In: ISOTAS96, LNCS 1049, Springer Verlag, 1996

Subtyping by Constraints in Object-Oriented Databases

Wolfgang Kowarschick[1], Gerhard Köstler[2], Werner Kießling[2]

[1] Technische Universität München, Institut für Informatik,
Orleansstr. 34, D-81667 München, Germany
email: kowa@informatik.tu-muenchen.de
[2] Universität Augsburg, Mathematisch-Naturwissenschaftliche Fakultät,
Universitätsstr. 14, D-86135 Augsburg, Germany
email: {koestler,kiessling}@informatik.uni-augsburg.de

Abstract. For many object-oriented database applications taxonomies with a set-inclusion semantics among the type extents are essential. In practical cases, however, common object-oriented modeling techniques often do not result in taxonomies as they ignore application specific constraints. We will elaborate that especially in domains like CAD or similar engineering environments integrity constraints on type attributes have a deep impact on the resulting hierarchy. We argue that subtyping by constraints may be superior to other object-oriented alternatives like subtyping for generalization or nearly-flat hierarchies. Subtyping by constraints achieves a logical set-inclusion hierarchy, and in addition enables a larger amount of semantically correct substitutability. This can even be improved by a novel framework of automatic method adaptation for enhanced substitutability. Moreover, a potential storage penalty caused by making constraints explicit can be avoided completely by applying a new storage optimization technique based on functional integrity constraints. Our results are illustrated by practical examples drawn from the OCAD project.[*]

Keywords. Object-oriented databases and modeling, subtyping, inheritance, constraints, update method adaptation, storage optimization, software reuse.

1 Introduction

Object-oriented database systems (OODBS) exhibit several well-accepted virtues, combining the strengths of databases technology with the power and flexibility of the object-oriented paradigm of programming languages like C++ or Smalltalk. In particular, the rich type system of an OODBS, supporting the natural modeling of complex attributes, is one feature that attracted the attention of database application builders. A crucial notion of the object-oriented paradigm is inheritance. One major interpretation of inheritance, which especially is used in the OODBS setting, can be characterized by the following two constituents ([ABD$^+$89; ZM90; Bud91; CY91; RBP$^+$91; Boo94; Kim95]):

[*] "OCAD: Object-Oriented Databases for CAD" is a joint project of Nemetschek Programmsystem GmbH, Munich (a large CAD tool manufacturer in Europe) and the Bavarian Research Center for Knowledge-Based Systems (FORWISS) under the direction of W. Kießling.

Definition 1.1 (Inheritance in Object-Oriented Systems).

– **Structural Inheritance**
  A subtype in a hierarchy inherits all attributes of its supertypes.

– **Substitutability**
  All methods of a type should be applicable to all instances of its subtypes. □

Some of the benefits of substitutability are software reusability, consistency of interfaces, and rapid prototyping. Therefore the need for overriding methods should be exceptional. The main purpose of overriding should be to provide more efficient implementations (e.g., to enhance a rapid prototype), but not to "repair" semantically incorrect methods.

Software reuse by inheritance is of major concern. But it is conspicuous that in many applications (especially from the areas of CAD, GIS, and engineering) hierarchies resulting from conventional object-oriented analysis and design often require the overriding of methods because inherited implementations do not match their intended semantics. We will argue that this does not happen coincidentally, but as a consequence of attributes that are (often implicitly) related by integrity constraints. In conventional object-oriented modeling approaches such attributes are usually omitted for reasons of simplicity and storage complexity. On the other hand, as the implicit constraints may be encoded in method implementations, these omissions may cause semantical mismatches. To enhance software reuse in hierarchies as described above, we propose to make the implicit attributes and constraints explicit and to organize the types in subtyping-by-constraints hierarchies. We show that such hierarchies have some very desirable properties, both from the object-oriented modeling perspective and for OODBS purposes.

Achieving these advantages requires that the integrity constraints have to be checked at run-time when we execute update methods. This is a consequence of a very general principle concerning hierarchies with integrity constraints: we show that in this kind of hierarchies some overriding or (as a special case) integrity checking is inevitable.

The rest of the paper is organized as follows: In Sect. 2 we illustrate by a case study from the CAD area the drawbacks caused by implicit attributes and constraints. In Sect. 3 we show that in hierarchies with integrity constraints (whether they are explicit or not) method overriding is inevitable, but can be restricted to adorning update methods with integrity checks if we organize the types in subtyping-by-constraints hierarchies. Section 4 presents the automatic adaptation of update methods to minimize these run-time checks and in Sect. 5 we provide a storage optimization algorithm based on the integrity constraints, avoiding storage overhead caused by the introduction of the implicit attributes. In Sect. 6 our approach is compared with related work. Finally, Sect. 7 summarizes our contributions and gives an outlook on future work.

2 A Motivating Case Study

Object-oriented analysis and design techniques ([SM88; Bud91; RBP$^+$91; Boo94]) often result in type hierarchies in such a way that methods inherited from superclasses

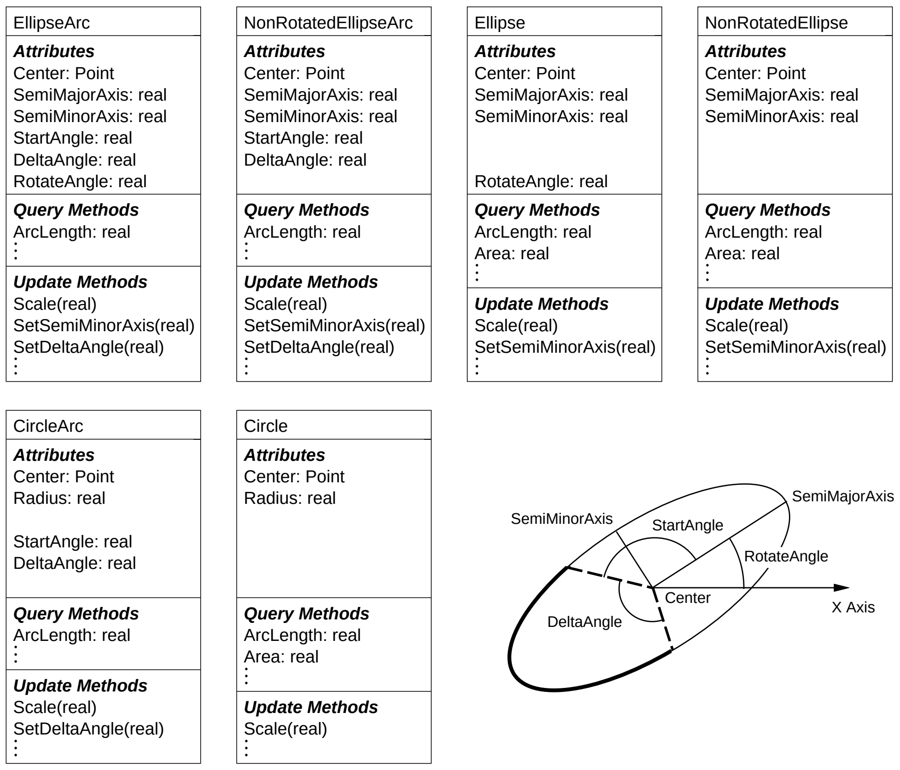

Fig. 1. Some CAD-types

do not conform with their intended semantics. This means, although they are syntactically applicable to instances of the subclasses they do not yield the desired result. We will argue that the intrinsic reason for that semantic mismatch is the disregard of integrity constraints during the design of the hierarchy, although they were *implicitly* employed for the implementation of the methods. In the following we will distinguish *query methods* from *update methods*. Query methods are used to inspect type instances without modifying them—update methods, on the other hand, modify objects without returning any value. This very natural distinction is necessary because in case of subtyping by constraints only update methods are to be overridden in a semantically correct way (see Sect. 3).

Example 2.1 (CAD modeling). Figure 1 shows some types of a realistic CAD application gained by object-oriented analysis and design techniques in the OCAD project. The types comprise ellipse arcs and specializations. Many update methods, query methods, and (graphical) attributes are left out for shortness. ☐

Ordering the types according to the requirement of structural inheritance results in a subtyping-for-generalization hierarchy ([HO88; Bud91]) depicted in Fig. 2. As all inherited methods can syntactically be applied to all instances of the subclasses, this hierarchy apparently satisfies the property of substitutability as well.

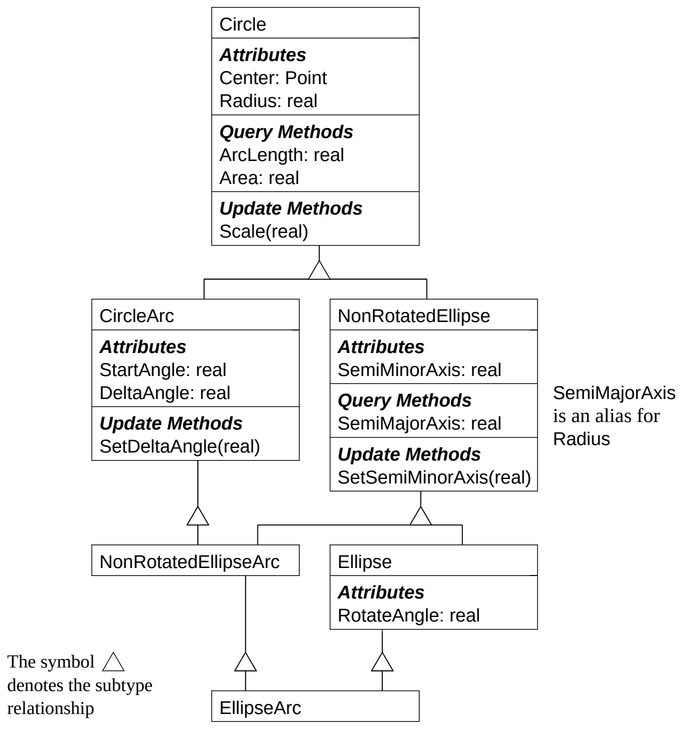

Fig. 2. Subtyping for generalization

Let us examine the two types Circle and CircleArc in more detail. CircleArc inherits the implementation $2 * \pi *$ Radius of ArcLength from Circle. This implementation, however, does not match the intended semantics of ArcLength for CircleArc because it describes the circumference of the circle corresponding to the circle arc. The reason for this semantical mismatch is that not only circle arcs but circles as well do have a delta angle—yet for circles it is by definition always equal to $2 * \pi$. That is, the type Circle has an *implicit* attribute DeltaAngle that is restricted by an *implicit* integrity constraint. Although this constraint was not stated explicitly, it was exploited for the implementation of ArcLength within Circle. For circle arcs, on the other hand, the arc length is depending on the DeltaAngle—thus the inherited implementation of ArcLength has to be overridden.

That means, Fig. 2 is not an appropriate model of the CAD-types. A further disadvantage of this hierarchy is that it does not match the corresponding extensional hierarchy (there are instances of type CircleArc that are not instances of Circle)—a property especially important in the OODBS setting.

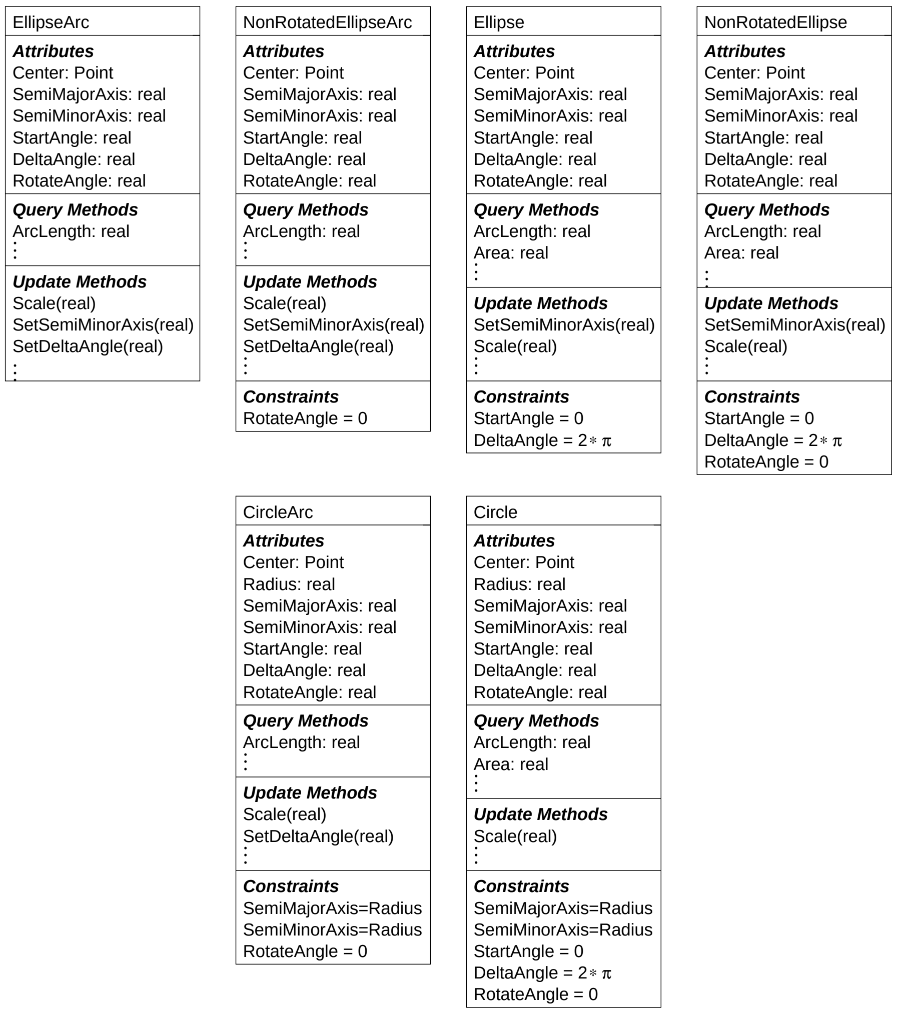

Fig. 3. Types with explicit constraints

To remedy these drawbacks we propose to make implicit attributes explicit and to represent the accompanying integrity constraints. Figure 3 depicts the CAD-types with all relevant attributes and the corresponding integrity constraints stated explicitly. The attributes of all types except EllipseArc are restricted by integrity constraints. In Sect. 5 we show that adding those attributes and constraints does not imply a storage penalty (except of the larger schema information that has to be stored).

To sum up, in many applications a lot of constraints are inherent to the types. Neglecting those constraints enables—as demonstrated above—the inheritance of semantically not justified method implementations. We conclude that such constraints should be an integral part of object-oriented analysis and design. Unfortunately, most design approaches, object-oriented programming languages, and OODBS do not cover integrity constraints sufficiently.

3 Impacts of Constraints on Type Hierarchies

In the last section we argued that implicit attributes and constraints that are inherent for types should be made explicit. In the sequel we will examine the impacts of this demand on the structuring of types into hierarchies. First let us define a relationship on types induced by structural inheritance only (see Def. 1.1).

Definition 3.1 (Candidate Subtype). Let T_1 and T_2 be two types, and let $\mathcal{A}_1$ and $\mathcal{A}_2$ the attributes of these types. If $\mathcal{A}_1 \subseteq \mathcal{A}_2$, then T_2 is called a *candidate subtype* of T_1. □

Let us assume that a type T_1 with attributes A_1 and one of its candidate subtypes T_2 are adorned with integrity constraints. We will show that if T_2 possesses additional constraints concerning the attributes A_1, then, in general, the property of substitutability is violated. In this case designing T_2 as a subtype of T_1 implies that some of the inherited method implementations have to be overridden. In Sect. 4, however, we will show that this overriding can often be accomplished automatically by exploiting the integrity constraints.

Theorem 3.2 (Applicability of inherited methods). *Let the type T_2 be a candidate subtype of the type T_1, let $\mathcal{C}_1$ be the set of integrity constraints on the attributes of T_1, and let $\mathcal{C}_2[T_1]$ be the set of integrity constraints on the attributes of T_2 that restrict some attributes of T_1 (i.e., those constraints in which at least one attribute of T_1 occurs).*

It can be guaranteed that all methods of T_1 can be applied to instances of T_2 in a semantical correct way, if and only if $\mathcal{C}_1 = \mathcal{C}_2[T_1]$.

Proof Sketch. Let us first prove the only-if-direction. We show that if $\mathcal{C}_1 \neq \mathcal{C}_2[T_1]$, then there might be a method of T_1 that is not applicable to all instances of T_2 in a semantically correct way:

> *Case 1.* Assume that $c \in \mathcal{C}_2[T_1] \setminus \mathcal{C}_1$. Then there might be an *update method* of T_1 modifying the attribute values of an instance of T_2 such that c is violated (because that method does not necessarily obey c).
>
> Case 2. Assume that $c \in \mathcal{C}_1 \setminus \mathcal{C}_2[T_1]$. Then there might be a *query* or an *update method* of T_1 whose semantics and/or implementation rely on the constraint c. Consequently this method cannot be applied in a semantically correct way to those instances of T_2 that do not satisfy c.

The if-direction is straightforward: All constraints that might have been exploited for the implementation of a query or update method still hold for T_2, and, moreover, as there are no additional constraints update methods are not obstructed. □

Note that Case 1 of our proof coincides with the claim stated by [ZM90] that subtyping by constraints is not possible if substitutability is required. As an example take the update method SetDeltaAngle that is inherited by Ellipse from EllipseArc (see Fig. 5). If this method is called for ellipses this nearly always will result in a pruned ellipse, i.e., in a violation of the integrity constraint DeltaAngle $= 2 * \pi$. Case 2 is illustrated by the query method ArcLength discussed in Sect. 1 (see Fig. 2).

Corollary 3.3 (Nearly-Flat Hierarchies). *Theorem 3.2 implies that the more constraints there are the flatter a type hierarchy will become, if the inheritance of semantically incorrect methods is to be avoided.* □

Let us demonstrate this corollary by our running example. For the CAD types depicted in Fig. 3 there is only one attribute, viz Center, not involved in any integrity constraint. Thus it is the only one that can be inherited from a superclass (see Fig. 4). [LP91] propose in case of type restrictions, which may be considered to be special constraints, to use such flat hierarchies (called subtype hierarchies by them) in order to avoid update anomalies, i.e., in order to guarantee substitutability. Flat hierarchies, however, are not very satisfying, because they sacrifice the fundamental object-oriented principle of organizing types in hierarchies to avoid redundancy and thus permit only minimal software reuse.

To sum up, we have demonstrated by Theorem 3.2 that organizing types with constraints may in general require the overriding of inherited methods. Thus, how to organize types with constraints? We claim that *subtyping by constraints* should be used to structure such types.

Definition 3.4 (Subtyping by Constraints). A subtype inherits all attributes, methods, and *constraints* from its supertypes. In contrast to methods constraints cannot be overridden (further constraints, of course, may be added). □

Type hierarchies organized by subtyping by constraints exhibit some major advantages:

1. **Structural Inheritance:** They satisfy the requirement of structural inheritance.
2. **Substitutability for Query Methods:** All inherited query methods are applicable in a semantically correct way without any overriding (as Case 2 of the proof of Theorem 3.2 cannot occur).
3. **Restricted Substitutability for Update Methods:** All inherited update methods are applicable in a semantically correct way, if the constraints are checked at runtime (see next section).
4. **Extent Hierarchies:** As physical extent of a type we consider the set of all actually created instances (e.g., the ODMG standard supports types with extents, [Cat94]; SQL3 supports table hierarchies that can be used to store type extents, [MM95]). Since in a subtyping-by-constraints hierarchy a type satisfies all the constraints of its supertypes, an instance of this type can also be regarded as an instance of these supertypes. This justifies the definition of the logical extent of a type as the physical extents of itself and all its (direct and indirect) subtypes. The logical extents of

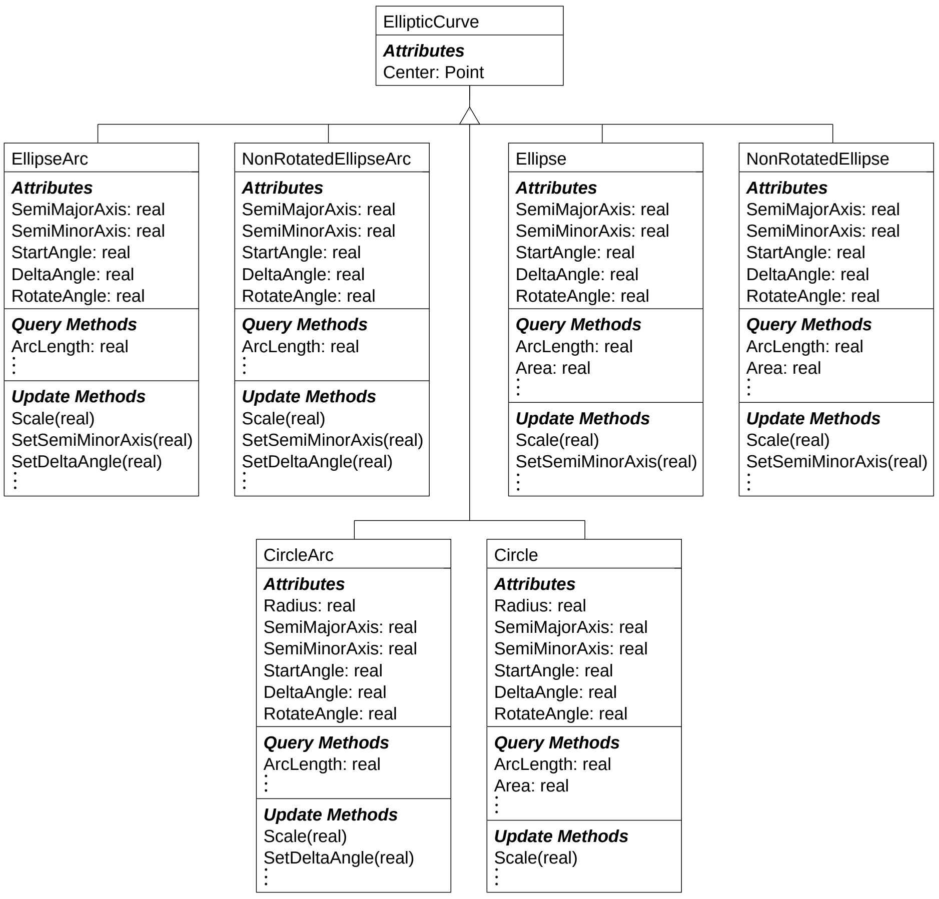

Fig. 4. Almost flat ellipse arc hierarchy

a subtyping-by-constraints hierarchy are ordered by set inclusion, i.e., the logical extent of a type is a subset of the logical extents of all its supertypes. Thus logical extents form a *taxonomy*. Taxonomies are especially desirable in the database setting to facilitate OQL queries and, moreover, coincide with the intuitive meaning of the subtype/supertype relationship.

Applying subtyping by constraints to the types of Fig. 3 yields the hierarchy depicted in Fig. 5.[2] In our opinion this hierarchy is preferable both to the subtyping for generalization hierarchy (see Fig. 2) as well as to the flat hierarchy (see Fig. 4). Its only disadvantage is that the calling of inherited update methods may cause run-time errors. These errors, however, can often be avoided, if such methods are adapted by the algo-

[2] The same hierarchy is achieved, if in a first step all types are arranged in a taxonomy (without paying regard to attributes and methods). Thereafter, in a second step, the types may be adorned by attributes, methods, and constraints.

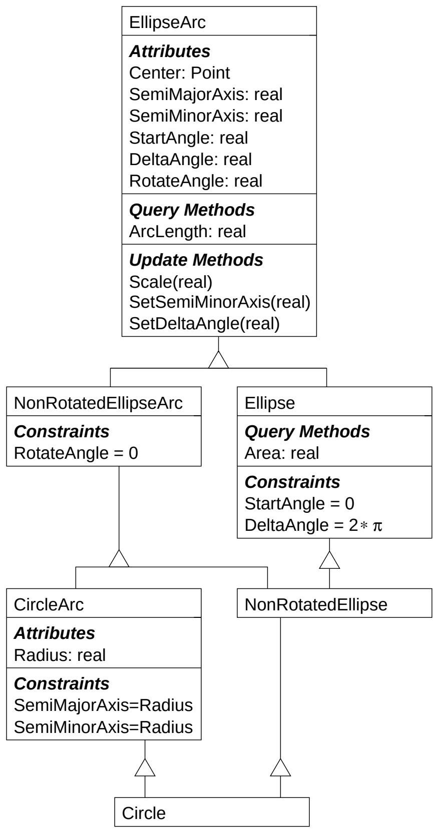

Fig. 5. Subtyping by Constraints

rithms described in the next section. That means, we do neither comply with [MD94], who claim that subtyping by constraints should not be supported by SQL3 at all because this requires run-time checks, nor do we comply with [LP91], who propose to use flat hierarchies in order to avoid update anomalies.

4 Adaptation of Update Methods

We have demonstrated in Sect. 3 that, generally, in case of subtyping by constraints some inherited update methods *must* be adapted, i.e. overridden in order to guarantee that all constraints are always satisfied. Adaptation is only necessary in those cases when a type T_2 inherits an update method u from a type T_1 where u modifies some of the attributes occurring in $\mathcal{C}_2[T_1] \setminus \mathcal{C}_1$ (see Theorem 3.2).

In this paper we suppose, for the sake of simplicity, that an update method only modifies attributes of that object to which it is sent and that update methods do not call other ones. We further assume that updates are performed by update methods only. As a consequence no direct update access to the attributes is possible. Update methods like SetSemiMinorAxis have to be used instead.

Definition 4.1 (Update Methods). Let T be a type with constraints $\mathcal{C}$ on its attributes. Moreover, let X_u be a set of variables and $\mathcal{C}_u \subseteq \mathcal{C}$ the set of those constraints in which some attributes of X_u occur. An *update method* u is defined as follows:

update method $u($<*parameters*>$)$ **begin** <*body*>; check($\mathcal{C}_u$); **end**;

<*body*> modifies the attributes X_u, and the check statement checks all constraints $\mathcal{C}_u$. If check fails, then u raises an exception.[3] □

In the sequel we examine three possibilities to adapt update methods automatically. Note, however, that manual overriding is not excluded. A programmer always can decide to replace an automatically adapted method implementation by another one.

The nop-*Adaptation.* The most trivial adaptation method is to replace the bodies of those inherited update methods that have to obey additional constraints by one statement only: nop. This statement does not have any side effect on the database but may raise a warning without interrupting the program run. That means, methods adapted in this way never raise any run-time error.

The philosophy behind this adaptation technique is that it is better to leave an object unchanged than to call an update method whose implementation is not semantically justified (as it does not take into account all applicable integrity constraints). For instance, within the type Ellipse the implementation of the inherited update method SetDeltaAngle may be overridden by nop. Then a call of this method—which is not useful for ellipses—cannot cause a run-time error any more.

Checking the Additional Constraints. Another rather trivial adaptation method is to insert the additional constraints into the check statement of an inherited update method.

Algorithm 4.2 (Adaptation of Constraint Checking). Let T_2 be a subtype of T_1 that inherits all attributes, methods, and constraints (see Def. 3.4). Moreover, let $\mathcal{C}_1$ and $\mathcal{C}_2[T_1]$ be defined as in Theorem 3.2 and an update method u as in Def. 4.1.

The adaptation is done by adding those constraints of $\mathcal{C}_2[T_1] \setminus \mathcal{C}_1$ to the set $\mathcal{C}_u$ of the check statement of u that contain some attributes of X_u. □

Additional constraints usually cause run-time errors, even if the original method u was designed in such a way that none of the constraints $\mathcal{C}_u$ ever becomes violated. Thus, if algorithm Algorithm 4.2 is used, we have to deal with exceptions. This is the task of exception handlers.

Unfortunately this simple adaptation mechanism often yields methods aborting for almost all inputs. For instance, the update method SetSemiMinorAxis inherited by CircleArc from EllipseArc is overridden as follows:

[3] Currently the issue of efficient integrity checking is an active topic in database research (see e.g. [GSUW94]).

```
update method SetSemiMinorAxis(V: real)
  begin
    SemiMinorAxis := V;
    check(SemiMajorAxis = Radius, SemiMinorAxis = Radius);
  end;
```

That is, an integrity check ensuring that the ellipse remains closed is included. Obviously, the method will only succeed for V = Radius . This behavior absolutely matches the literal semantics of SetSemiMinorAxis. Modifying only one semi axis of a circle arc does not make any sense (unless object migration [LT95] to supertypes is possible, which is not the case for most existing OODBS).

The Deterministic Adaptation. A smarter overriding mechanism should not only modify the check statement by Algorithm 4.2 but also *deterministically* adapt the body of an update method in such a way that the constraints remain satisfied after each call (c.p. [STSW93]) without causing too many run-time errors. In general, a deterministic adaptation is impossible for arbitrary constraints, since this problem is closely related to the view update problem ([GPZ88; LS91]). Fortunately, for many *functional constraints* a deterministic adaptation is possible and can be performed automatically.

Definition 4.3 (Functional Constraints). Let T be a type with attributes $\mathcal{A}$. Moreover, let $A, A_1, \ldots, A_n \in \mathcal{A}$ be different attributes, and let f_A be an n-ary function.

1. An equation *fc* $\equiv A = f_A(A_1, \ldots, A_n)$ is called *functional constraint* (FC). A is called *head* of *fc* (head(*fc*)), $\{A_1, \ldots, A_n\}$ is the *body* of *fc* (body(*fc*)). The function $f_A(A_1, \ldots, A_n)$ is called *FC-function* of *fc* (funct(*fc*)).
2. Let $\mathcal{F}$ be a set of FCs over $\mathcal{A}$, and let $A, B \in \mathcal{A}$ be two attributes. Then A *directly depends on* B, $A \leftarrow_d B$, iff there is an FC $A = f_A(X_1, B, X_2) \in \mathcal{F}$, where $X_1, X_2 \subseteq \mathcal{A}$. The irreflexive transitive closure of this relation is denoted by $\overset{+}{\leftarrow}_d$.
3. A set $\mathcal{F}$ of FCs is named *acyclic* if the corresponding "depends-on" relation $\overset{+}{\leftarrow}_d$ is acyclic. □

Functional constraints are very common; think of equality of attributes or constant attributes (see Fig. 3). Now let us define a set of adaptable update methods based on functional constraints.

Definition 4.4 (Adaptable Update Methods). Let u be an update method within type T, let T have attributes $\mathcal{A}$ and functional constraints $\mathcal{F}$, and let Y_u be the attributes accessed (*read and/or write*) by u.

1. The set *DEP*$(u) \subseteq \mathcal{A}$ of *update-dependent attributes* is defined as *DEP*$(u) := \{A \in \mathcal{A} \setminus Y_u : \exists B \in Y_u : A \overset{+}{\leftarrow}_d B\}$.
2. An attribute $A \in$ *DEP*(u) is called *adaptable*, iff there is a FC $A = f_A(X_A) \in \mathcal{F}$ such that each attribute $B \in X_A$ is either adaptable or an element of Y_u. The set of adaptable attributes is denoted by *ADAPT*(u).

3. u is called *adaptable*, iff $\mathit{DEP}(u) = \mathit{ADAPT}(u)$. □

$\mathit{DEP}(u)$ denotes those attributes not accessed by the body of u that must be modified because of the functional dependencies between them and the attributes Y_u. $\mathit{ADAPT}(u)$ contains those attributes of $\mathit{DEP}(u)$ that can actually be adapted by means of the FCs. Both $\mathit{DEP}(u)$ and $\mathit{ADAPT}(u)$ can be determined in polynomial time.

If u obeys all functional constraints then there is no attribute A that functionally depends on Y_u but is no member of Y_u. In this case $\mathit{DEP}(u)$ is empty, and thus no attribute has to be adapted. That is, attributes of inherited update methods have to be adapted only in Case 1 of Theorem 3.2.

Note that in general there are integrity constraints that are not taken into account by the definition of adaptability: Neither non-functional constraints nor functional constraints $A = f_A(X_A)$, where $A \in Y_u$ are utilized by Def. 4.4. In those cases, in general, only *non-deterministic* adaptation is possible.

More functional constraints imply the adaptability of more attributes. This emphasizes our claim of Sect. 2 that integrity constraints should be made explicit. In particular, apparently redundant inverse constraints [4] often improve the result, since $\mathit{ADAPT}(u)$ contains only head attributes of functional constraints. The generation of inverse constraints may be user-guided or supported by tools (e.g., symbolic mathematical programs for arithmetic constraints). In the sequel for the trivial but frequent case A=B $\in \mathcal{F}$ (where A and B are attributes) we assume that the inverse constraint B=A is always contained in $\mathcal{F}$, too.

Adaptation functions a_{A} are used to compute the correct value of an adaptable attribute A. They are constructed by the following recursive algorithm (with polynomial worst case complexity):

Algorithm 4.5 (Adaptation Functions). Let the update method u, the attributes $\mathcal{A}$, the accessed attributes Y_u, and the functional constraints $\mathcal{F}$ be defined as in Def. 4.4, and let A $\in \mathit{ADAPT}(u)$.

```
function adaptation_function(A)
  begin
    $fc$ := some $fc' \in \mathcal{F}$ where head($fc'$) $= A$ and
                               body($fc'$) $\setminus Y_u \subseteq \mathit{ADAPT}(u)$;
    $a_{\mathsf{A}}$ := funct($fc$);
    for B $\in$ body($fc$) $\setminus Y_u$ do
      begin
        $a_{\mathsf{B}}(Y_u)$ := adaptation_function(B);
        <substitute $a_{\mathsf{B}}$ for variable B in $a_{\mathsf{A}}$>;
      end;
    return $a_{\mathsf{A}}(Y_u)$;
  end;
```
□

Let us illustrate Def. 4.4 and Algorithm 4.5 by an example.

[4] Inverse constraints are functional constraints that are equivalent to other ones but have different head attributes (e.g., Z = X $-$ Y is an inverse constraint of X = Y $+$ Z)

Example 4.6 (Adaptability of Scale*).* Let the update method Scale for type EllipseArc be defined as:

```
update method Scale(V: real)
  begin
    SemiMajorAxis := V $*$ SemiMajorAxis;
    SemiMinorAxis := V $*$ SemiMinorAxis;
    check();
  end;
```

The set Y_u is equal to $\{\text{SemiMajorAxis}, \text{SemiMinorAxis}\}$, and for type CircleArc the set of functional constraints $\mathcal{F}$ consists of SemiMajorAxis = Radius, SemiMinorAxis = Radius, Radius = SemiMajorAxis, Radius = SemiMinorAxis, and RotateAngle = 0. Thus, we get $\mathit{DEP}(u) = \{\text{Radius}\}$ and $\mathit{ADAPT}(u) = \{\text{Radius}\}$. That means, Scale is adaptable in this type. The adaptation function a_{Radius} is due to the **some**-operator used in Algorithm 4.5 non-deterministically either implemented as

$$a_{\text{Radius}}(\text{SemiMajorAxis}, \text{SemiMinorAxis}) = \text{SemiMajorAxis}$$
(by constraint Radius = SemiMajorAxis)

or

$$a_{\text{Radius}}(\text{SemiMajorAxis}, \text{SemiMinorAxis}) = \text{SemiMinorAxis}$$
(by constraint Radius = SemiMinorAxis)

□

Adaptable update methods, such as Scale, can be adapted by the following algorithm.

Algorithm 4.7 (Automatic Adaptation of Update Methods). Let T be a type with attributes $\mathcal{A}$ and constraints $\mathcal{C}$, and let u be an (inherited) adaptable update method of type T that has already been adapted by Algorithm 4.2. Moreover, let $\mathcal{C}_u \subseteq \mathcal{C}$ be the set of constraints checked by u, let $\mathcal{C}_{\mathit{DEP}(u)} \subseteq \mathcal{C}$ be the set of constraints that contain some attributes of $\mathit{DEP}(u)$, and let a_A, for $A \in \mathit{ADAPT}(u)$, be the adaptation functions computed by Algorithm 4.5. Then u defined as

```
update method $u$(<parameters>)
  begin <body>; check($\mathcal{C}_u$); end;
```

is *deterministically* adapted as follows:

```
update method $u$(<parameters>)
  begin <body>; <$A := a_A(Y_u)$, for $A \in \mathit{DEP}(u)$>; check($\mathcal{C}_u \cup \mathcal{C}_{\mathit{DEP}(u)}$); end;
```

The constraints $\mathcal{C}_{\mathit{DEP}(u)}$ have to be checked additionally, as now the attributes $\mathit{DEP}(u)$ are also modified. Note, however, that $\mathcal{C}_{\mathit{DEP}(u)}$ contains many constraints whose satisfaction is always guaranteed due to the adaptation (viz, all constraint that were used by Algorithm 4.5 and the corresponding inverse constraints). It, of course, is not necessary to test those constraints—i.e., they might be deleted from the adapted check statement. □

The update method is rewritten in such a way that all non-accessed attributes depending on the accessed attributes are consistently modified, too.

Example 4.8 (Automatic Adaptation of Scale*).* Let the update method Scale of type EllipseArc be defined as in Example 4.6. Type CircleArc inherits this method. Because Scale is adaptable in this class, it can, e.g., be rewritten in the following way:

```
update method Scale(V: real)
  begin
    SemiMajorAxis := V * SemiMajorAxis;
    SemiMinorAxis := V * SemiMinorAxis;
    Radius := SemiMajorAxis;                /* adaptation of dependent attributes */
    check(SemiMajorAxis = Radius, SemiMinorAxis = Radius,
          RotateAngle = 0);
  end;
```
□

5 Storage Optimization

An apparent advantage of subtyping-for-generalization hierarchies is the profitable storage demand of this model. Let us illustrate this aspect by our running example.

Example 5.1 (Storage Demand for CAD Types). Consider the CAD types and assume the following storage requirements for basic types: real requires 4 bytes, Point 8 bytes, and the OID 8 bytes. Then in the subtyping-for-generalization hierarchy (see Fig. 2) *each single* Circle object (i.e., its OID and attributes) occupies 20 bytes, in contrast to 40 bytes in the subtyping-by-constraints hierarchy (see Fig. 5). □

Many a database designer may decide that the advantageous storage complexity of the subtyping-for-generalization hierarchy outweighs the disadvantages of this model concerning software reuse aspects. A low storage demand may especially be desirable in a distributed OODBS to reduce the net load caused by transferring objects from servers to client applications. In this section, however, we will argue that the same storage demand can be achieved for the subtyping-by-constraints hierarchy by an *automatic storage optimization*.

Storage optimization has always been a concern for OODBS. Here we will present an algorithm performing a storage optimization based on the additional functional constraints a subtype may define. Functional constraints $A = f_A(X_A)$ imply classical functional dependencies $X_A \to A$ between the attributes of a type ([Ull88]). Thus, the basic idea of this algorithm relies on an application of the *relational normalization theory*. The algorithm returns two disjoint sets of attributes, $\mathcal{A}_{\text{store}}$ that are physically stored in the database and $\mathcal{A}_{\text{comp}}$ that are computed from $\mathcal{A}_{\text{store}}$, as well as a set $\mathcal{F}_{\text{comp}}$ of functional constraints determining how to compute $\mathcal{A}_{\text{comp}}$ attributes.

Algorithm 5.2 (Attribute Storage Optimization).

Input: Attributes $\mathcal{A}$ and an *ordered* list of functional constraints $\mathcal{F}$ on the attributes.

1. Compute the sets $\mathcal{A}_{\text{store}} \subseteq \mathcal{A}$ and $\mathcal{F}_{\text{comp}} \subseteq \mathcal{F}$:[5]

```
$\mathcal{A}_{\text{store}}$:= $\mathcal{A}$; $\mathcal{F}_{\text{comp}}$:= $\emptyset$;
for $fc \in \mathcal{F}$ do
    if $\text{head}(fc) \in \mathcal{A}_{\text{store}}$ and $\mathcal{F}_{\text{comp}} \cup \{fc\}$ is acyclic then
      begin
        $\mathcal{A}_{\text{store}} := \mathcal{A}_{\text{store}} \setminus \{\text{head}(fc)\}$;
        $\mathcal{F}_{\text{comp}} := \mathcal{F}_{\text{comp}} \cup \{fc\}$;
      end;
```

2. $\mathcal{A}_{\text{comp}} := \mathcal{A} \setminus \mathcal{A}_{\text{store}}$;

Output: $\mathcal{A}_{\text{store}}$, $\mathcal{A}_{\text{comp}}$, and $\mathcal{F}_{\text{comp}}$ □

The quality of the optimization result depends on two parameters:

1. *Cardinality of* $\mathcal{F}$. Again $\mathcal{F}$ should contain as many inverse constraints as possible, because the computation of $\mathcal{F}_{\text{comp}}$ depends on the heads of functional constraints.

   Let, e.g., $\mathcal{F} = \{\mathsf{A} = \mathsf{B}, \mathsf{A} = \mathsf{C}\}$, then the Algorithm 5.2 yields $\{\mathsf{B}, \mathsf{C}\}$ as stored attributes. Adding the inverse constraints $\{\mathsf{B} = \mathsf{A}, \mathsf{C} = \mathsf{A}\}$ improves the optimization as then only *one* attribute will be stored (see Example 5.4).

2. *Processing ordering of* $\mathcal{F}$. Distinct orderings of the functional constraints $\mathcal{F}$ may result in optimizations of various quality.

   For instance, consider the functional constraint A = compress(B) and its inverse B = uncompress(A) (where A is a BLOB, B a CLOB, and compress a function to compress character strings). Then there are two possibilities: first selecting A = compress(B) results in storing B and computing A being much more storage-extensive than storing A and computing B by uncompress(A). The latter solution is achieved if the algorithm first selects B = uncompress(A).

The possibly large search space of orderings can be reduced by heuristic search—a quite common technique in database optimization. The worst time complexity of the storage optimization algorithm is polynomial, and for each $A \in \mathcal{A}_{\text{comp}}$ there exists a unique *computation function* $c_A(X_A)$, $X_A \subseteq \mathcal{A}_{\text{store}}$, composed of FC-functions of $\mathcal{F}_{\text{comp}}$ ([KKK95]). The computation functions will be used to compute the values of computed attributes from the stored ones. They are constructed by the following recursive algorithm:

[5] Both, $(OID\,\mathcal{A}_{\text{store}})$ as well as $(A\,X_A)$ for $A = f_A(X_A) \in \mathcal{F}_{\text{comp}}$ can be regarded as the decomposed relational schemes of the relation schema formed by the attributes and the OID of the objects of a type.

Algorithm 5.3 (Computation Functions). Let $A \in \mathcal{A}_{\text{comp}}$.

```
function computation_function(A)
  begin
    $fc$ := that $fc' \in \mathcal{F}_{\text{comp}}$ where head($fc'$) = A;
    $c_{\mathsf{A}}$ := funct($fc$);
    for B $\in$ body($fc$) $\cap \mathcal{A}_{\text{comp}}$ do
        begin
          $c_{\mathsf{B}}(\mathcal{A}_{\text{store}})$ := computation_function(B);
          <substitute $c_{\mathsf{B}}$ for variable B in $c_{\mathsf{A}}$>;
        end;
    return $c_{\mathsf{A}}(\mathcal{A}_{\text{store}})$;
  end;
```
□

Note that the recursion terminates because $\mathcal{F}_{\text{comp}}$ is acyclic. Its worst case complexity again is polynomial. We demonstrate the storage optimization performed by Algorithm 5.2 for the CAD types.

Example 5.4 (CAD Types continued). Let us examine the proceeding of the algorithm for type CircleArc in detail. Here we have:

$$\begin{aligned}\mathcal{A} = \{&\mathsf{Center}, \mathsf{SemiMajorAxis}, \mathsf{SemiMinorAxis}, \mathsf{Radius},\\ &\mathsf{StartAngle}, \mathsf{DeltaAngle}, \mathsf{RotateAngle}\},\\ \mathcal{F} = \{&\mathsf{RotateAngle = 0},\\ &\mathsf{SemiMajorAxis = Radius}\ , \mathsf{Radius = SemiMajorAxis},\\ &\mathsf{SemiMinorAxis = Radius}, \mathsf{Radius = SemiMinorAxis}\}\ .\end{aligned}$$

When the elements of $\mathcal{F}$ are ordered as above, then step 1 of Algorithm 5.2 computes $\mathcal{A}_{\text{store}}$ and $\mathcal{F}_{\text{comp}}$ as follows:

0. $\mathcal{A}_{\text{store}} = \{\mathsf{Center}, \mathsf{SemiMajorAxis}, \mathsf{SemiMinorAxis}, \mathsf{Radius}, \mathsf{StartAngle}, \mathsf{DeltaAngle}, \mathsf{RotateAngle}\}$,
   $\mathcal{F}_{\text{comp}} = \emptyset$
1. $\mathcal{A}_{\text{store}} = \{\mathsf{Center}, \mathsf{SemiMajorAxis}, \mathsf{SemiMinorAxis}, \mathsf{Radius}, \mathsf{StartAngle}, \mathsf{DeltaAngle}\}$,
   $\mathcal{F}_{\text{comp}} = \{\mathsf{RotateAngle = 0}\}$
2. $\mathcal{A}_{\text{store}} = \{\mathsf{Center}, \mathsf{SemiMinorAxis}, \mathsf{Radius}, \mathsf{StartAngle}, \mathsf{DeltaAngle}\}$,
   $\mathcal{F}_{\text{comp}} = \{\mathsf{RotateAngle = 0}, \mathsf{SemiMajorAxis = Radius}\ \}$
3. $\mathcal{A}_{\text{store}}$ and $\mathcal{F}_{\text{comp}}$ are not modified since the insertion of Radius = SemiMajorAxis into $\mathcal{F}_{\text{comp}}$ would cause a cycle.
4. $\mathcal{A}_{\text{store}} = \{\mathsf{Center}, \mathsf{Radius}, \mathsf{StartAngle}, \mathsf{DeltaAngle}\}$,
   $\mathcal{F}_{\text{comp}} = \{\mathsf{RotateAngle = 0}, \mathsf{SemiMajorAxis = Radius}, \mathsf{SemiMinorAxis = Radius}\}$
5. $\mathcal{A}_{\text{store}}$ and $\mathcal{F}_{\text{comp}}$ are not modified since the insertion of Radius = SemiMinorAxis into $\mathcal{F}_{\text{comp}}$ would cause a cycle.

In step 2 we get $\mathcal{A}_{\text{comp}} = \{\mathsf{RotateAngle}, \mathsf{SemiMajorAxis}, \mathsf{SemiMinorAxis}\}$. The corresponding computation functions are $c_{\mathsf{RotateAngle}}(\mathcal{A}_{\text{store}}) = 0$, $c_{\mathsf{SemiMajorAxis}}(\mathcal{A}_{\text{store}}) = \mathsf{Radius}$, and $c_{\mathsf{SemiMinorAxis}}(\mathcal{A}_{\text{store}}) = \mathsf{Radius}$.

For the remaining CAD types Algorithm 5.2 yields the following storage optimizations.

EllipseArc**:**
Since $\mathcal{F} = \emptyset$, all attributes are stored.

NonRotatedEllipseArc**:**
RotateAngle is computed by $c_{\mathsf{RotateAngle}}(\mathcal{A}_{\text{store}}) = 0$, the other ones are stored.

Ellipse**:**
StartAngle and DeltaAngle are respectively computed by $c_{\mathsf{StartAngle}}(\mathcal{A}_{\text{store}}) = 0$ and $c_{\mathsf{DeltaAngle}}(\mathcal{A}_{\text{store}}) = 2 * \pi$, the other ones are stored.

NonRotatedEllipse**,** Circle**:**
As there are no additional constraints, the set of *computed* attributes is the *union* of the computed attributes of their supertypes, and the set of *stored* attributes is the *intersection* of the stored attributes of their supertypes.

For type Circle, the attributes Center and Radius are stored requiring 20 bytes of storage. Thus storage optimization resulted in the same storage demand as the subtyping-for-generalization hierarchy. □

In [KKK95] we demonstrate how the resulting storage optimization can be implemented on top of every C++-based OODBS.

6 Related Work

A very good survey on object-oriented analysis and design (OOAD) techniques is given by [MP92]. In this paper the authors inter alia claim that placing classes is a weakness of current OOAD research. Most OOAD methods neglect this topic completely (only four out of fifteen OOAD methods evaluated by [MP92] do mention class placement). The difficulties of class placement in the presence of constraints is a direct consequence of our Corollary 3.3 implying that each placement has some drawbacks.

[LP91] propose to distinguish subtyping hierarchies, implementation hierarchies, and specialization hierarchies (i.e., taxonomies). Subtyping hierarchies support reusability for the user, whereas implementation hierarchies support reusability for the implementor. The specialization hierarchy, on the other hand, is only needed for understanding logical relationships. In case of type restrictions, which may be considered to be special constraints, [LP91] suggest to use nearly flat hierarchies as subtyping hierarchies in order to avoid update anomalies. In our opinion at least subtyping hierarchies and specialization hierarchies should coincide, because it is possible to avoid update anomalies even in taxonomies by update method adaptation (see Sect. 4). All that is needed is to distinguish query from update methods, a not excessive price to pay, indeed.

Case 1 of Theorem 3.2 was already proven by [ZM90]. Using this observation the authors have argued that it is impossible to support the four features *substitutability*, *static type checking*, *mutability* (i.e., the possibility of updating objects), and *specialization via constraints* all together in a single type system although any three of the four features seem to work just fine. They suggest to reduce the ability of static type checking, i.e., to permit run-time checks. This coincides with the adaptation done by our Algorithm 4.2. They also claim that at least three different types of hierarchies (which are similar to those described by [LP91]) should be distinguished. In our opinion, however, such a distinction does cause more confusion than clarify the meaning of the types.

[MD94] discuss the question which one of the four incompatible features described by [ZM90] should be eliminated in the upcoming SQL3 standard. They claim not to support specialization via constraints because the other three features are more important. However, they remark that in SQL3 sets and lists are defined by means of constraints as subtypes of collection, and that it is not clear how this can be handled in SQL3. In our opinion SQL3 (or at least SQL4) should support specialization via constraints by all means. In SQL3 it is possible to define constraints on tables that have to be checked at run-time. So it seems to be no problem to permit run-time constraint checking for types, too. Furthermore, it always is possible to automatically adapt inherited update methods in such a way that no run-time errors occur (see Sect. 4).

Many researchers are of contrary opinion as to whether an object-oriented language should support covariance or contravariance specification (see, e.g., [CCHO89; KA90; Sha94]). The distinction between query and update methods as proposed in this paper entails in case of taxonomic modeling a coexistence of both the contravariance rule for query methods and the covariance rule for update methods: On the one hand, *query methods* that are inherited by subclasses never need be adapted, but their input parameter types may be enlarged (contravariance). The set of possible input parameters of *update methods*, on the other hand, is, in general, restricted by additional constraints. That is, our approach reveals that both the contravariance rule *and* the covariance rule are needed for natural modeling—thus they should not be considered combats but partners with equal rights.

In this paper we have dealt with strict inheritance, i.e., *all* attributes, methods, and constraints are inherited by subtypes. Often, however, partial inheritance is favored (inheritance by default, [MMM93]; as-a inheritance, [MHM95]). By partial inheritance we denote that only some attributes, methods, and/or constraints are inherited. Others may either be overridden or even discarded. This kind of inheritance is important, if there is a difference between types that are currently available and types that are to be developed ([MHM95]). In such cases, however, the resulting hierarchies do not reflect application semantics but are established for code reuse reasons only. Especially in the database setting, however, taxonomies should be the outcome of a modeling process capturing as much semantics as possible. On the other hand, a partial attribute inheritance mechanism may turn out as an elegant implementation of the storage optimized hierarchy. Furthermore, the storage optimization presented in Sect. 5 is applicable to partial inheritance any way.

For reusing predefined types partial inheritance is very important. Nonetheless, it should not be misused as a substitute for subtyping-by-constraints. [MHM95], e.g., state that a square is no rectangle as it must not inherit the methods stretch_x and stretch_y. They propose to use as-a (i.e. partial) inheritance instead. However, it has been common sense for at least two thousand years that each square *is a* rectangle; every *query* method applicable to a rectangle can be applied to squares as well (contravariance!). Only updating must be restricted (covariance!). And there are algorithms to do this restriction automatically in such a way that substitutability is guaranteed (see Sect. 4).

Moreover, it is remarkable that [MHM95] design a square to have the three attributes left, top, and side_length although they assume that a definition of rectangle with four attributes left, right, top,and bottom is already available and shall be reused. Due to those differences they have ("by hand") to prove formally that, e.g., the inherited operation move of rectangle satisfies the specification of square, too. Nevertheless, such a design is very common as it apparently saves space. Using our approach, on the other hand, one would first simply define square as a subtype of rectangle and then add one attribute, side_length, and two constraints, right = left $+$ side_length and top = bottom $+$ side_length. Utilizing these constraints the storage optimization described in Sect. 5 would delete the storage overhead by computing right and top instead of storing them. Thus it is possible to reuse rectangle without much effort and without storage penalty.

7 Summary and Outlook

We have investigated object-oriented modeling issues for application domains where integrity constraints play a crucial role. Common modeling practice often yields subtyping for generalization or nearly flat hierarchies, seriously hampering software reuse. In contrast, we propose to use a modeling technique called subtyping by constraints that relies on the principle of making all integrity constraints involved explicit in the design process. Making constraints explicit produces semantically more desirable hierarchies with enhanced substitutability. Moreover, we have provided an automatic adaptation algorithm to even broaden the amount of substitutable update methods. A potential counterargument against our modeling technique of making all constraints explicit, namely the storage overhead compared to subtyping for generalization, was refuted by a new storage optimization algorithm. We think that our proposed modeling method of subtyping by constraints should apply both to the OOPL and the OODBS environment. In the OODBS setting it is even more important, since subtyping by constraints renders it possible to maintain semantically meaningful extent hierarchies under a set-inclusion semantics (i.e. taxonomies), being especially important for object-oriented declarative query languages (c.p. [KBA91]). We have implemented already a prototype for our methodology, realized by a preprocessor translating into C++/Versant. Currently we experiment with a realistic CAD application and we plan to analyze large type hierarchies emerging from existing object-oriented GIS applications.

Acknowledgments

We would like to thank Felix Dutkowski, Robert Hitzelberger, and Hans Kempe for fruitful discussions during OCAD project and also Jutta Seidel for providing us with

insights into architectural CAD applications.

References

[ABD$^+$89] Malcolm Atkinson, Francois Bancilhon, Klaus Dittrich, David DeWitt, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In *Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD'91)*, Kyoto, 1989.

[Boo94] G. Booch. *Object-Oriented Analysis and Design—with Applications*. Benjamin/Cummings, 2nd edition, 1994.

[Bud91] T. Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley Publishing Company, 1991.

[Cat94] R.G.G. Cattel, editor. *The Object Database Standard: ODMG–93*. Morgan Kaufmann Publishers, San Mateo, California, 1994.

[CCHO89] P.S. Canning, W.R. Cook, W.L. Hill, and W.G. Olthoff. Interface for strongly-typed object-oriented programming. In *Proceedings of the 4th Int. Conf. on Object-Oriented Programming Languages, Systems and Applications*, Portland, October 1989.

[CY91] P. Coad and E. Yourdon. *Object-Oriented Design*. Yourdon Press Computing Series. Prentice Hall, Englewood Cliffs, 1991.

[GPZ88] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Trans. on Database Systems*, 13(4):486–524, 1988.

[GSUW94] A. Gupta, Y. Sagiv, J.D. Ullman, and J. Widom. Constraint checking with partial information. In *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems.*, pages 45–55, Minneapolis, 1994.

[HO88] D. Halbert and P. O'Brien. Using types and inheritance in object-oriented programming. *IEEE Software*, 4(5), September 1988.

[KA90] S. Khoshafian and R. Abnous. *Object Orientation—Concepts, Languages, Databases, User Interfaces*. Wiley, New York, 1990.

[KBA91] Setrag Khoshafian, Roger Blumer, and Razmik Abnous. Inheritance and generalization in intelligent SQL. *Computer Standards and Interfaces*, 13:213–220, 1991.

[Kim95] W. Kim. *Modern Database Systems: The Object Model, Interoperability, and Beyond*. ACM Press, Addison-Wesley, New York, 1995.

[KKK95] W. Kowarschick, G. Köstler, and W. Kießling. Taxonomic modeling in C++ based object-oriented databases is efficient. Technical Report 320, Institut für Mathematik, Universität Augsburg, 1995.
(see http://www.informatik.uni-augsburg.de/info2/literature/Tech-reports/m320.html).

[LP91] W. LaLonde and J. Pugh. Subclassing $\neq$ subtyping $\neq$ is-a. *Journal of Object-Oriented Programming*, 3(5):57–62, January 1991.

[LS91] J.A. Larson and A.S. Sheth. Updating relational views using knowledge at view definition and view update time. *Inf. Sys.*, 16(2):145–168, 1991.

[LT95] T.W. Ling and P.K. Teo. Object migration in ISA hierarchies. In T.W. Ling and J. Masunaga, editors, *Proceedings of the Fourth International Conference on Database Systems for Advanced Applications (DASFAA'95)*, pages 292–299, Singapore, April 1995. World Scientific Publishing.

[MD94] N. Mattos and L.G. DeMichiel. Recent design trade-offs in SQL3. *ACM SIGMOD RECORD*, 23(4):84–89, December 1994.

[MHM95] Richard Mitchel, John Howse, and Ian Maung. As-a: a relationship to support code reuse. *Journal of Object-Oriented Programming*, 8(4), July/August 1995.

[MM95] J. Melton and N. Mattos. Sigmod tutorial: An overview of the emerging third-generation SQL standard. In *ACM SIGMOD International Conference on Management of Data*, 1995.

[MMM93] N.M. Mattos, K. Meyer-Wegner, and B. Mitschang. Grand tour of concepts for object-orientation from a database point of view. *Data and Knowledge Engineering*, 1992/93(9):321–352, 1993.

[MP92] D.E. Monarchi and G.I. Puhr. A research typology for object-oriented analysis and design. *Communications of the ACM*, 35(9):35–47, September 1992.

[RBP$^+$91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Sha94] David L. Shang. Covariant specifications. *ACM SIGPLAN Notices*, 29(12):58–65, Dec. 1994.

[SM88] S. Shlaer and S.J. Mellor. *Object-Oriented System Analysis—Modeling the World in Data*. Yourdon Press Computing Series. Prentice Hall, Englewood Cliffs, 1988.

[STSW93] K.-D. Schewe, B. Thalheim, J. W. Schmidt, and I. Wetzel. Integrity enforcement in object-oriented databases. In *Modelling Database Dynamics*, pages 174–195. Springer, 1993.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge–Base Systems*, volume 1. Computer Science Press, 1988.

[ZM90] S.B. Zdonik and D. Maier. Fundamentals of object-oriented databases. In S.B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 1–32. Morgan Kaufmann Publishers, 1990.