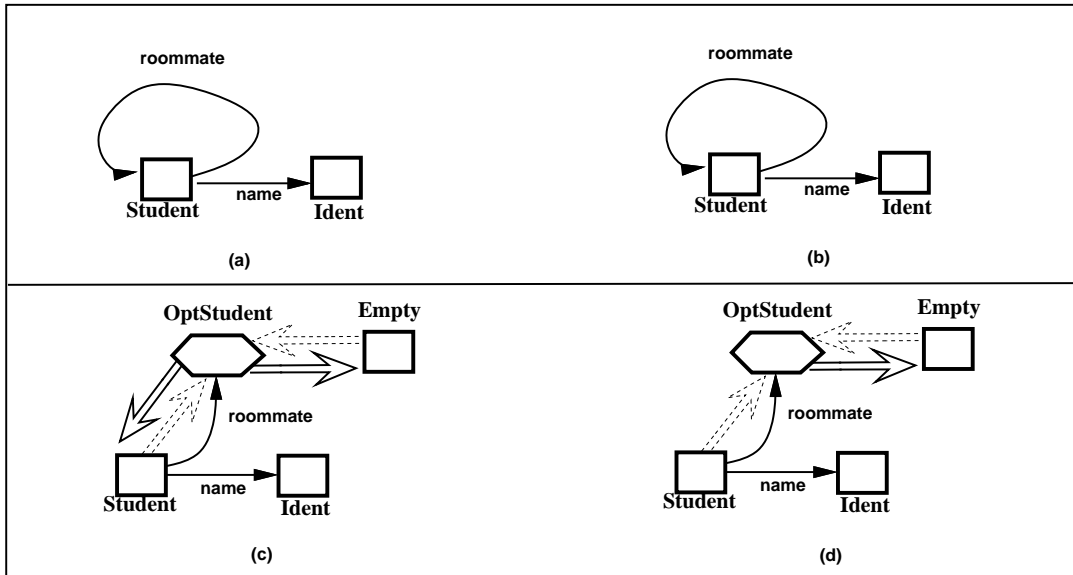


Figure 14: Students and Roommates

Consider the class dictionary graph in Fig. 14a which defines `Student`-objects which must always have a part called `name` and a part called `roommate` which is also a `Student`-object. We

have to stress an important point: The class definition of `Student` enforces that a `Student`-object has two required parts, none of which can be null. Readers who are used to object-oriented programming should remember this point. To express that the part `roommate` can be optional (or null) we have to use the class definition in Fig. 14c in which student objects can have an empty roommate part. The distinction between required and optional parts simplifies the programming : If we know that a part must exist, we don't need an extra method for class `Empty` or a conditional statement which checks whether the part is empty.

The class definition in Fig. 14c provides an inductive definition of `Student`-objects. The base case are students who don't have a roommate or whose roommates are not specified. Out of those simple `Student`-objects, we can build more complex `Student`-objects, using the class definition for `Student`.

The class definition in Fig. 14a, however, is not an inductive definition of `Student`-objects. Therefore, we cannot use structural induction to reason about such objects and therefore, we want to rule out the class dictionary graph in Fig. 14a by an axiom. It is a non-trivial task to find a least restrictive axiom which rules out all class dictionary graphs which don't provide an inductive object definition for all the classes. This problem has been solved with the Inductiveness Axiom.

Consider a university which has a policy that each student must have a roommate. Why should our data model disallow to express this integrity constraint? We distinguish between two phases of the `Student`-objects: the input phase, during which we read in `Student`-objects from a linear, pointer-free description, and the processing phase, during which the `Student`-objects are processed. Our data model is tuned to model requirements for the input phase while allowing stronger requirements for the processing phase.

To allow concise and short descriptions for objects, class dictionary graphs are extended with terminals to define languages [20][27][28]. A class dictionary graph with terminals is called a *class dictionary*, a context-free grammar. A printing procedure of a few lines defines how objects are printed as sentences. The set of all legal tree objects in their printed form is the language defined by a class dictionary. To allow a fast transformation of a sentence into a tree object, a class dictionary needs to satisfy also the Inductiveness Axiom and two LL(1) rules. Under those conditions, the printing function is a bijection between objects and sentences and it has an inverse which is naturally called a parsing function. The parsing function is easily implemented by a recursive-descent parser. The concise object descriptions are used in the C++ Demeter System to debug object-oriented designs, before any C++ code is hand-coded. High-level debugging of object-oriented designs is very valuable and the Inductiveness Axiom is exactly what is needed to offer this capability. The semantic checker of the C++ Demeter System checks all three axioms (plus some additional rules) and is a valuable tool in debugging object-oriented data models.

We distinguish two kinds of object descriptions. The sentences defined by a class dictionary are called *declarative* object descriptions. The statements for constructing objects in some object-oriented programming language are called *imperative* object descriptions, like `new C()` in C++. Sentences are also called linear, pointer-free object descriptions, since there is no object pointer

in sentences.

During the input phase, the university policy, that every student must have a roommate, is not enforced, otherwise the sentence, a linear, pointer-free student description would be infinite. During the processing phase the university policy that every student must have a roommate can be enforced.

We conclude the discussion of the motivation for the Inductiveness Axiom with the following comparison:

- If the Inductiveness Axiom is violated for a class dictionary graph Φ , then there is a vertex $v \in V_\Phi$ such that $Objects(v)$ contains only circular objects. A circular object is an object which contains cycles in its part-of relationships.
- If the Inductiveness Axiom is satisfied, then for all vertices $v \in V, Objects(v)$ contains both inductively defined, non-circular objects as well as circular objects.

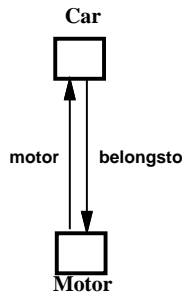


Figure 15: Car and Motor

Sometimes, people may want to keep their class dictionary graphs noninductive for some purposes, as shown in Fig. 15. Every **Car**-object must have a **Motor**-object. Every **Motor**-object must have a **Car**-object that it is installed on. Therefore we propose an approximation of the Inductiveness Axiom.

Law of Demeter for Classes

Minimize the number of noninductive vertices in a class dictionary graph.

We claim that when people minimize the number of noninductive vertices in a class dictionary graph, they minimize the complexity of building objects for it and the software associated with it.

In Fig. 16, we give two different class dictionary graphs for defining the structure of a company. In Fig. 16a, a company must be located in at least one area. In each area, there must be at least one division. Inside each division, there must be at least one department. Each department must have at least one employee. Each employee must work for this company. The vertices,

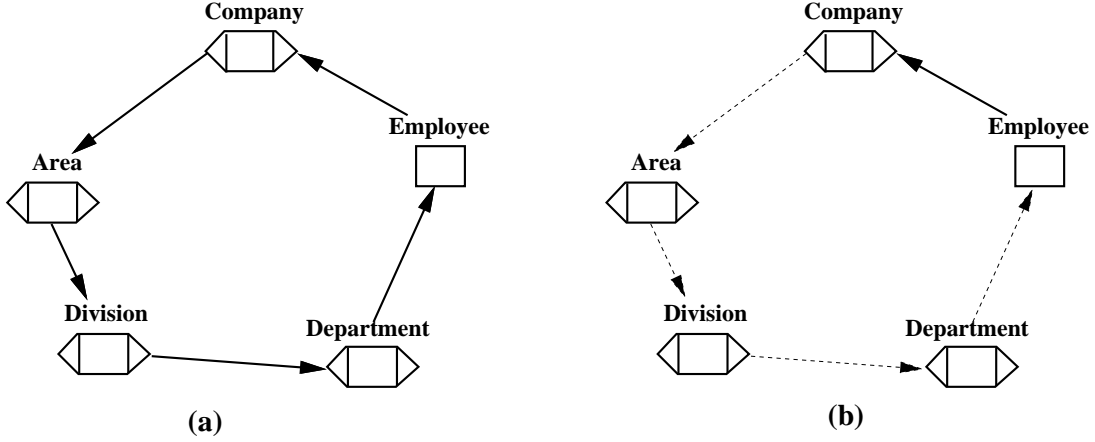


Figure 16: Illustration of the Law of Demeter for Classes

Company, Area, Division and Department, are repetition vertices. The solid lines outgoing from repetition vertices in Fig. 16a mean each repetition object must own at least one part-object.

If we build a Company-object for the class dictionary graph in Fig. 16a, we have to build Company-objects, Area-objects, Division-objects, Department-objects and Employee-objects “at the same time” and link them “at the same time” in order to have legal objects.

We don’t have such a restriction to build objects for the class dictionary graph in Fig. 16b. The dotted lines mean each repetition object can have zero or more part-objects. We can check the missing semantics or even link them together by running some code after we build the objects.

The point we want to make is that the class dictionary graph in Fig. 16b really simplifies the processing of objects.

The Unique Label Axiom, Cycle-Free Alternation and Inheritance Axiom and Inductiveness Axiom imply a mathematical theory.

Theorem 1 *In a class dictionary graph $\Phi = (VC, VA, \Lambda; EC, EA, EI)$, if Φ satisfies the Inductiveness Axiom, then Φ satisfies the Cycle-Free Alternation Axiom.*

Proof: If there is an alternation cycle in Φ , as shown in Fig. 17, then the inheritance cycle, $A \rightsquigarrow B \rightsquigarrow C \rightsquigarrow A$, is always forced to be in any partial class dictionary graph anchored at A. Therefore the Inductiveness Axiom is violated.

Theorem 2 *There is no cyclic construction path in a class dictionary graph $\Phi = (VC, VA, \Lambda; EC, EA, EI)$, i.e., for all $v \in VC \cup VA$ there is no construction path from v to v .*

Proof: If there is a cyclic construction path then no vertex v on the path will have a cycle-free partial class dictionary graph since all construction edges leaving a vertex must be included in

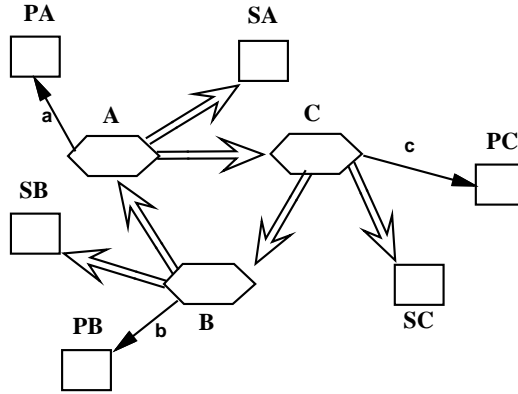


Figure 17: An alternation cycle

the partial class dictionary graph containing that vertex. Therefore the Inductiveness Axiom axiom is violated.

Further results of the theory of class dictionary graphs are given in [22], [21], [4], [5] and [14].

4 Programming with class dictionary graphs

We show with a simple example how we use class dictionary graphs to simplify programming. We have developed a CASE tool for C++ [34], the C++ Demeter System [26], which maps class dictionary graphs into a C++ class library. The C++ class library is then enhanced with propagation patterns [19, 17] or manually with C++ member functions implementing the application. To each vertex corresponds a C++ class with a constructor and to each alternation vertex corresponds an abstract C++ class.

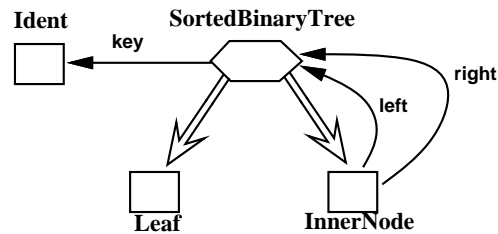


Figure 18: Sorted binary tree

Consider the class dictionary graph in Fig. 18. We want to implement a searching algorithm which looks for some leaf with a given key. The parts of the C++ program, shown below, are generated from the class dictionary graph in Fig. 18 by the Demeter System.

```
class SortedBinaryTree : public Construction {
private:
    Ident *key;
```

```

public:
    SortedBinaryTree( Ident * = NULL );
    ~SortedBinaryTree();

    Ident *get_key() { return( key ); }
    void set_key( Ident *new_key ) { key = new_key; }

    virtual void      DEM_abstract() = 0;
};

class InnerNode : public SortedBinaryTree {
private:
    SortedBinaryTree *left;
    SortedBinaryTree *right;
public:
    InnerNode( SortedBinaryTree * = NULL, SortedBinaryTree * = NULL );
    ~InnerNode();

    SortedBinaryTree *get_left() { return( left ); }
    void set_left( SortedBinaryTree *new_left ) { left = new_left; }
    SortedBinaryTree *get_right() { return( right ); }
    void set_right( SortedBinaryTree *new_right ) { right = new_right; }

    void DEM_abstract() { }
};

class Leaf : public SortedBinaryTree {
public:
    Leaf();
    ~Leaf();

    void DEM_abstract() { }
};

class Ident : public Terminal {
private:
    char * val;
public:
    Ident( char * = 0 );
    ~Ident();

    char *get_val() { return( val ); }
    void set_val( char * new_val ) { val = new_val; }
};

```

```

int operator==(Ident* iIdent)
    { return (strcmp(val,iIdent->get_val())==0); }
int operator>(Ident* iIdent)
    { return (strcmp(val,iIdent->get_val())>0); }
int operator<(Ident* iIdent)
    { return (strcmp(val,iIdent->get_val())<0); }
};

```

`Construction` and `Terminal` are predefined system classes. There are many generic member functions defined for them, including printing and parsing functions. The member function, `DEM_abstract()`, makes alternation classes uninstantiable.

The actual implementation of the searching algorithm is the following.

```

void SortedBinaryTree::find(Ident *akey) {
    // this is a virtual function
}

void Leaf::find(Ident *akey) {
    if (akey==this->get_key())
        cout << "found\n";
    else
        cout << "not found\n";
}

void InnerNode::find(Ident *akey) {
    if (akey>this->get_key())
        right->find(akey);
    else
        if (akey<this->get_key())
            left->find(akey);
        else
            cout << "found\n";
}

```

5 Class dictionary transformations

The purpose of this section is to reduce the problem of checking the Inductiveness Axioms for class dictionary graphs to a simpler problem: We show that it is sufficient to consider only flat class dictionary graphs where all inheritance has been flattened. To give the reduction, we need several definitions.

Two class dictionary graphs Φ, Ψ are object-equivalent, if both define the same set of objects,

i.e., $Objects(\Phi) = Objects(\Psi)$. We give an object-equivalence definition without referring to objects.

Definition 15 Let Φ and Ψ be two class dictionary graphs, where $\Phi = (VC_\Phi, VA_\Phi, \Lambda_\Phi; EC_\Phi, EA_\Phi, EI_\Phi)$, $\Psi = (VC_\Psi, VA_\Psi, \Lambda_\Psi; EC_\Psi, EA_\Psi, EI_\Psi)$. Class dictionary graphs Φ and Ψ are *object-equivalent* if

$$VC_\Phi = VC_\Psi$$

and for all $v \in VC_\Phi$

$$PartClusters_\Phi(v) = PartClusters_\Psi(v).$$

■

Informally, a class dictionary graph is flat, if all inheritance has been flattened.

Definition 16 A class dictionary graph Φ is *flat* if no alternation vertex has an outgoing construction edge, i.e.,

$$\{v \xrightarrow{l} w | v \in VA, w \in V, l \in \Lambda\} = \emptyset.$$

■

Theorem 3 Given a legal class dictionary graph Φ , we can efficiently find a flat class dictionary graph Ψ which is object-equivalent to Φ . ■

Proof:

The theorem is proved by introducing an algorithm called **CommonExpansion**.

Algorithm CommonExpansion(Φ)

Input :A class dictionary graph Φ . The immediate parts of vertex v are stored in $v.immediate_parts$.

Output:A class dictionary graph Ψ with “common parts expanded”, i.e., Φ is object-equivalent to Ψ and Ψ is flat. The expanded common parts of vertex v are stored in $v.expanded_common_parts$.

for each vertex v of Φ **do**

$v.mark = \mathbf{false}$

for each vertex v of Φ **do**

if not $v.mark$ **then** **DFS**(Φ, v)

for each vertex v of Φ **do**

$v.immediate_parts = v.expanded_common_parts$

DFS(Φ, v)

$v.expanded_common_parts = v.immediate_parts$

for each $v \rightsquigarrow w \in EI$ **do**

if not $w.mark$ **then** **DFS**(Φ, w)

$v.expanded_common_parts = v.expanded_common_parts \cup w.expanded_common_parts$

$v.mark = TRUE$

Let $\Psi = \text{CommonExpansion}(\Phi)$. It follows immediately from the above algorithm that Ψ has the following properties:

1. $VC_\Psi = VC_\Phi$
2. $VA_\Psi = VA_\Phi$
3. $EA_\Psi = EA_\Phi$
4. $EI_\Psi = EI_\Phi$
5. for all $v \in VC_\Psi$, $Parts_\Psi(v) = Parts_\Phi(v)$.

Therefore $Objects(\Psi) = Objects(\Phi)$. Fig. 19 shows the flat class dictionary graph of the class dictionary graph in Fig. 2.

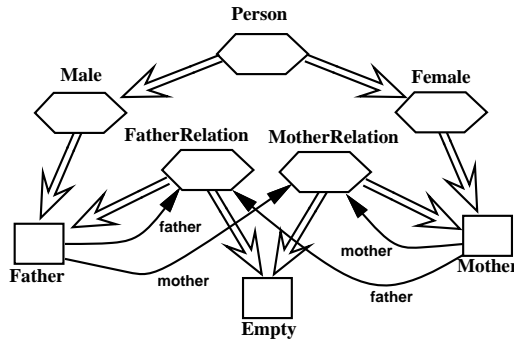


Figure 19: A flat class dictionary graph

Next we show that every time we check a class dictionary graph against the Inductiveness Axiom, we can check its flat class dictionary graph instead.

Theorem 4 *Given a legal class dictionary graph Φ and a class dictionary graph Ψ with $\Psi = \text{CommonExpansion}(\Phi)$,*

- Ψ is also legal and
- Ψ satisfies the Inductiveness Axiom iff Φ satisfies the Inductiveness Axiom.

We prove the theorem using several lemmata.

Lemma 1 *Assume that a class dictionary graph Φ satisfies the Cycle-Free Alternation Axiom and the Unique Label Axiom and $\Psi = \text{CommonExpansion}(\Phi)$. Ψ satisfies both axioms.*

Proof:

Ψ has the following properties:

1. $VC_\Psi = VC_\Phi$
2. $VA_\Psi = VA_\Phi$
3. $EA_\Psi = EA_\Phi$
4. $EI_\Psi = EI_\Phi$
5. for any $v \in VC_\Psi$, $Parts_\Psi(v) = Parts_\Phi(v)$.

In other words, for any vertex $v \in VA_\Psi$, we do not add or delete any outgoing alternation edges; we do not add any part to $Parts_\Psi(v)$ or delete any part from $Parts_\Psi(v)$. Therefore, if class dictionary graph Φ satisfies the Cycle-Free Alternation Axiom and the Unique Label Axiom, Ψ also satisfies the same two axioms.

Lemma 2 *Assume that a class dictionary graph Φ satisfies the Cycle-Free Alternation Axiom and the Unique Label Axiom and $\Psi = CommonExpansion(\Phi)$. For any vertex v in Φ , if there is a cycle-free partial class dictionary graph P anchored at vertex v , then $CommonExpansion(P)$ is a cycle-free partial class dictionary graph in Ψ .*

Proof:

Let P' be $CommonExpansion(P)$. When we build a partial class dictionary graph P anchored at vertex v , we include all the inheritance edges outgoing from any vertex in P . The vertices in P' have all the parts as they have in Ψ . Therefore if P is a partial class dictionary graph anchored at vertex v in Φ , P' is also a partial class dictionary graph anchored at vertex v in Ψ .

In P , each alternation vertex u may have some outgoing construction edges, but in P' vertex u doesn't have any outgoing construction edge. Instead, all construction vertices, which are alternation-reachable from u , have all the construction edges. This is the only difference between P and P' . If some parts in P can be reached from one vertex through an alternation path or an inheritance path, they can also be reached in P' from the same vertex through an alternation path. So if there is a cycle C in P , there must be a cycle in P' . Therefore P' is also cycle-free.

Lemma 3 *Assume that a class dictionary graph Φ satisfies the Cycle-Free Alternation Axiom and the Unique Label Axiom and $\Psi = CommonExpansion(\Phi)$. For any vertex w in Ψ , if there is a cycle-free partial class dictionary graph P anchored at vertex w , then we can find a cycle-free partial class dictionary graph P' anchored at vertex w in Φ such that $P = CommonExpansion(P')$.*

Proof:

Let P' satisfy the following properties:

1. $VC_{P'} = VC_P$
2. $VA_{P'} = VA_P$
3. $EA_{P'} = EA_P$
4. $EI_{P'} = EI_P$
5. $\Lambda_{P'} = \Lambda_P$
6. $EI_{P'} = EI_P$
7. for any $v \in VC_{P'} \cup VA_{P'}$,

$$v \xrightarrow{l} u \in EC_{P'} \text{ iff } v \xrightarrow{l} u \in EC_{\Phi}$$

where $u \in VC_{P'} \cup VA_{P'}$ and $l \in \Lambda_{P'}$.

Then the vertices in P' have all the parts as in Φ . Therefore if P is a partial class dictionary graph anchored at vertex w in Ψ , P' is also a partial class dictionary graph anchored at vertex w in Φ . For the same reason as in Lemma 2, P' is also cycle-free.

Now we are ready to prove theorem 4.

Lemma 1 proves the first half of the theorem. Next we give the proof for the second half.

Proof: \Leftarrow : Suppose for any vertex v in Φ there is a cycle-free partial class dictionary graph S anchored vertex v . Let S' be $CommonExpansion(S)$. Then according to Lemma 2, S' is also a partial class dictionary graph anchored vertex v in Ψ and is cycle-free.

Therefore Ψ is inductive if Φ is inductive.

\Rightarrow : Suppose for any vertex w in Ψ there is a cycle-free partial class dictionary graph P anchored at vertex w . Then according to Lemma 3, there is a partial class dictionary graph P' anchored at vertex w in Φ such that $P = CommonExpansion(P')$, and P' is cycle-free.

Therefore Φ is inductive iff Ψ is inductive.

6 Data model axiom checking

It is easy to derive efficient algorithms for checking the Cycle-Free Alternation Axiom and the Unique Label Axiom. We leave them as an exercise to the reader. However, it is not obvious how to check the Inductiveness Axiom efficiently since a vertex can have many partial class dictionary graphs. In this section we present two efficient algorithms the first of which is

based on strongly connected components and the second is based on finding “useful classes”. We present both algorithms because in conjunction with other design checking tasks, a combination of both algorithms achieves best results.

These two algorithms work on flat class dictionary graphs. Therefore the algorithms can ignore inheritance edges since there are no outgoing construction edges from alternation vertices and consequently none of the paths in a flat class dictionary graph can contain inheritance edges. By ignoring inheritance edges, we can work in the conceptual world of standard graph theory, i.e., the paths are in E^* where $E = EC \cup EA$.

Next, we present the first algorithm, called **SCC_Check**.

6.1 Strongly-connected components

Consider the class dictionary graph which is shown in Fig. 20a.

This class dictionary graph is not inductive, since vertex B has no cycle-free partial class dictionary graph. When we construct a partial class dictionary graph anchored at vertex B , we must always choose the three edges $B \xrightarrow{f} F$, $F \implies E$ and $E \implies B$.

We recall some of the definitions from graph theory. For the definitions and algorithms not given here, we refer the reader, e.g., to [30] or the original source for computing strongly connected components efficiently [35]. A directed graph is **strongly connected** if, for every pair of vertices v and w , there is a path from v to w and a path from w to v . A **strongly connected component (SCC)** is a maximal subset of the vertices such that its induced subgraph is strongly connected.

The **SCC** graph of a graph is constructed in this way:

- the nodes in the **SCC** graph correspond to the strongly connected components;
- there is a directed edge from node a to node b if there is a directed edge (in the original graph) from some vertex in node a to some vertex in node b .

For convenience, when we refer to a strongly connected component, we sometimes also include the edges leaving the strongly connected component.

An **SCC** graph has the following properties:

- The **SCC** graph is acyclic.
- If a node contains more than one vertex of the original graph, the vertices must be on a cycle in the original graph.

Algorithm **SCC_Check** is now described informally. It decomposes a given class dictionary graph into its strongly connected components. For each component we delete all construction

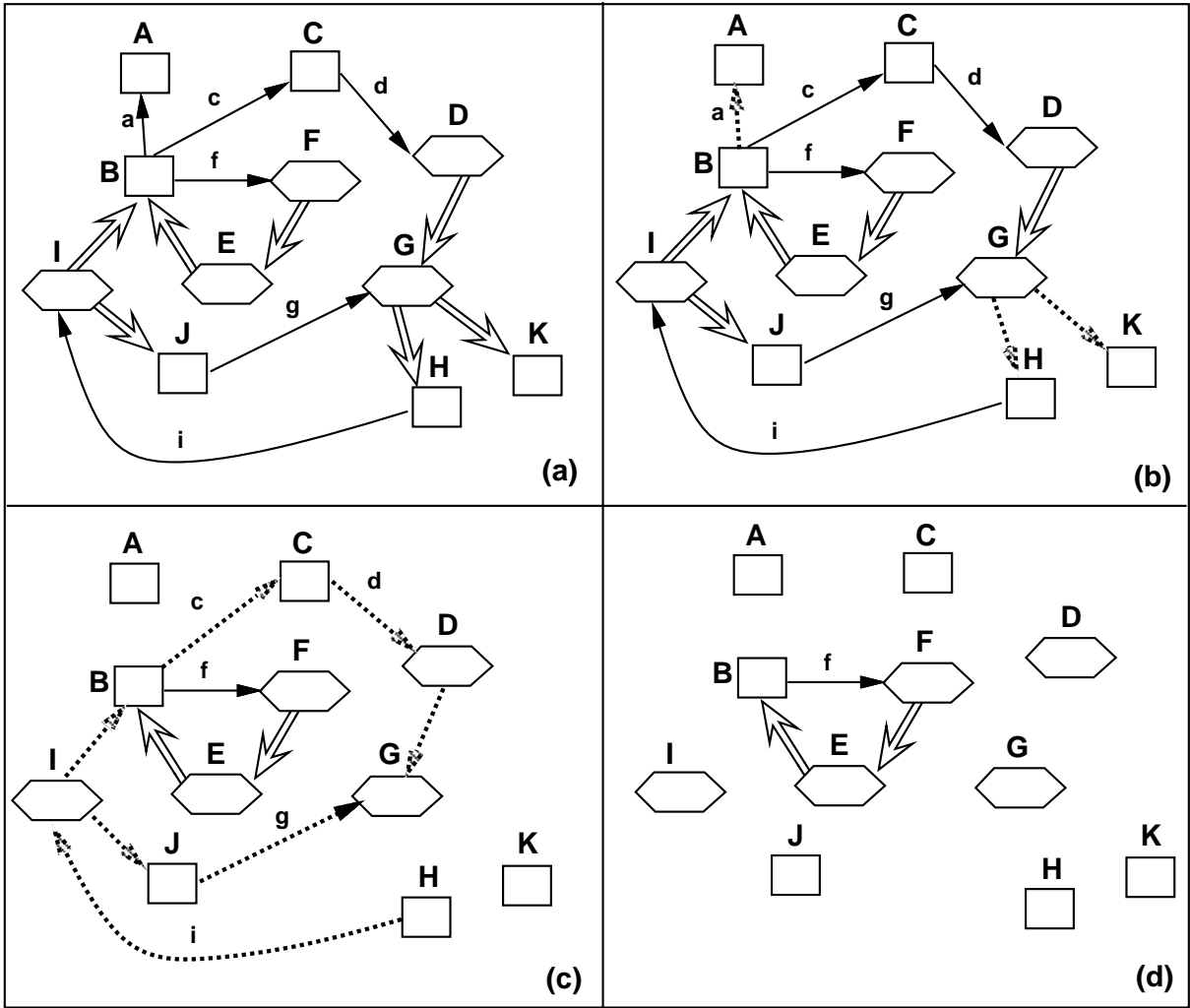


Figure 20: A violation graph for the Inductiveness Axiom

SCC_Check(Φ);

input: A class dictionary graph Φ .

output: An error message if the Inductiveness Axiom is violated.

SCC(Φ);

{Treat construction edges and alternation edges equally, label each vertex in Φ with *Component Number*. Vertices in the same **SCC** are labeled with the same number.}

SCC_list=**Decompose**(Φ); {split Φ into **SCCs**}

if **SCC_list** contains only one element C **then**

Check_Cycle(C);

 { If C contains a cycle, report Axiom violation. }

else

for each **SCC** C in **SCC_list** **do**

for each edge e in C **do**

if e is an alternation edge $v \Longrightarrow w$ and w is not in C **then**

 delete all alternation edges that exit vertex v ;

if e is a construction edge $v \xrightarrow{l} w$ and w is not in C **then**

 delete e ;

SCC_Check(C);

Figure 21: **SCC_Check**

edges which exit the component and for each alternation vertex which has at least one alternation edge leaving the component we delete all its outgoing alternation edges (also the ones inside the component). This decomposition gives us a list of semi-class dictionary graphs which have to be checked recursively. We repeat the decomposition until the process stops, i.e., until each subgraph is invariant under the above decomposition. We will show that the original class dictionary graph is illegal iff the decomposition results in a subgraph which has more than one vertex or a vertex with a self-loop.

For the graph in Fig. 20a, there are three **SCCs**, which are $\{B, C, D, E, F, G, H, I, J\}$, $\{A\}$ and $\{K\}$. There is no cycle in the last two **SCCs**. For the first **SCC**, $G \Longrightarrow K$ is an outgoing alternation edge. We delete all outgoing alternation edges of vertex G . Vertex B has an outgoing construction edge $B \xrightarrow{a} A$ which we delete. Now we have a new derived graph shown in Fig. 20b. The dotted arrows are deleted edges.

We decompose the new graph again, and we have seven **SCCs** which are $\{C\}$, $\{D\}$, $\{G\}$, $\{H\}$, $\{I\}$, $\{J\}$ and $\{B, F, E\}$. They are shown in Fig. 20c. The first six reduced subgraphs are acyclic. The seventh subgraph contains a cycle with three vertices (Fig. 20d). Therefore the original class dictionary graph is not inductive.

The algorithm is given formally in Fig. 21.

6.1.1 Correctness

Assume that Φ is a class dictionary graph with **SCC** graph $\mathbf{SCC}(\Phi)$. Assume that there exists a vertex v in Φ which has only cyclic partial class dictionary graphs, and that vertex v is in node N of $\mathbf{SCC}(\Phi)$. Since $\mathbf{SCC}(\Phi)$ is acyclic, we assume that all the vertices in the nodes of $\mathbf{SCC}(\Phi)$ reachable from node N have a cycle-free partial class dictionary graph.

After the deletion of a construction edge which exits from N , vertex v still has only cyclic partial class dictionary graphs. Similarly, After the deletion of all alternation edges of a vertex which has at least one alternation edge leaving from N , vertex v still has only cyclic partial class dictionary graphs.

When we apply the algorithm again to the reduced strongly connected components we must eventually obtain a strongly connected component which contains a cycle. Therefore, the algorithm will correctly report that a given noninductive class dictionary graph is not inductive.

Assume that the algorithm checks a graph Φ , and that it finds during the decomposition process an **SCC** graph which has only one node N with a cycle in it. For any vertex v in node N , no matter what vertices and edges that are added by the inverse of the decomposition process, we cannot avoid selecting all the edges in the cycle. So vertex v has no cycle-free partial class dictionary graph in the original class dictionary graph. The original class dictionary is not inductive.

6.1.2 Analysis

Let Φ be a class dictionary graph with $\Phi = (VC_\Phi, VA_\Phi, \Lambda; EC_\Phi, EA_\Phi, EI_\Phi)$ and $V_\Phi = VC_\Phi \cup VA_\Phi$. Assume $n = |V_\Phi| + |EC_\Phi| + |EA_\Phi|$. The amount of space needed by algorithm **SCC_Check** is $O(n)$.

The total time needed by algorithm **SCC_Check** is

$$\mathbf{T}(\Phi) = \begin{cases} \mathbf{T}_{\mathbf{SCC}}(\Phi) + \sum \mathbf{T}_{\mathbf{delete}}(\Phi_i) + \sum \mathbf{T}(\Phi_i) & \text{if } \Phi \text{ is not strongly connected and } \Phi_i \text{ is a SCC of } \Phi; \\ \mathbf{T}_{\mathbf{SCC}}(\Phi) + \mathbf{T}_{\mathbf{check}}(\Phi), & \text{if } \Phi \text{ is strongly connected.} \end{cases}$$

We use a linked adjacency list representation for Φ . The list could be constructed in time $O(\log(|V_\Phi|) \cdot n)$.

$$\begin{aligned} \mathbf{SCC} \text{ decomposition cost: } & \mathbf{T}_{\mathbf{SCC}}(\Phi) = O(n) \\ \text{edge deletion cost: } & \mathbf{T}_{\mathbf{delete}}(\Phi) = O(|EC_\Phi| + |EA_\Phi|) \\ \text{cycle checking cost: } & \mathbf{T}_{\mathbf{check}}(\Phi) = O(n) \end{aligned}$$

There exists a constant $c > 0$ such that for all (sufficiently big) subgraphs \mathcal{G} (including Φ) found by the above decomposition:

$$\mathbf{T}(\mathcal{G}) \leq \begin{cases} c \cdot \mathcal{N}(\mathcal{G}) + \sum \mathbf{T}(\mathcal{G}_i) & \text{if } \mathcal{G} \text{ is not strongly connected and } \mathcal{G}_i \text{ is a SCC of } \mathcal{G}; \\ c \cdot \mathcal{N}(\mathcal{G}), & \text{if } \mathcal{G} \text{ is strongly connected.} \end{cases}$$

$$\mathcal{N}(\mathcal{G}) = |V(\mathcal{G})| + |EC(\mathcal{G})| + |EA(\mathcal{G})|.$$

There exists a function $\mathcal{F}(n)$ such that

$$\mathbf{T}(\Phi) \leq \mathcal{F}(\mathcal{N}(\Phi)) \leq \begin{cases} c \cdot \mathcal{N}(\Phi) + \sum_{i=1}^l \mathcal{F}(\mathcal{N}(\Phi_i)) & \text{if } \Phi \text{ is not strongly connected and } \Phi_i \text{ is a SCC of } \Phi, l > 1; \\ c \cdot \mathcal{N}(\Phi), & \text{if } \Phi \text{ is strongly connected.} \end{cases}$$

We know that

$$\mathbf{T}(\Phi) = O(\mathcal{F}(\mathcal{N}(\Phi))).$$

$\mathcal{F}(\mathcal{N}(\Phi))$ is defined recursively with a recursion depth of not more than $|V_\Phi|$. At each recursion level the subgraphs don't overlap with each other and the sum of the total number of vertices and edges is no more than $\mathcal{N}(\mathcal{G})$. So

$$\mathcal{F}(\mathcal{N}(\Phi)) \leq |V_\Phi| \mathcal{N}(\mathcal{G}).$$

Therefore, $\mathbf{T}(\Phi) = O(|V_\Phi| (|V_\Phi| + |EC_\Phi| + |EA_\Phi|))$.

6.2 Useful vertices

Next we present a faster algorithm which is inspired by the theory of formal languages. It is very convenient to view a class dictionary graph as a language definition [20].

Informally, a construction vertex C is useful in a class dictionary graph Φ if C is used in some finite cycle-free object in $Objects(\Phi)$, i.e., $Objects(\Phi)$ contains (finite) cycle-free objects of class C .

Definition 17 A construction vertex C of a class dictionary graph Φ is *useful*, if there is a cycle-free C-object in $Objects(\Phi)$. ■

The algorithm is based on the following theorem.

Theorem 5 The following holds for a legal class dictionary graph: The Inductiveness Axiom is satisfied iff all construction vertices are useful. ■

Proof:

We first show that if the Inductiveness Axiom is satisfied then all construction vertices are useful. The Inductiveness Axiom implies that for any construction vertex C there exists a cycle-free partial class dictionary graph P anchored at C . We select P such that each alternation vertex has exactly one outgoing alternation edge. There is only one way to map this subgraph P into an object graph, and this object graph is cycle-free. Therefore C is useful.

The reverse we show by contraposition: if the Inductiveness Axiom is not satisfied, there must exist a construction vertex C which has only cyclic partial class dictionary graphs anchored at C . We build a C -object by recursively selecting all construction edges and at least one alternation edge. We must return to some vertex which we have already selected and which does not have a cycle-free partial class dictionary graph. Since the Cycle-Free Alternation Axiom is satisfied, the size of the object has increased when we return to this vertex. Therefore, if we don't build an object whose corresponding object graph contains a cycle, there is no way to build a finite C -object.

Determining useful construction vertices in a class dictionary graph is similar to determining useful symbols in a grammar. Our algorithm follows the pattern of the algorithm given in [13, page 88] for determining useless symbols in a grammar. Informally, the algorithm first puts all construction vertices without outgoing construction edges into set *Useful_Vertices* and then new vertices are added to *Useful_Vertices* according to the following rule: A vertex becomes useful when *all* its construction successors are useful and if *at least one* alternation successor is useful.

The algorithm is given formally in Fig. 22.

We use the class dictionary graph in Fig. 20. Initially *Useful_Vertices* contains vertices A and K. Since K is an alternation descendant of G, we add G. Since J has G as a part class, we add J to *Useful_Vertices*. This process repeats until *Useful_Vertices* contains A, K, G, J, I, H, D and C. But B, E and F are never added and therefore the Inductiveness Axiom is violated.

6.2.1 Correctness and analysis

If a class dictionary graph Φ satisfies the Inductiveness Axiom, every vertex v of Φ should have at least one cycle-free partial class dictionary graph anchored at v . This algorithm will eventually add vertex v into *Useful_Vertices*.

Assume that after completion of the algorithm there are some vertices left without being added into *Useful_Vertices*. Consider the reduced graph which we obtain from the original class dictionary graph by deleting the vertices in *Useful_Vertices* using the rule given in the strongly-connected components based algorithm. This reduced graph must not contain any leaf and therefore the Inductiveness Axiom is not satisfied.

The running time of this algorithm is $O(|V_\Phi| + |EC_\Phi| + |EA_\Phi|)$.

We compare the two algorithms whose names we abbreviate as Connected and Useful. Useful

Algorithm Useful_Check(Φ);

input: A class dictionary graph Φ . V is the set of vertices of Φ .

output: An error message if the Inductiveness Axiom is violated.

{Initialize}

for each vertex v **do**

$v.part_number$ = number of outgoing construction edges of v

for each alternation vertex v **do**

$v.alternatives$ = 0

for each construction vertex v **do**

$v.alternatives$ = 1

for each vertex v **do**

$v.treated$ = 0

Collect all the vertices with no outgoing edge into the set, *Useful_Vertices*.

for each vertex v in *Useful_Vertices* with $v.treated = 0$ **do**

$v.treated$ = 1

for all parents p of v **do**

if v is a part

then decrement $p.part_number$ by 1

if $p.part_number = 0$ and $p.alternatives = 1$

then add p to *Useful_Vertices*

if v is an alternative

then $p.alternatives = 1$

if $p.part_number = 0$ and $p.alternatives = 1$

then add p to *Useful_Vertices*

if $|Useful_Vertices| = |V|$

then the Inductiveness Axiom is satisfied

else the Inductiveness Axiom is violated (print error message)

Figure 22: Useful_Check

is faster and easier to implement than Connected. So if we are only interested in checking the Inductiveness Axiom, then we should use algorithm Useful. However, when we analyze a class dictionary graph, we also would like to know what the start classes are, i.e., a minimum set of classes from which any class can be reached along some edges. In graph-theoretic terms, this is called a vertex basis. To compute the vertex basis of a class dictionary graph, one best computes the strongly connected components which are then topologically sorted. The vertex basis can be easily computed from the resulting **SCC** graph.

7 Related work

Our data model has a strong grammatical orientation: a class dictionary graph defines a context-free language for describing the objects [20, 26]. Therefore, the papers [11, 2] are related, but they address other issues than an axiomatic foundation for grammatical data modelling. The main benefit of grammatical data modelling is to have an automatic and cheap mechanism to populate object-oriented data bases from text which describes the objects. The text is often several factors shorter than a complete object composition description, yet it contains the same information.

Class dictionary graphs are an abstraction of semantic networks with part-of and is-a links. Semantic networks have been studied extensively in the AI literature [7]. Class dictionary graphs are also related to AND-OR graphs which have been used in AI [7]. The Interface Description Language described in [33] uses structures related to class dictionary graphs. However, the specific axiomatic structure which we introduce here as well as our efficient axiom checking algorithm are new, to the best of our knowledge.

The relationships between this paper and previous papers on the Demeter Kernel Model are as follows:

- New

The following concepts are new in this paper:

Law of Demeter for classes, efficient algorithms for checking the Inductiveness Axiom (algorithms CommonExpansion, SCC_Check, Useful_Check).

- Improvements

We have attempted a definition of partial class dictionary graphs in [21]. Here we improve that definition.

We also improve the definition and axioms of object-graphs in [22].

- Reuse

In our work of developing a theoretical foundation for engineering adaptive software, we need to quote several of the key definitions in order to keep this paper self-contained.

Definitions are reused verbatim or almost verbatim as follows: definitions 1 through 10, 12, 13 and the Inductiveness axiom are from [17]; definition 15 is from [4]; Fig. 6 and 10 are reproduced from [17].

8 Conclusions

Class dictionary graphs are fundamental to object-oriented data modeling and programming; most data modelling and programming languages which support the object-oriented paradigm can express class dictionary graph structures. Therefore it is worthwhile to study the formal properties of class dictionary graphs.

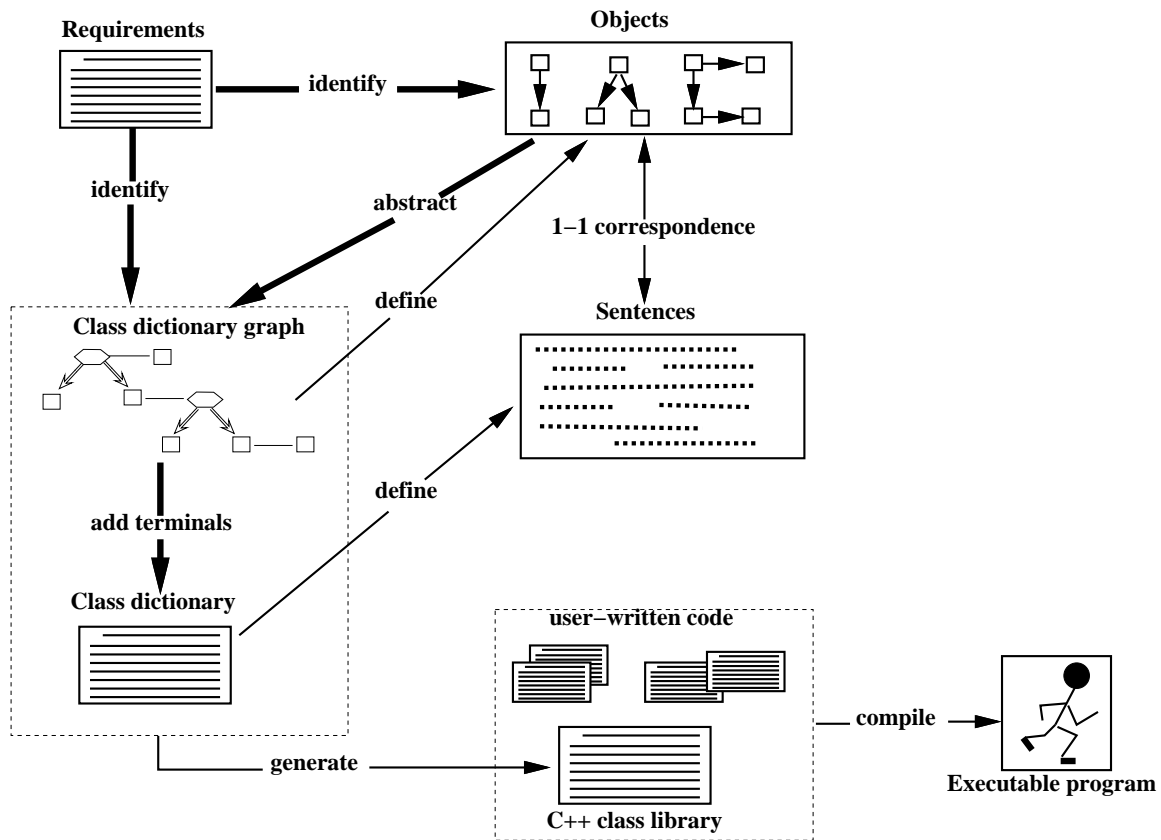


Figure 23: Design and program using the Demeter Method

Class dictionary graphs are the foundation of the Demeter system which consists of the Demeter Method and a set of Tools. The Tools support the Method on the Unix operating system. The Method and the Tools support iterative software development. The essence of the Method is summarized in Fig. 23.

From a requirement specification, either a class dictionary graph is extracted manually or objects are defined which are then abstracted by the learning tool into a class dictionary

graph[22, 5]. The class dictionary graph is then checked whether it satisfies the axioms, and it is made inductive, if necessary. Optimization [22, 21] and normalization [20] are performed on the class dictionary graph.

The class dictionary graph is then extended into a class dictionary by adding terminals. The resulting class dictionary is checked for the LL(1) conditions [28, 20]. To debug the class dictionary, it is translated into a parser which is fed several object descriptions (in sentence form defined by the class dictionary). If the parser does not accept a sentence, then there is an error either in the class dictionary or in the sentence or in both.

After debugging the class dictionary, we generate a C++ environment which provides the primitive services for working with the objects defined by the class dictionary: parsing, printing, drawing, copying, creating, comparing, getting, setting etc.

Based on the design requirements, the user writes C++ code in the generated environment. The C++ code includes propagation patterns [19, 17] which are a high level abstraction of object-oriented programs. When using a different class dictionary graph to interpret a given propagation pattern, we will get a different program. In other words, each propagation pattern defines a family of programs. We can use a class dictionary graph to select a member from the family. A propagation tool automates the selection process.

Our experiences show that the Method and the Tools smooth the software development process and promote developers' productivity.

Acknowledgments: Many thanks to the other members of the current Demeter team, including Paul Bergstein, Ian Holland, Walter Hürsch and Nacho Silva-Lepe for their encouragement and support with implementations, paper feedback and reorganization proposals.

Paul Bergstein and Nacho Silva-Lepe were influential in developing the key definitions for class dictionary graphs and their transformations [21, 22, 5, 4]. Ken Baclawski influenced the Law of Demeter for classes. We would like to thank Juan Rodriguez for implementing the cycle-free subgraph axiom checker which is based on strongly connected components.

This research was supported in part by grants from IBM Corporation, Mettler-Toledo AG and National Science Foundation(Grant CCR-9102578) and CDA-9015692 (Research Instrumentation).

References

- [1] 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.
- [2] Ken Baclawski. Panoramas and grammars: A new view of data models. Technical Report NU-CCS-91-2, Northeastern University, Feb. 1991.

- [3] Catriel Beeri. Formal models for object-oriented databases. In W. Kim, J. Nicolas, and S. Nishio, editors, *Proceedings of 1st International Conference on Deductive and Object-Oriented Databases*, pages 370–395, Kyoto, Japan, December 1989. Elsevier.
- [4] Paul Bergstein. Object-preserving class transformations. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 299–313, Phoenix, Arizona, 1991. ACM Press. SIGPLAN Notices, Vol. 26, 11 (November).
- [5] Paul Bergstein and Karl Lieberherr. Incremental class dictionary learning and optimization. In *European Conference on Object-Oriented Programming*, pages 377–396, Geneva, Switzerland, 1991. Springer Verlag Lecture Notes 512.
- [6] Wolfgang Bibel. Advanced topics in automated deduction. Technical Report 87-39, Dep. of Computer Science, University of British Columbia, Vancouver, B.C., Canada V6T 1W5, November 1987.
- [7] Paul R. Cohen and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence*, volume 3. William Kaufmann, Inc., 1982.
- [8] Klaus R. Dittrich. Object-oriented database systems: The next miles of the marathon. *Information Systems*, 15(1):161–167, 1990.
- [9] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison Wesley, 1983.
- [10] A.V. Goldberg and K.J. Lieberherr. GEM: A generator of environments for metaprogramming. In *SOFTFAIR II, ACM/IEEE Conference on Software Tools*, pages 86–95, San Francisco, 1985.
- [11] Gaston Gonnet and Frank Wm. Tompa. Mind your grammar: A new approach to modelling text. In *International Conference on Very Large Data Bases*, pages 339–346, Brighton, England, 1987. Morgan Kaufmann.
- [12] 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.
- [13] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [14] Walter L. Hürsch, Karl J. Lieberherr, and Sougata Mukherjea. Object-oriented schema extension and abstraction. In *ACM Computer Science Conference, Symposium on Applied Computing*, Indianapolis, Indiana, February 1993. ACM, ACM Press.
- [15] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

- [16] Christophe Lecluse, Philippe 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.
- [17] Karl Lieberherr and Cun Xiao. Object-Oriented Software Evolution. *IEEE Transactions on Software Engineering*, March 1993.
- [18] Karl Lieberherr and Cun Xiao. Object-Oriented Software Evolution. *IEEE Transactions on Software Engineering*, March 1993.
- [19] Karl Lieberherr, Cun Xiao, and Ignacio Silva-Lepe. Propagation patterns: Graph-based specifications of cooperative behavior. Technical Report NU-CCS-91-14, Northeastern University, September 1991.
- [20] Karl J. Lieberherr. Object-oriented programming with class dictionaries. *Journal on Lisp and Symbolic Computation*, 1(2):185–212, 1988.
- [21] 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.
- [22] Karl J. Lieberherr, Paul Bergstein, and Ignacio Silva-Lepe. From objects to classes: Algorithms for object-oriented design. *Journal of Software Engineering*, 6(4):205–228, July 1991.
- [23] Karl J. Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.
- [24] Karl J. Lieberherr and Ian Holland. Tools for preventive software maintenance. In *Conference on Software Maintenance*, pages 2–13, Miami Beach, Florida, October 16-19, 1989. IEEE Press.
- [25] 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.
- [26] 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.
- [27] 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.

- [28] Karl J. Lieberherr and Cun Xiao. The Demeter Kernel Model for Object-Oriented and Language Design. Technical Report NU-CCS-90-11 (revised), Northeastern University, April 1991.
- [29] 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.
- [30] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.
- [31] Bertrand Meyer. *Object-Oriented Software Construction*. Series in Computer Science. Prentice Hall International, 1988.
- [32] 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.
- [33] Richard Snodgrass. *The interface description language*. Computer Science Press, 1989.
- [34] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.
- [35] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, June 1972.
- [36] T.J. Teorey, D. Yang, and J.P. Fry. A logical design methodology for relational data bases. *ACM Computing Surveys*, 18(2):197–222, June 1986.

Contents

1	Introduction	1
2	Structures for object-oriented design	6
2.1	Structures for defining classes	7
2.2	Object graphs	13
3	Inductiveness of class dictionary graphs	18
4	Programming with class dictionary graphs	24
5	Class dictionary transformations	26
6	Data model axiom checking	30
6.1	Strongly-connected components	31
6.1.1	Correctness	34
6.1.2	Analysis	34
6.2	Useful vertices	35
6.2.1	Correctness and analysis	36
7	Related work	38
8	Conclusions	39

List of Figures

1	Classes and objects	2
2	The relaxed graph	3
3	Students and Courses	4
4	Equivalent class dictionary graph with repetition vertices	5
5	A semi-class dictionary graph	8
6	Forbidden graph	10
7	A semi-class dictionary graph which cannot define objects	11
8	A partial class dictionary graph	12
9	A partial class dictionary graph	13
10	The relations between concepts	14
11	An object of vertex Graduate	16
12	Illegal object	17
13	Illustration of partial class dictionary graphs	19
14	Students and Roommates	20
15	Car and Motor	22
16	Illustration of the Law of Demeter for Classes	23
17	An alternation cycle	24
18	Sorted binary tree	24
19	A flat class dictionary graph	28
20	A violation graph for the Inductiveness Axiom	32
21	SCC_Check	33
22	Useful_Check	37
23	Design and program using the Demeter Method	39

List of Tables