Typesafe Dynamic Classification

Jürgen Schlegelmilch

University of Rostock, Computer Science Department, Database Research Group,
D-18051 Rostock, Germany
schlegel@informatik.uni-rostock.de, WWW:
http://wwwdb.informatik.uni-rostock.de/~schlegel/

Abstract. Object-oriented systems rely on classification of objects as a basic principle. This classification can depend on the type of the object, or its state. Many systems, including almost all object-oriented programming languages, only support classification by type, making classes independent of state changes. Many application domains, however, use taxonomies based on classification by state. Views in database systems can achieve this kind of classification but object-oriented database systems do not accept these views as classes.
The problem with classification by state is the need to reclassify objects after updates, and to maintain the type-safety in the presence of references to reclassified objects: if an object drops out of a class that a reference to it expects, then the reference is left ill-typed. Role models which allow explicit reclassification face the same problem.
For SQL3, classification by state was considered but dropped in favour of mutability, substitutability, and static type checking; all four properties were considered incompatible but are not completely. Our proposal to handle the reclassification problem uses a powerful relationship mechanism instead of simple references. Relationships are multi-directional, thus allowing to find objects related to the reclassified one. We then either remove the link between the objects, or roll back the change that caused the reclassification. We also present an approach with less overhead that employs dynamic type checking. While the first approach allows to use views and role classes in the application schema, the second handles them for local variables in methods. We therefore combine both, which permits us to use view and role classes almost arbitrarily. This enables the important use of views in the schema to help maintaining consistency, as known from relational database systems.
Finally, we discuss the combination of classification by properties, known as subclassing by constraining, and classification by type.

1 Overview

Object-oriented data models (OODMs) describe the application domain using objects, and group these objects into classes. These classes are placed into a class hierarchy to enable polymorphism. From already defined classes, we can derive subclasses in two ways: either by adding new properties (extension), or by placing constraints on existing properties. While extension is offered by almost all object-oriented programming languages (OOPLs), constraining is considered not type-safe and is therefore unsupported.

Object-oriented database systems (OODBSs), on the other hand, offer constraining in the form of selection queries: a set of objects can be defined by selecting those objects of a larger set that meet some constraint. However, current databases do not regard such query results as classes.

The main problem in both OOPLs and OODBSs is the need to reclassify an object after updates to properties constrained by the class definition. If there is a reference to that object that assumes membership in a class that the object just dropped, the reference is ill-typed.

Some database programming languages offer roles that objects can gain or drop dynamically. These can simulate reclassification, but face the same problem, namely handling references to dropped roles.

We show how a relationship mechanism together with dynamic typechecks can be used to avoid invalid references, thus enabling constraining as well as view classes and roles, without compromising type-safety. First, we give an informal introduction in Sect. 2 and review existing solutions in Sect. 3. Then we provide the necessary definitions in Sect. 4. In Sect. 5, we discuss two possible solutions and present our approach as a combination. Section 6 shows how extension and constraining can be combined in the definition of a single class, and Sect. 7 concludes the presentation with a summary.

2 Introduction

Classes[1] group objects with common properties together. These commonalities can be found either in structure and behaviour, called the *type*, or in the properties, called the *state*, of the objects. So, a class has two aspects: a type specification and a condition on the state of its objects. There are correspondingly two ways to derive new classes from existing ones:

Extension defines a class with a subtype, i.e. a type with more properties.
Constraining yields a class with a stronger condition.

In both cases, objects of the new class are substitutable for objects of the base class.

Example 1. Both extension and constraining are natural forms of class definitions: Given a class `PERSON` with attributes `name` and `address`, we can define a class `STUDENT` by extending `PERSON`s with a student ID number and a university. On the other hand, we can derive a class `NEW_YORKER` from `PERSON` by constraining the attribute `address` to contain `'New York'`. □

Similarly, classes could be defined as queries, selecting all objects of superclasses whose state meet certain criteria. The type of such a class is the union

[1] We assume basic familiarity with object-oriented concepts, e.g. from [26]. Most notions like class, type, subclass, derived class etc. are defined formally in Sect. 4.

of the types of the superclasses, while the condition is the conjunction of the query with the conditions of the superclasses. This makes classification automatic, without the need to explicitly reclassify objects after state changes, and helps keeping the database consistent.

Methods are operations defined in classes; they are the only way to manipulate objects. Update methods may change the state of an object, and thus leave it in a state violating the class condition. It must consequently drop membership in that class.[2]

There are two possible approaches to the problem of constraining:

1. Always maintain the class condition, so objects are kept consistent, thus avoiding reclassification. Methods that may leave an object inconsistent must be redefined.
2. Move the object out of the class; this is called demigration.

Example 2. Suppose we have a method `move(new_address: STRING)` in class `PERSON` that updates the attribute `address` according to its argument. In class `NEW_YORKER`, we now have the choice:

1. Redefine `move` so that its argument must contain `'New York'`.
2. Let `move` migrate an object out of class `NEW_YORKER`; it will still be in class `PERSON`.

□

Both approaches have drawbacks:

1. Methods defined in the base class may have to be redefined in the derived class to respect the stronger class condition, thus possibly becoming incompatible with their original definition due to covariance. So, objects of the derived class may no longer be substitutable for objects of the base class.
2. Variables annotated with the derived class may refer to the object. Applying a method may remove the object from the class, so the variable is then either ill-typed, or holding a dangling reference, or must be set to a null value.

Example 3. With two variables of type `PERSON` and `NEW_YORKER`, resp., referring to the same object, the method call `move('Washington')` to the first variable shows the problem:

If method `move` has been redefined in `NEW_YORKER` to not accept this argument, then `NEW_YORKER`s are no longer `PERSON`s because they cannot be used as such; both variables are then not allowed to refer to the same object, thus avoiding the problem.

If alternatively this application of `move` removes the object from `NEW_YORKER`, then the second variable must no longer refer to it, for example by setting it to the null value `void`. To do so, we have to know about all variables annotated with class `NEW_YORKER` that refer to this object. □

[2] In typical OOPLs, methods cannot change the type of objects, so there is no similar problem with extension. Any method can be safely applied to an object without references becoming ill-typed.

A database programming language has to offer a solution to this problem, or must not support class definition by constraining or views.

Widening the scope

OODBSs hold objects for longer periods of time and have to reflect possible type-changes. Some database programming languages therefore offer means to explicitly change the type of an object; this is called *migration*.

Example 4. A person may become a student by enrolling on a university, and eventually ceases to be a student when graduating. So, a method `enrol` should be applicable to objects of class `PERSON`, migrating them into the class `STUDENT`, and likewise a migrator `graduate` for `STUDENT`s to move them out again. If there is then any variable annotated with class `STUDENT` referring to such an object, it is left ill-typed. □

The classes that an object can acquire or drop dynamically are called *roles*, and OODMs supporting this kind of type-change are called *role models*. They include special methods called *migrators* (`enrol` and `graduate` in Example 4) to change the class and type of objects. It is clear that in these models the demigration problem shows up even without constraining. The only difference is that in role models migration is always performed explicitly while with constraining migration is implicit as a side-effect of some updates.

3 Related work

To our knowledge, for constraining the problem was first presented in [26, p. 15]. In this article, four properties of class hierarchies are shown to be incompatible, and dropping one of them is sufficient:

Mutability Immutable objects cannot change their state; this would disallow the method `move` and therefore the implicit migration.

Substitutability Objects cannot be attached to variables declared for superclasses; this would disallow variables annotated with type `PERSON` to refer to `NEW_YORKER`s and avoid the polymorphic application of `move` with inappropriate arguments to that objects.

Static type checking At run-time, we can decide whether the object attached to a variable actually is a `NEW_YORKER` or not, and react accordingly to avoid type errors.

Subclassing by constraining This would disallow the definition of class `NEW_YORKER` by constraining the class `PERSON`.

According to [26, 16], one can only combine three of these properties into a single language. However, we will show in Subsection 5.1 that these properties are sufficiently local to narrow this restriction to single class hierarchies.

Cecil In the object-oriented programming language Cecil, so-called predicate classes [7] are derived from base classes by constraining with a predicate. These predicate classes are quite limited: they must not redefine common methods unless they are either ordered or disjoint, and all predicate classes of a set of base classes must partition these base classes and have the same type. This ensures that objects are in exactly one predicate class and always have the same type and only one implementation for each method. However, it also disallows arbitrary constraining and extension and is therefore not a general solution. The intended usage for predicate classes is to model state-dependent methods of their base classes. Thus, the predicates each describe a partition of the possible states in the base classes. [7] provides a comparison of other approaches that use disjointness and coverage, all of them using explicit declarations instead of predicates and inference.

Eiffel The programming language Eiffel 3 [17] offers class conditions and constraining, but no migration; thus the conditions are invariants. Derived classes may strengthen the class invariants of their base classes and have to redefine methods that may leave objects in inconsistent states. Objects of such classes may then no longer be substitutable for objects of base classes. A set of rules called CAT rules extends the type check to prevent invalid substitutions. This solution leaves the programmer alone with the problem. Even worse, there is no syntactic difference between a derived class, whose objects are substitutable, and other subclasses, and the compiler does not enforce the redefinition of inherited methods when the invariant has been strengthened.

Fibonacci The database programming language Fibonacci [3] offers migrators, but no class conditions. Fibonacci allows objects to migrate into classes with a subtype, but not out of classes; because of substitutability this is type-safe, so variables do not have to be checked. Fibonacci can therefore use a static type check without compromising type-safety. On the other hand, this solution does not help the programmer since it makes modelling the application domain very hard; the demigration problem has simply been avoided.

DOOR, BCOOL The database object model DOOR [25] and the functional object database language BCOOL [15] offer migrators that allow objects to gain and loose types freely. References that became ill-typed due to an object dropping a type are set to a null value. However, this requires to check the whole database for such references and therefore does not scale well.

LOOM The knowledge representation language LOOM [4] offers constraining, migration, mutability, and substitutability; it consequently drops static type checking. It is based on predicate logic and includes a classifier that associates objects with classes, which are called concepts in LOOM. Primitive concepts correspond to normal classes, while defined concepts correspond to classes defined by queries. Methods are not tightly bound to classes; rather, their applicability is defined by predicates called situations, making them more flexible and deferring the class membership test to run-time. The programmer has to invoke the classifier explicitly, and the inference engine ensures that all references are well-typed since type annotations are represented as constraints. Links between objects are modelled using relations, and ill-typed links are removed from the knowledge base. Here, the programmer has all means to model the application domain closely, at the cost of possible run-time errors.

Role models Several role models have proposed solutions:

- In [10], roles are themselves objects that are components of other objects, so application domain objects are represented by hierarchies of implementation objects. Migration is performed by manipulating the internal hierarchy of the object. The underlying language Smalltalk [9] supports only automatic memory management, so role objects are kept alive as long as there are references to them. The owning object may have dropped the role (migrated out of a class) long before. Also, Smalltalk is dynamically typed and can deal with run-time type errors.
- [19] introduces role objects just as in [10] and calls them aspects; however, aspects may hide features of their base object and are therefore not substitutable for them. [19] proposes to disallow the deletion of aspects as long as there are references to them, without discussing an implementation.

Database views Views in OODBSs provide a means to define classes by constraining. However, most approaches [18] do not address the demigration problem but concentrate on issues like positioning of derived classes in the class and type hierarchy, combining constraining with extension, and updatability of objects in derived classes. In [14], views can be used as classes; invalid references are set to a null value like in [15], requiring a full scan over the whole database.

Database standards The ODMG standard [6] does not treat sets of objects as classes, whether defined using OQL or not; instead it uses the term 'class' to mean 'abstract data type implementation'. It does not even use the term 'view' and lacks support for roles, too.

The current draft for the SQL3 standard [1] does not allow subtypes to constrain inherited properties (following the proposal of [16]). However, using the standard means for ensuring referential integrity, one can achieve cascading delete or nullification if a row identifier (value of a `REF` type) becomes invalid because the identified row was dropped from the referenced relation; this way, roles could be simulated. This approach does not work for views since these cannot be referenced, so it cannot support dynamic classification. It also does not handle references from variables in (activation records of) procedures.

Schema evolution While migration is related to schema evolution, it is not the same, nor can we adapt techniques from that area. With schema evolution, classes are changed and their objects are converted to conform to the new class;

this is similar to migration. However, schema evolution is performed explicitly, for all objects of a class at once[3], and often excludes the implementation of methods. This puts the burden on the programmer, without support on the language-level. Schema evolution is not performed while methods are being executed, much unlike general object migration.

4 Definitions

We now define an object model that supports constraining and migration.

4.1 Signatures, types, and their hierarchies

Types are sets of operation signatures, where a signature consists of the method name, the number and types of the arguments, and the result type[4]; migrators (see Sect. 4.2) have an additional fourth component in their signature. We require the method name to be unique within a type as a means of identification. The implementation of a type consists of a sort and a function for each of its signatures; a sort is a set of attributes and its elements are tuples from the cartesian product of the attributes. An element of the type is an element of the sort.

Types form a hierarchy: if a type T supports at least the operations of a type U, it is called a subtype of U ($T \preceq_{type} U$). This depends on a corresponding hierarchy on signatures where subsignatures of a signature s can safely handle argument lists intended for a call to s.

Definition 1 (Signature hierarchy). *Let $S = n_s : s_1 \times \cdots \times s_k \to s_r$ and $T = n_t : t_1 \times \cdots \times t_l \to t_r$ be signatures.*

$$\begin{array}{lll} S \prec_{sig} T \iff & & \\ & n_s = n_t \wedge k = l & (a) \\ & \wedge\ s_r \preceq_{type} t_r & (b) \\ & \wedge\ \forall_{i=1\ldots k}\, t_i \preceq_{type} s_i & (c) \end{array}$$

Thus, the names and number of arguments must be equal (a), the type of the result may vary with the hierarchy (b), and the types of the arguments may vary against the hierarchy (c). This relation on signatures is called contravariance *[5].*

The signature hierarchy allows types to not only add new operation signatures but also change those they have in common with any supertype. Thus, the subtype relationship is defined as follows:

[3] although the migration may technically be performed in a lazy way, i.e. on demand.
[4] Abstract data types also include a set of axioms describing relationships among the signatures; these are not relevant to type checking.

Definition 2 (Type hierarchy). *Let $T = \{t_i | i \in I\}$ and $U = \{u_j | j \in J\}$ be types with signatures t_i and u_j, respectively, for some finite index sets I, J.*

$$T \preceq_{type} U \iff \forall u_j \exists t_i : t_i \preceq_{sig} u_j$$

where $\preceq_{sig}$ already assumes $T \preceq_{type} U$.

Note that a subtype may add new signatures arbitrarily; this is called type *extension.*

4.2 Objects and classes

OODMs are built around the notion of objects: an object is a unique identifier (its identity) and has an associated state. A class C consists of a domain $\mathsf{dom}(C)$ of possible objects, two types $\mathsf{type}(C)$ and $\mathsf{mod}(C) \subseteq \mathsf{type}(C)$, and a condition $\mathsf{cond}(C)$. A partial function $\mathsf{state}_C : \mathsf{dom}(C) \hookrightarrow \mathsf{type}(C)$ maps objects of the domain to their local state; the domain $\{o \in \mathsf{dom}(C) \mid \mathsf{state}_C(o) \neq \perp\}$ of that function is called the extent $\mathsf{ext}(C)$ of class C.

$\mathsf{cond}(C)$ is a term of some predicate logic relating the results of methods that are executed on the state of an object of the class. An element of the type is a valid state for an object of the class if it satisfies the condition, i.e. $\mathsf{cond}(C)(o) = \mathsf{true}\, \forall o \in \mathsf{ext}(C)$ must hold.

The elements in $\mathsf{type}(C)$ are called methods and divided into *selectors* and *modifiers*; $\mathsf{mod}(C)$ is the set of modifiers of class C. The result of a modifier $m \in \mathsf{mod}(C)$ is the new state of an object: $\mathsf{state}_C(o) := m(o, \ldots)$ iff the result type of m is $\mathsf{type}(C)$. Modifiers with a result type $T \neq \mathsf{type}(C)$ are *migrators*: if $T = \emptyset$, then they move objects out of the class (*demigrators*), otherwise their signature must contain as fourth component the target class D, into which they move objects, and $T = \mathsf{type}(D)$ must hold. A class is a *role class* iff it is the target class of some migrator.

Objects are manipulated by sending messages to them. Valid messages cause the implementation associated to a matching signature in the type of the object's class to be executed. This run-time matching is called method lookup or late binding. It allows the type of the first argument, the object, to vary with the type hierarchy, thus in Definition 1:

$$s_1 \preceq_{type} t_1$$

without compromising type-safety with static type checks.

Finally, we note that the changes caused by modifiers are visible only via selectors:

Definition 3 (Modifier for a selector). *A modifier m is called a* modifier for selector s

$$\iff \exists \text{ object } o, \text{values } v_i : s(o) \neq s(m(o, v_i))$$

So, applying m to o causes a visible change in the state of o.

4.3 Subclasses

Objects can be in (the extent of) many classes; the resulting subset hierarchy is called the class hierarchy. Classes have to be placed into this hierarchy by the programmer using a binary relation $\prec_{class}$ among classes:

Definition 4 (subclass). *Let $A, B_i, i \in I$ be some classes. If $A \prec_{class} B_i$ holds, then A is called a* subclass *of each B_i (B_i a* superclass*, or* base class*, of A), and $\mathsf{ext}(A) \subseteq \mathsf{dom}(A) := \bigcap_{i\in I} \mathsf{ext}(B_i)$. Therefore, objects in $\mathsf{ext}(A)$ have $\mathsf{type}(A)$ as well as all $\mathsf{type}(B_i)$; in general, the* global type *of an object is the union of the types of all classes it is in, which is a subtype of the type of any such class:*

$$\mathsf{type}(o) := \bigcup_{D\in\{C\,|\,o\in \mathsf{ext}(C)\}\,\mathsf{type}(D)}$$

A class C is called a direct subclass *of a class E if*

$$C \prec_{class} E \wedge \neg\exists D\colon C \prec_{class} D \prec_{class} E$$

We require that no conflicts occur among the signatures in the global type of an object; conflicts among implementations are resolved according to the class hierarchy (see [22] for details). We call the union of the types of the superclasses of a class its inherited type*.*

After placing a new class into the class hierarchy by inserting tuples into $\prec_{class}$, we can define its local type and condition:

- Specifying a type results in type extension.
- Specifying a condition can make the class a constrained class.

Note that objects of a superclass are not automatically objects of a subclass; they have to be migrated explicitly. Views in object-oriented databases [18] and predicate classes in Cecil [7] define classes where objects of superclasses migrate automatically if they meet the condition; because of substitutability migration into a class is type-safe. We call a class *derived* iff objects migrate into this class implicitly. A derived class is always defined by a query and therefore a constrained class. Demigrators of constrained and derived classes must not take arguments besides the object that has to be demigrated, because they may have to be called implicitly.

We now have to define when a class is considered a constrained class; specifying a condition is necessary but not sufficient. For this, we allow the condition of a class to contain method names from signatures of the types of its superclasses. This can be used to either restrict the corresponding state, or to relate the new local state to that defined in superclasses; these latter conditions involving selectors of the new type are not considered constraining.

In order to specify exactly when the condition of a class C has been strengthened in a subclass, we need to take transitive comparisons into account that may relate features defined in C via features of the subclass. To do so, we need the comparison closure of a predicate.

Definition 5 (comparison closure of a predicate). *A function* θ *is a* comparison operator *iff*

- *its signature is* $\theta : T \times T \to \{\mathsf{true}, \mathsf{false}\}$ *for some type* T,
- *and for all* $a, b, c \in T$ *the following holds (using infix notation):*

$$a\,\theta\,b \wedge b\,\theta\,c \Rightarrow a\,\theta\,c$$
$$a\,\theta\,b \Rightarrow (\neg(b\,\theta\,a) \vee a = b)$$

In abstract data types, these properties can be expressed with axioms.

Let T *be a type and* p *be a predicate. A* permutation *of* p *is a predicate constructed from* p *by applying any number of transformations according to boolean equivalences*[5].

The comparison closure $\mathit{closure}(p)$ *of* p *is defined by an iterative transformation process. Let* p_i *denote the current result of this process, with* $p_0 = p$.

- *For any comparison operator* θ, *if there is a permutation* p' *of* p_i *with a sub-expression* $\alpha = c\,\theta\,d \wedge d\,\theta\,e$ *and there is no permutation of* p_i *with the subexpression* $\beta = c\theta d \wedge d\theta e \wedge c\theta e$, *i.e.* α *extended by the transitive comparison* $c\,\theta\,e$, *then we replace* α *in* p' *by* β *to get* p_{i+1}: $p_{i+1} := p'[\beta/\alpha]$.

This step is repeated until it is no longer applicable, resulting in a predicate $p_k =: \mathit{closure}(p)$.

Lemma 1. *For any predicate* p *with finite denotation,* $\mathit{closure}(p)$ *exists and* $p \equiv \mathit{closure}(p)$ *holds.*

Proof. To proof existence, we show that the iteration stops eventually. The number of subexpressions $a\theta b$ in p is finite since p itself is finite, and so the number of comparison arguments is finite, too. The iteration step adds a new combination of comparison arguments, and since their number is finite, that of the combinations is also finite. Since the iteration step adds only new combinations, there must eventually be a p_l where every possible combination has been added, so the iteration step is not applicable to p_l and the process stops with $\mathsf{closure}(p) = p_l$.

To proof the equivalence, we use induction because $\mathsf{closure}(p)$ is constructed iteratively, starting with $p_0 = p$. For this, we show that $p_{i+1} \equiv p_i$ holds.

The predicate p_{i+1} is constructed starting with a permutation p' of p_i. Since all permutations are built by applying boolean equivalences, $p' \equiv p_i$ holds. Then, p_{i+1} is identical to p' except for the subexpressions α and β, respectively. Since θ is a comparison operator, $\alpha \Rightarrow c\,\theta\,e$ holds, and consequently $\alpha \equiv \beta$. It follows that $p_{i+1} \equiv p' \equiv p_i$ and, since $\equiv$ itself is a comparison operator, $p_{i+1} \equiv p_i$.

Since $p_0 = p \equiv p$ holds, induction proves that $p \equiv p_k = \mathsf{closure}(p)$ holds. $\square$

[5] These are transformations that preserve the truth value, e.g. $a \vee b \equiv b \vee a$, $a \wedge b \equiv b \wedge a$, $\neg\neg a \equiv a$, $(a \vee b) \vee c \equiv a \vee (b \vee c)$, $(a \wedge b) \wedge c \equiv a \wedge (b \wedge c)$, $(a \vee b) \wedge c \equiv (a \wedge c) \vee (b \wedge c)$, etc.

In the next step, we filter out comparisons from predicates that involve features defined in the subclass since these do not constrain inherited features; instead, they restrict the possible results of the new features.

Definition 6 (projection of a predicate). *Let p be a predicate and T a type, i.e. a set of signatures. The* projection *of p on T, denoted by $\pi[T](p)$, is defined as follows:*

1. *Let $p' = Q : \bigwedge_{i=[1:k]} p_i$ be the prenex conjunctive normal form of p, with a sequence Q of quantifiers, and s_0 an empty set of predicates.*
2. *For each p_i, if it contains a method name n for which there is no corresponding signature in T, then $s_i = s_{i-1}$ else $s_i = s_{i-1} \cup \{p_i\}$.*

Then, $\pi[T](p) := Q : \bigwedge_{q \in s_k} q$ and contains only methods described by signatures in T.

Now we have all the tools to define constrained classes. Using the comparison closure, we can find constraints involving only inherited features via transitive operators, and by projection we ignore comparisons with new features which are irrelevant to constraining.

Definition 7 (constrained class). *We call a class E* constrained *iff its condition is strictly stronger on its inherited type than the condition of a superclass C:*

$$\begin{aligned}
&E \text{ constrained} \iff \\
&\exists C, D\colon E \prec_{class} D \preceq_{class} C \\
&\qquad \wedge \neg(\pi[\mathsf{type}(C)](\mathsf{closure}(\mathsf{cond}(D))) \Rightarrow \pi[\mathsf{type}(C)](\mathsf{closure}(\mathsf{cond}(E))))
\end{aligned}$$

In words, E is constrained if it has a superclass D, whose condition is strictly stronger than that of E, when only features of a common superclass C are considered.

We call E directly constrained from D *iff this class D is a direct superclass of E.*

Applying a modifier to an object of a constrained class can leave the object in a state violating the class's condition; it must consequently demigrate from that class and all its subclasses. So, normal modifiers can be demigrators for constrained classes (and migrators for derived classes; see Lemma 2). It is obviously sufficient to check only the conditions of directly constrained classes, in order to find out from which classes an object may have to demigrate.

4.4 Variables

Variables are memory locations holding values which are said to be bound or attached to the variable; we will consider only variables holding objects. Each variable is annotated with a type, and it may only hold objects having that type[6] (see Definition 4 for types of objects). For a state i, let $\mathcal{V}_i$ denote the set

[6] We assume all variables to be *polymorphic*, i.e. objects with different types can be attached to them.

of all variables and $\mathcal{O}_i$ the set of all objects. Basically, the binding of objects to variables is a function $\mu_i : \mathcal{V}_i \to \mathcal{O}_i$ mapping variables to objects that varies over time. Let $\tau(v), v \in \mathcal{V}$, be the type annotation of variable v. Similarly, let $\tau_i(o)$ denote the global type of object o in state i; this may change due to o migrating in and out of classes. The binding μ is type-safe iff

$$\forall i \forall v \in \mathcal{V}_i\colon \tau_i(\mu_i(v)) \preceq_{type} \tau(v)$$

because the subtype supports all operations of the supertype.

4.5 Classes as types

Classes can be used as types in most OOPLs. This has two aspects:

1. Variables annotated with a class may only refer to objects in the extent of the class. In databases, this is called referential integrity; in programming languages, variables bound to objects not in their class are called dangling references.
2. Messages sent to the object attached to a variable are executed against the state of the object[7]. So, the state of objects attached to variables annotated with a class must be of a subtype of the type of that class.

Since subclasses are subsets and have (global) subtypes (both in the non-strict sense), it is type-safe to bind objects of a class to any variable annotated with a superclass; this is called substitutability.

5 Solutions

In Sect. 2 (Example 3), we have seen that constraining is dangerous because an update may require demigration of an object, and most current OOPLs are lacking support for migration. It follows that adding support for object migration allows for constraining. However, the new problem is not easier to solve than the old one. So, before discussing object migration in general in Subsection 5.2, we offer a solution for constraining only.

5.1 Constraining reconsidered

To avoid the problem of constraining, [16] proposes to disallow the definition of subclasses in this way altogether, in favour of mutability, substitutability and static type checking. However, this decision is not appropriate for many application domains. For example, in mathematics all objects are immutable and constraining is common, so we would rather drop mutability. In fact, [16] is too pessimistic: all four properties can be combined into a single language, although not in a single class definition.

Mutability and constraining are mutually exclusive, if we want to retain static type checking and substitutability. However, based on Definition 3 we can push the choice between constraining and mutability into the class definition:

[7] except for copying and assignment, of course.

Lemma 2. *A modifier for a selector s can be a demigrator for any constrained subclass with an invariant involving s.*

Proof. Let C be a class with a selector $s \in (\mathsf{type}(C) \setminus \mathsf{mod}(C)$ and a modifier $m \in \mathsf{mod}(C)$, and D a constrained subclass of C with $\mathsf{cond}(D)$ containing the method name of s.

According to Definition 3, if m is a modifier for s then there exists an object $o \in \mathsf{ext}(C)$ such that $s(o) \neq s(m(o, v_i))$. Since $\mathsf{ext}(D) \subseteq \mathsf{ext}(C)$, this object can be in class D with a state fulfilling $\mathsf{cond}(D)$. Applying m to the object o may cause it to no longer fulfill $\mathsf{cond}(D)$ since $\mathsf{cond}(D)$ uses the selector s and m is a modifier for s. Now, if $\mathsf{cond}(D)(\mathrm{o})$ does not hold, the object must be removed from class D by applying the demigrator. Thus, the application of m on o includes that of the demigrator, making m itself a demigrator for class D. □

Therefore, we have the choice:

– If there is a modifier m for selector s, we disallow the definition of constrained subclasses with invariants involving s since in any such subclass m would be a demigrator.
– If a class is constrained by an invariant involving selector s, we disallow the definition of modifiers for s for the same reasons.

So, a class is either mutable or constrained with respect to selector s.

This policy avoids implicit object migration caused by updates, and makes constraining practicable without excluding mutability, substitutability and static type checking from the whole language. However, it still disallows many class definitions where constraining would be natural.

Example 5. Class `PERSON` is mutable with respect to selector `get_address` because of the modifier `move`. Therefore, we are not allowed to define the class `NEW_YORKER`. This requires the programmer to manually make sure that `PERSON`s are really `NEW_YORKER`s where they should be. □

5.2 Managing object migration

While avoiding migration as described in Subsection 5.1 looks like a solution, it is generally preferable to handle it because of the benefits in modelling power. There are several proposals for role models but none handles the demigration problem satisfactorily (see Sect. 3). We found two ways to cope with demigration:

1. disallow the annotation of variables with constrained classes, so no variable can become ill-typed because of a demigration, or
2. modify ill-typed variables after a demigration, by taking advantage of a mechanism for general relationships.

The first solution is very limiting, but can be made practical with suitable support; this is discussed in Subsection 5.2. The second one offers a general solution, but adds some overhead; we present its details in Subsection 5.2.

5.2.1 No variables of constrained classes The demigration of an object from a class will leave variables annotated with that class ill-typed if they reference this object. If there are no variables annotated with a constrained class, they trivially cannot become ill-typed. This is the approach taken by most object-oriented database systems: even though they support selection views, they do not regard them as classes, and consequently one cannot annotate variables with them.

However, even if we accept these sets of objects as classes, this solution makes constrained classes less useful because there is no way to access their local features. A dynamic type check facility can help here:

Type guards like in Oberon [24] can help to simulate local variables of constrained classes, because they provide a dynamic type test. A type guard controls a block by narrowing the type of a variable in that block: if the object bound to the variable does not conform to the type, the block is skipped. So, if the block is executed, it can safely assume that the object bound to that variable has the required type, which can be that of a constrained class. However, the object may not migrate out of that class within the scope of the block, so updates are not possible, except for the very last statement in the block. Due to late binding, it is hard to predict which update methods can safely be used, and due to aliases, even method applications to objects attached to other variables might really effect the object in question. Thus, the controlled block may only contain calls to selectors, plus an optional last call to a modifier on the constrained variable; note that a controlled block itself counts as a modifier if it ends with a modifier.

Example 6. In Example 3, no variable may be annotated with class NEW_YORKER. If we want to access parts specific to NEW_YORKERs, for example the club they visit, we have to use a type guard:

```
local p: PERSON
with (p as NEW_YORKER)
  do
    -- p has type NEW_YORKER in this block
    print(p.club)
  end
```

□

Note that this restriction is not necessary for role classes that are not constrained because with them demigration is explicit. If you do not call a demigrator, directly or indirectly, then you can annotate local variables with role classes. Since it is possible to determine statically whether a relevant demigrator is in the closure of called methods, we can apply a static check. Because of late binding, we have to consider all implementations of methods in all subclasses when building the closure.

Correctness of the approach We now sketch a proof of the correctness of this approach. In Sect. 4.4, we defined the mapping μ of variables to objects to be type-safe iff

$$\forall i \forall v \in \mathcal{V}_i \colon \tau_i(\mu_i(v)) \preceq_{type} \tau(v)$$

First, we partition $\mathcal{V}_i$ according to the block in which variables are declared. We interpret variables controlled by type guards as new variables in the controlled block and call them *controlled variables*; let $\mathcal{V}_{c,i}$ be the set of controlled variables. Since for $v \in (\mathcal{V}_i \setminus \mathcal{V}_{c,i})$ the type annotation $\tau(v)$ is not a constrained class, the standard attachment rule from Sect. 4.4 ensures type-safety, and we only have to care for controlled variables.

Let $\langle s_1; \cdots; s_n\rangle$ be the body of the controlled block s with controlled variable v_s, and $\langle i_0; \cdots; i_n\rangle$ the corresponding program states with $i_{k-1} \overset{s_k}{\mapsto} i_k$. The type $\tau(v_s)$ may be a constrained class. Because of the type guard, $\tau_{i_0}(\mu(v_s)) \preceq_{type} \tau(v_s)$ holds. Only the last statement s_n may be a modifier application and thus may cause the object $\mu(v_s)$ to lose the type $\tau_{i_{n-1}}(\mu(v_s))=\tau_{i_0}(\mu(v_s))$. However, state s_n is the state right after the controlled block s, and the variable v_s does no longer exist. So, we conclude that $\mu_i(v_s)$ is type-safe for all states i in which v_s exists.

Since we have now examined all variables, the approach is type-safe for all variables and in all program states.

5.2.2 Using a relationship mechanism

After a demigration, some variables may be left ill-typed. To avoid type errors due to such dangling references, we have to either redirect them, or set them to a null value. However, this amounts to browsing the whole database for such references, plus local variables in methods up the call chain. This is clearly undesirable, and should be avoided.

Fortunately, some object-oriented database models offer relationship mechanisms that can handle this task more efficiently. Relationships describe relations between objects that are navigable in all directions, thus allowing to find any object holding a reference to a given one. Several relationship mechanisms have been proposed in the literature [20, 8, 2] including one for the OODB standard ODMG 2.0 [6].

Definition 8 (Relationship). *A* relationship R *consists of a relation schema, a condition, and an exception policy. The relation schema is a set att*(R) *of attributes, some of them of class types (references). The extent* $\mathsf{ext}(R)$ *of a relationship is a relation over the given schema. Each tuple in the relation describes a* link *between the objects in the class-typed attributes. The condition describes valid tuples, with the special case of cardinality constraints. The exception policy specifies the behaviour in case of integrity violations; possible reactions are removal of the offending tuples or abort of the transaction.*

The relationship mechanism introduced here is presented in more detail in [21].

Object-oriented systems prefer an object-centred view on the world, and relationships can support this preference by offering a per-object view on their extent. All objects referenced from an attribute of a relation form a derived class (defined by selection); in this class, we can define *access methods* to select the objects related to a given one, thus giving the illusion of a simple reference. Such methods can be defined for any class of objects referenced from attributes of the relation, allowing navigation in all directions. It is therefore possible to find all links that an object participates in, by simply selecting tuples from relations. These relations are generally much smaller than the set of all objects, making the selection much more efficient than a scan of the whole database. Relationships also allow to react flexibly on integrity violations like the dangling reference problem shown in Sect. 2. Both properties together make implicit migrations harmless: Suppose an object participating in a relationship migrates out of the class that the relationship assumes. This constitutes an integrity violation, and the exception policy of the relationship is automatically checked:

- The standard behaviour is to remove all referencing tuples. This is equivalent to setting referencing variables to a null value, and thus avoids type errors.
- If this is inappropriate, we can take advantage of database semantics and specify to abort the transaction. This rolls back the change that caused the demigration, and is suitable whenever the object must be in that class as long as the link exists.

The suitable policy depends on the application domain:

Example 7. Consider a library in New York and its customers. The library may not want customers to move away if they still have books, so it chooses the abort policy for its relationship `BORROWED_BOOKS` with attributes `the_book: BOOK` and `the_customer: NEW_YORKER`. Each tuple in the extent of `BORROWED_BOOKS` describes who has borrowed which book. If a customer tries to move away from New York and still has a book from the library, the demigration will cause an integrity violation check on `BORROWED_BOOKS` and consequently an abort of the transaction: the customer must not leave New York with the book.

On the other hand, the New York clubs mentioned in Subsection 5.2 will have to let members leave them and therefore choose the removal policy for their relationship `MEMBERSHIP`. If a member migrates out of class `NEW_YORKER`, the integrity check will remove the tuple with the dangling reference. So, the object will simply cease to be a club member. □

Relationships are powerful but introduce some overhead. If the tuples are really stored in a relation, then updates are simple but navigational access may be slower. If the tuples are stored distributed in instance variables in the related objects, then updates require consistent changes in all objects but navigational access is fast. Relationships also require that all participants are objects; to apply this approach to variables in blocks, the latter need to be modelled as objects (like in Smalltalk [9]).

Correctness of the approach According to Sect. 4, we have to show that

$$\forall i \forall v \in \mathcal{V}_i \colon \tau_i(\mu_i(v)) \preceq_{type} \tau(v)$$

where μ is the binding of variables to objects and i ranges over states. In this approach, all relevant variables are attributes in some relationship, and other occurences of references are computed by access methods. So the task is to show that μ is correct for all relationship attributes. The concept of relationships makes this easy, since the detection of inconsistencies is part of the system[8], not the application. Let us assume that the current transaction t started in state i_0 and object o migrates out of class C; let $i_k > i_0$ be the state immediately before the demigration. Further, let rel be the set of all relationships and $rel_{i_k}(o) := \{r \in rel \mid \exists a \in att(r){:}\, \tau(a) = C \wedge \exists t \in \mathsf{ext}(r){:}\, \mu_{i_k}(t(a)) = o\}$ those with a tuple t that contains o in some attribute of type C in state i_k.

It is enough to show that either of the exception policies leaves all relationships $r \in rel_{i_k}(o)$ in a consistent state:

removal: Let $t \in \mathsf{ext}(r)$ be a tuple with $\mu_{i_k}(t(a)) = o$ and $\tau(a) = C$. Due to the demigration, t would be inconsistent in state i_{k+1}, so it is removed from $\mathsf{ext}(r)$. It follows that $\mathsf{ext}(r)$ holds no such tuple in state i_{k+1}, thus the relationship r is consistent in that state[9].

abort: Rolling back transaction t will cause state i_{k+1} to be equal to state i_0. If the relationship r was consistent at the start of the transaction, it will also be consistent in state i_{k+1}. If r was inconsistent in state i_0, it became so in a previous transaction, and we perform the proof with that instead of t; eventually we get a consistent state i_α since the empty database is consistent.

So, we have shown that relationships preserve type-safety under demigration of objects.

5.2.3 The Perfect Mix Type guards and controlled variables can only be used in blocks while relationships are best used only for inter-object links. Since there is no overlap, we can combine both approaches: For inter-object references, relationships must be used, while in blocks we allow constrained classes only for variables controlled by type guards. Only non-controlled local variables in blocks and method arguments[10] may not be annotated with constrained classes, and with role classes only if no demigrator is called, directly or indirectly, within the block.

As a result, we are able to support demigration, whether implicit or explicit, allowing constrained classes as well as general roles and views. There is no restriction on the use of these concepts in the data model.

6 Combining extension and constraining

Defining classes by constraining is uninteresting if these classes cannot have additional local state or methods. It is therefore necessary to check how extension and constraining can be combined. Classes can be extended in three ways:

[8] If an object o migrates out of a class, the system has to check immediately all relationships with an attribute of that class, to see whether they contain o.

[9] at least with respect to typing; a similar proof can be given for other possible inconsistencies.

[10] except for the first one, which is the target object. Late binding ensures type-safety for this argument.

– local state to hold additional information
– new methods to manipulate the new state, or to offer functionality that only applies to objects of the constrained class
– new implementations for inherited methods, typically in the form of additional actions

We will now examine each of these ways.

6.1 Extension by local state

Adding local state in a constrained class is possible, but the corresponding methods can only be accessed if the object is known to be in the class defining them (or a subclass). This is discussed in the next Subsection 6.2. The state itself, i.e. the element of the sort, is directly accessible only in implementations of methods of the class.

Each object of a class is mapped to its local state, as specified by the modifier of this class that was last applied to the object. It follows that constructors and migrators of a class determine the first local state; this is called *initialisation*.

Because objects migrate implicitly into derived classes, there is no way to initialise local state in these classes from arguments. The modifier that caused the migration cannot initialise them because it is defined in a superclass. It follows that the derived class must have a parameter-less constructor; the initial state can thus only be derived from the current state of the object (and related objects). There is no similar requirement for general constrained or role classes.

6.2 Extension by methods and implementations

For new methods, there is no initialisation problem because implementations depend on classes, not on individual objects. But with static type checking, a method can only be applied to objects that are known to be in a class with a type that contains this method. Variables are annotated with types so that only objects with conforming types can be attached to them. With the solution presented in Sect. 5.2, new methods of constrained or role classes can only be accessed via relationship links, inside the scope of a type guard, or, thanks to late binding, in other methods of these classes[11]; new methods of role classes may also be used in blocks with no demigrator.

Adding implementations for inherited methods can be done in two ways, depending on the language support:

conventional languages: redefine inherited methods
event-based languages: associate additional actions to events

Both approaches are described in the following subsections.

[11] Late binding is a type guard for the current object, so methods are controlled blocks with the controlled variable `Current` or `this`.

6.2.1 Redefining methods Conventional OOPLs map method calls to function executions, and let subclasses redefine this mapping by supplying an implementation for an inherited method. With late binding, the new implementation will completely replace those defined in superclasses. It is therefore not possible to simply add an action; the new implementation has to explicitly call the inherited implementation to achieve that effect. This leads to problems with late binding.

Late binding will execute the implementation of the most specific class an object belongs to. With constrained or role classes, objects may be in a set of classes simultaneously, and there is often not a unique highest lower bound for this set—and even if there is, the object does not necessarily belong to that class. [22] presents an algorithm that adds conflict resolution classes to a given class hierarchy to make late binding unambiguous even in the presence of role classes.

All role models have this method lookup problem if they support late binding. However, constrained classes make the situation even worse: the relative position of constrained classes in the class hierarchy is undecidable in general [12, 13] because of the constraining predicates, so we have to assume that constrained classes with the same superclasses are incomparable siblings. We can provide means to place them explicitly in the class hierarchy so the programmer can decide. Once the hierarchy is unambiguous, we can apply the conflict resolution algorithm presented in [22].

6.2.2 The event-based approach Event-based OOPLs can associate several actions, i.e. method executions, in different classes with an event. If an event happens, all associated actions are performed in parallel. If the event is equivalent to a selector, all results have to be combined into a net result using an aggregation function (see [10, 11] for examples of such languages). In constrained classes, an action can be associated with an event already handled in superclasses; this action constitutes an extension of the implementation of the event.

This approach is simple and safe but limited to pure additions of actions. There is no way to modify actions defined in superclasses, or to optimize them by replacing them with algorithms that can take advantage of properties of the subclass. Also, the aggregation of partial results may not be suitable to calculate the net result, but there is no way to process intermediate results by different aggregation functions. On the other hand, this approach is well suited to support constrained classes because it does not require the classes to be explicitly placed in the class hierarchy.

7 Conclusion

Subclassing by constraining and views are important modelling concepts that lack support in current OODMs. Besides giving guidelines how to use constraining safely in OOPL data models, we have shown that role support is sufficient to allow unrestricted constraining; role models are a well accepted concept in

OODMs. Our solution is based on another accepted concept, namely relationships. Using relationship links instead of references helps to find ill-typed variables efficiently and handle invalid links flexibly. With our approach, constrained classes, view classes and role classes can be used arbitrarily in the schema of any application, and with some restrictions also in methods. Finally, we have shown that constraining and extension can be safely combined under acceptable restrictions.

References

[1] Database Language SQL (SQL/Foundation [SQL3]). ISO-ANSI working draft, October 1997.

[2] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. A relationship mechanism for a strongly typed object-oriented database programming language. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 565–575, September 1991.

[3] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. Fibonacci: A programming language for object databases. *The VLDB Journal*, 4(3):403–444, July 1995.

[4] David Bril. LOOM Reference Manual Version 2.0. Technical report, University of Southern California, December 1993.

[5] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[6] R.G.G. Cattell, Douglas Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland, and Drew Wade, editors. *The Object Database Standard, ODMG 2.0.* Morgan Kaufmann, San Mateo, CA, 1997.

[7] Craig Chambers. Predicate Classes. In Oscar Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 268–296, Kaiserslautern, Germany, July 1993. Springer Verlag.

[8] Oscar Diaz and P. M. D. Gray. Semantic-rich User-defined Relationships as a Main Constructor in Object Oriented Databases. In *Conf. on Object-Oriented Databases*, Windermere, July 1990.

[9] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.

[10] Georg Gottlob, Michael Schrefl, and Brigitte Röck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3):268–296, July 1996.

[11] Torsten Hartmann, Gunter Saake, Ralf Jungclaus, Peter Hartel, and Jan Kusch. Revised Version of the Modelling Language TROLL (Version 2.0). Informatik-Bericht 94–03, Technische Universität Braunschweig, 1994.

[12] Andreas Heuer and Peter Sander. Classifying object-oriented query results in a class/type lattice. In *Proceedings of the 3rd Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems, Rostock, MFDBS 91*, volume 495 of *Lecture Notes in Computer Science*, pages 14–28, Berlin, May 1991. Springer-Verlag.

[13] Andreas Heuer and Marc H. Scholl. Principles of object-oriented query languages. In Hans-Jürgen Appelrath, editor, *Proceedings GI-Fachtagung "Datenbanksysteme für Büro, Technik und Wissenschaft", Kaiserslautern*, volume 270 of *Informatik-Fachberichte*, pages 178–197, Berlin, 1991. Springer-Verlag.

[14] Christian Laasch. Deskriptive Sprachen für Objekt-Datenbanken. Master's thesis, Fakultät Informatik, Universität Ulm, May 1994.

[15] Christian Laasch and Marc H. Scholl. A functional object database language. In Catriel Beeri, Atsushi Ohori, and Dennis E. Shasha, editors, *Proceedings of the 4th International Workshop on Database Programming Languages*, Workshops in Computing, pages 136–156. Springer Verlag, August 1993.

[16] Nelson Mattos and Linda G. DeMichiel. Recent design trade-offs in SQL3. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(4):84–89, December 1994.

[17] Bertrand Meyer. *Eiffel: The Language.* Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.

[18] Renate Motschnig-Pitrik. Requirements and Comparison of View Mechanisms for Object-Oriented Databases. *Information Systems*, 21(3):229–252, 1996.

[19] Joel Richardson and Peter Schwarz. Aspects: Extending Objects to Support Multiple, Independent Roles. In *ACM SIGMOD Record*, pages 298–307, May 1991.

[20] James E. Rumbaugh. Relations as Semantic Constructs in an Object-Oriented Language. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, pages 466–481, 1987.

[21] Jürgen Schlegelmilch. An Advanced Relationship Mechanism for Object-Oriented Databases. Technical Report 19/1996, University of Rostock, Computer Science Dept., 1996.

[22] Jürgen Schlegelmilch. Conflict Resolution using Derived Classes. In Dilip Patel, Yuan Sun, and Shushma Patel, editors, *Proceedings of the 3rd International Conference on Object-Oriented Information Systems (OOIS'96), London, UK*, pages 267–279, Berlin, December 1996. Springer Verlag.

[23] Sven Thelemann. Assertion of Consistency Within a Complex Object Database Using a Relationship Construct. In Micheal P. Papazoglou, editor, *Proceedings of the 14th International Conference on Object-Oriented and Entity-Relationship Modeling, Gold Coast, Australia*, volume 1021 of *Lecture Notes in Computer Science*, pages 32–43, Berlin, December 1995. Springer Verlag.

[24] Niklaus Wirth and Martin Reiser. *Programming in Oberon - Steps Beyond Pascal and Modula.* Addison-Wesley, first edition, 1992. source code from the book is available at: ftp://ftp.inf.ethz.ch/pub/software/Oberon/Books/PinOberon/.

[25] Raymond K. Wong, H. Lewis Chau, and Frederick H. Lochovsky. DOOR: A Dynamic Object-Oriented Data Model with Roles. Technical Report HKUST-CS96-12, The Hong Kong University of Science and Technology, Department of Computer Science, Clear Water Bay, Kowloon, Hong Kong, 1996.

[26] Stanley B. Zdonik and David Maier, editors. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.