On Generalization and Overriding in UML 2.0

Fabian Büttner and Martin Gogolla

University of Bremen, Computer Science Department, Database Systems Group

Abstract. In the upcoming Unified Modeling Language specification (UML 2.0), subclassing (i.e., generalization between classes) has a much more precise meaning with respect to overriding than it had in earlier UML versions. Although it is not expressed explicitly, UML 2.0 has a covariant overriding rule for methods, attributes, and associations. In this paper, we first precisely explain how overriding is defined in UML 2.0. We relate the UML approach to the way types are formalized in programming languages and we discuss which consequences arise when implementing UML models in programming languages. Second, weaknesses of the UML 2.0 metamodel and the textual explanations are addressed and solutions, which could be incorporated with minor efforts are proposed. Despite of these weaknesses we generally agree with the UML 2.0 way of overriding and provide supporting arguments for it.

1 Introduction

The Unified Modeling Language (UML) [OMG03b,OMG04] is a de-facto standard for modeling and documenting software systems. Generalization in class diagrams (i.e., subclassing) is one of the key concepts in the object-oriented methodology and in UML: It allows us to express that one class is a specialization of another one. The more special class is described as a set of changes and extensions to the more general class.

However, the concrete interpretation of subclassing in an executable environment was rather undefined with respect to overriding (see [Beu02,Pon02]) in previous UML versions (namely, 1.x). In UML 2.0, along many other major changes, generalization is defined much more precisely than in 1.x.

Although it is never mentioned explicitly on 640 pages, UML 2.0 introduces a *covariant overriding* rule for operations and properties. Hence, a subclass overriding a superclass operation may replace the parameter types by subtypes of them. The same rule applies for attributes (the attribute type can be replaced by a subtype) and associations (the association end types can be replaced by subtypes), which can be redefined in UML 2.0 as well. The UML 2.0 provides a meaning for specialization which is consistent across operations, attributes, and associations.

There has been a never ending discussion about whether covariance is a good meaning for overriding in the areas of programming languages and type theory. There has been no final agreement, but a general conclusion was that subclassing in the presence of covariant overriding cannot be used to define subtyping (for a formal explanation see [AC96], a good overview can be found in [Bru96]). Thus on the one hand, statically type checked programming languages cannot have a sound type system when covariant subclassing is generally allowed. On the other hand, it is argued that many situations in the real world are better modeled covariantly, a good argument for this is made, for example in [Duc02]. A more formal comparison of both concepts which does not advocate either of them can be found in [Cas95].

Commonly, UML models are finally implemented in a statically typed programming languages such as Java, C++, and C#. Most of these programming languages do not permit covariant method overriding for the aforementioned type safety reasons (Eiffel [Mey88] is one of the few exceptions). Hence, there is a gap between subclassing semantics in UML and common OO programming languages, which must be considered during implementation of UML models. Nevertheless, we support the UML 2.0 view of subclassing as we think that the richer expressiveness outweighs the typing problems.

This paper precisely explains how overriding is defined in UML 2.0. We relate the UML approach to the way types are formalized in programming languages and we discuss which consequences arise when implementing UML models in programming languages. Despite of the mentioned typing problems we generally agree with the UML 2.0 way of overriding and provide supporting arguments.

However, we have some concerns regarding overriding and subclassing in the final adopted UML 2.0 specification, which could be corrected with minor efforts: (i) There are two inconsistencies in the metamodel parts which deal with redefinition and subclassing. (ii) The definition of subclassing in UML 2.0 is scattered over a large number of class diagrams, several textual semantics sections, and a couple of additional interdependent OCL operations. Thus understanding how subclassing works in UML is a complex task. Since subclassing is such an important concept in object-oriented analysis and design, an explaining section is definitely missing in order to carry the meaning of UML subclassing to the broader audience. This is especially important in the context of Model Driven Architecture [KWB03,OMG02]. (iii) Different from earlier versions, UML 2.0 uses the term 'subtyping' at various locations where 'subclassing' is intended. Because of the mentioned covariant overriding rule in subclassing, the term 'subtyping' should be used more carefully in the specification.

This paper is structured as follows: Section 2 explains our notions of class, type, subtyping, variance, and subclassing used in this paper and relates subclassing to subtyping. Section 3 shows how subclassing and overriding is handled in UML 2.0. Section 3 also illustrates the consequences of having covariant overriding when implementing object models in statically typed programming languages. In Section 4, we justify the existence of a covariant overriding rule in UML and address the mentioned concerns with regard to the technical realization in the specification. We close the paper with a conclusion in Section 5.

2 Background

In this section we shortly explain our notions for type, subsumption (subtype polymorphism), covariance, and contravariance, following [CW85,AC96]. We relate the notions of subclassing and subtyping. The section is designed to summarize central relevant notions in programming languages and type theory employing minimal formalization overhead. Readers familiar with these notions may skip this section.

2.1 Type, Subsumption, and Variance

In a programming language, a *type* represents a set of values and the operations that are applicable to them. For example the type `Integer` may denote the set of natural numbers $\mathbb{N}$ and the operations 1,+, and $-$. In object-oriented programming, an *object type* represents a set of objects and the messages that can be sent to the objects. Types can be used to form a membership predicate over expressions: If an expression e evaluates to a result of type T we say e has type T, denoted as $e : T$. A strongly typed programming language provides mechanisms to ensure that only appropriate operations are applied to values. For example, `"Hello"-5` would be rejected, because a `String` type typically does not include a `'-'` operation for strings and numbers. In statically typed programming languages like C++, Pascal, Java, C# and many others, expressions are assigned static types by a type checker before actual execution. The type checker guarantees that if an expression has the static type T, its evaluation at runtime will always be a value of type T.

A powerful typing rule which is implemented in nearly all common programming languages and in modeling languages like UML is the *subsumption rule*, also known as *subtype polymorphism*. The subsumption rule states that if an expression e has type S and S is a *subtype* of a type T, denoted as $S \ <: \ T$, then e has also type T.

$$\text{if } e : S \text{ and } S <: T \text{ then } e : T$$

As a consequence, expressions may have more than one type. In order to have a sound type system (i.e., no wrong types can be derived for expressions), only certain types can be related by the subtype relation <:. Typically, subtyping for complex types (i.e., for function types and object types) is derived from simpler types (e.g., for function types, <: is derived from the parameter types and return types of a function). Several sound type systems exist, with varying rules for subtyping, including virtual types, higher-order type systems (generic types) and other more elaborated concepts. The following general considerations hold for these systems as well.

For languages with functions, the $\rightarrow$ type constructor can be used to construct *function types*. For example, Integer$\rightarrow$String denotes the type of a function from Integer to String. As explained in [CW85], for given function types $X \rightarrow Y$ and $Z \rightarrow W$ the subtype relation $X \rightarrow Y <: Z \rightarrow W$ can be defined as follows:

$$X \rightarrow Y <: Z \rightarrow W \ \text{ iff } Z <: X \text{ and } Y <: W$$

For example the function type Real→Integer is a subtype of the function type Integer→Real, assuming Integer <: Real. Because the argument types X and Z are related by <: in the opposite direction as $X \rightarrow Y$ and $Z \rightarrow W$, this subtyping rule for function types is contravariant w.r.t. to the argument type, and covariant w.r.t. the return type. Strictly speaking, we must always specify to which part of a complex type we refer to when using the term variance. Formally, variance is defined as follows: Let $T\{-\}$ denote a type T with some 'hole' (i.e., a missing type expression in it). Let $T\{A\}$ denote the type T when the hole is filled with a type A. $T\{-\}$ is: *covariant* if $A <: B$ implies $T\{A\} <: T\{B\}$ and *contravariant* if $A <: B$ implies $T\{B\} <: T\{A\}$. However, some 'default' references for variance have been established in the literature, such as the parameter type for a function type, so the subtype rule for functions is generally known as the *contravariance rule for functions*.

In object-oriented languages, the most important concept is sending a message to an object (i.e., invoking an operation). Thus in a statically typed programming language, the type checker prevents that inappropriate messages are sent to objects.

We denote an *object type* as follows: $T = [l_1 : T_1, ..., l_n : T_n]$, where l_i are the elements (labels) of T (i.e., methods and fields). In the case of a field l_i, which is actually a method without parameters, T_i is simply another object type (or a basic type). In the case of a method l_i, T_i is a function type. For example, a simple Point type may be modeled as follows:

$$\text{Point} = [x : \text{Integer}, y : \text{Integer}, \text{distanceTo} : \text{Point}\rightarrow\text{Integer}]$$

Obviously, an object type T' for which $T' <: T$ holds must contain the labels $l_1, ..., l_n$ since an object of type T' must understand all methods and field access operations that an object of the supertype T understands. Furthermore, in general we cannot allow that the individual label types $T_1, ..., T_n$ change in T'. Subtyping for object types is sound, if we require $T_i = T_i'$ for $i = 1..n$:

$$[l_1 : T_1', ..., l_{n+m} : T_{n+m}'] <: [l_1 : T_1, ..., l_n : T_n] \text{ if } T_i = T_i',\ i = 1..n$$

Formal proofs for this can be found in [AC96]. The basic idea is as follows: Let o be an object of type T'. If we allowed $T_i' <: T_i$ for some i in the above subsumption rule for object types then we could derive $o.l_i : T_i$. Thus the type checker would accept an assignment $o.l_i := x$ for a value $x : T_i$. But this assignment would be valid only if $T_i <: T_i'$ (contradiction). The other way round, if we allowed $T_i <: T_i'$, a similar contradiction occurs for a selection operation $x : T_i := o.l_i$. Hence, type systems having a general co- or contravariant subtyping rule for object types cannot be sound.

However, in class based languages, where all methods of an object are declared statically (at compile-time), the types for method labels may change in subtypes in a *covariant* way (both, the object type and the method label type become more special).

Thus for $S = [l : X\rightarrow Y]$ and $T = [l : Z\rightarrow W]$, S is a subtype of T ($S <: T$) iff $Z <: X$ and $Y <: W$ and l cannot be updated. Although the type of the label l varies covariantly with the object type, this rule is commonly known as the *contravariance rule for method overriding*, because the parameter type varies contravariantly with respect to the object type. Also common in the literature is the (unsound) *covariance rule for method overriding* which also refers to the parameter types.

2.2 Classes and Subclasses

Classes describe objects with same implementations [Mey97]. A class serves as a generator for objects. It specifies which state (i.e., which attributes) and which behavior (i.e., which methods) objects generated by the class have.

Subclassing is a technique for reusing object descriptions (classes). A new class (the subclass) is described as a set of changes to an existing one (the superclass). The partial order $\preceq$ denotes if a class is a direct or indirect subclass of another class.

There is no general agreement about the exact semantics of subclassing (see e.g., [CHC90,PS92,Bru96]). Common definitions of subclassing in programming languages involve the following mechanisms to describe how a new class is derived from an existing one: (i) *Inheritance*, properties of the superclass become properties of the subclass (often, this is the default), (ii) *Overriding*, properties of the superclass are redefined in the subclass (in typical OO languages, overriding is restricted to methods), (iii) *Extension*, new properties are added to the subclass.

Classes can be used to define types (a class c defines a type $\mathrm{type}(c)$). Then, the subclass relationship can be used to define subtyping. This is done in most common statically typed OO programming languages as follows:

$$\mathrm{type}(s) <: \mathrm{type}(c) \text{ iff } s \preceq c$$

To achieve this behavior, types must be extended and distinguished by names (i.e., $c \neq c'$ implies $\mathrm{type}(c) \neq \mathrm{type}(c')$). Thus, two distinct classes can have completely identical definitions but do not create the same type. This is known as *name subtyping* in the literature and is used, for example, in Java, C++, and C#. Other languages allow distinct classes to create the same type. However, having name subtyping or not has no impact on the variance aspects of overriding when subclassing is subtyping.

While defining subtyping as subclassing has no consequences for inheritance and extension, it restricts the way overriding can be applied in subclassing. Especially, as explained above, method overriding cannot be covariant (i.e., the parameter types of an overriding method cannot be subtypes of the parameter types of the overridden method) as explained above. After having discussed programming languages let us now turn to modeling languages.

3 How UML Handles Subclassing and Overriding

In this section, we explain how subclassing and overriding is handled in the UML 2.0 metamodel using a simple example. We focus on overriding of methods, although the same principles apply to attributes and associations ends.

3.1 The Animals Example

Fig. 1 shows an example of a class diagram containing a generalization relationship. The class Animal generalizes the classes Cow and Rat. Read the other way, we say Cow and Rat are specializations or simply subclasses of Animal.

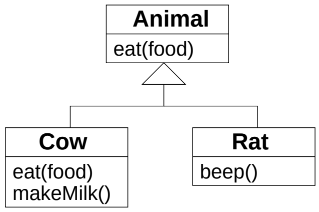

Fig. 1. Example class diagram

As said above, along with generalization comes inheritance and overriding. The *eat(food)* operation defined in Animal is inherited by Rat. Thus, instances of class Rat do not only have the operation *beep()*, but also *eat(food)*. In class Cow, we have repeatedly defined *eat(food)* to indicate that the class provides a new definition of the *eat(food)* operation. In this case, we say Cow overrides, or, in UML 2.0 terms, redefines *eat(food)* from Animal.

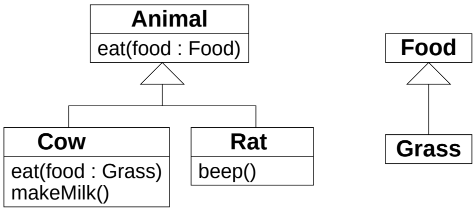

Fig. 2. Example class diagram with parameter types

However, the reader may notice that we have omitted the parameter types in Fig. 1. If we fully specified our operations, the class diagram may look as in Fig. 2. We now have clarified the fact that Cows shall only eat a certain kind of food: Grass. But, is this still overriding? Earlier UML versions left the way operations (methods) override intentionally undefined:

> The way methods override each other is a semantic variation point. [OMG03b, p.2-74].

The upcoming UML 2.0 specification [OMG04] is more precise:

> An operation may be redefined in a specialization of the featured classifier. This redefinition may specialize the types of the formal parameters or return results, add new preconditions or postconditions, add new raised exceptions, or otherwise refine the specification of the operation. [OMG04, p.78]

Thus, in UML 2.0, *Cow::eat(food:Grass)* may override *Animal::eat(food:Food)* if Grass is a specialization (i.e., a subclass) of Food. As we explained in Section 2, this kind of overriding is covariant. The following subsection shows how this restriction is modeled in the UML 2.0 metamodel.

3.2 Relevant Excerpts of the UML 2.0 Metamodel

The metamodel elements relevant for generalization and redefinition are scattered over five class diagrams in the specification. Furthermore, constraints and additional operations are defined separately in the textual part. Thus, the description is distributed over more than 20 (!) locations. The class diagrams in Figs. 3 and 4 combine all relevant aspects. Constraints and additional operations are attached either in-place or as comments. We have given names to the invariants (which do not occur in the UML 2.0 specification) in order to refer to them in the following. Three invariant constraints occur in Figs. 3 and 4. Two of them, RedefinitionContextValid and RedefinedElementsConsistent, belong to the metaclass RedefinableElement. The third, SpecializeValidType, belongs to the metaclass Classifier.

We first look at RedefinitionContextValid. This constraint is straightforward. Its meaning is, that for an element e' which redefines another element e, e' must belong to some subclass of the class in which e is defined. The additional operation *isRedefinitionContextValid(e:RedefinableElement)* must yield true for each redefined element.

The second constraint, RedefinedElementsConsistent, is more subtle and has a lot of impact on the UML 2.0 semantics: Its meaning is that all redefined elements must be consistent with the redefining element. The concrete meaning of 'is consistent' is deferred to subclasses of RedefinableElement, e.g., to Operation. To illustrate this constraint, we consider our animals example from Fig. 2. On the metamodel level, it looks like the object diagram in Fig. 5 (we have omitted the operation *makeMilk()* and the class Rat for simplicity).

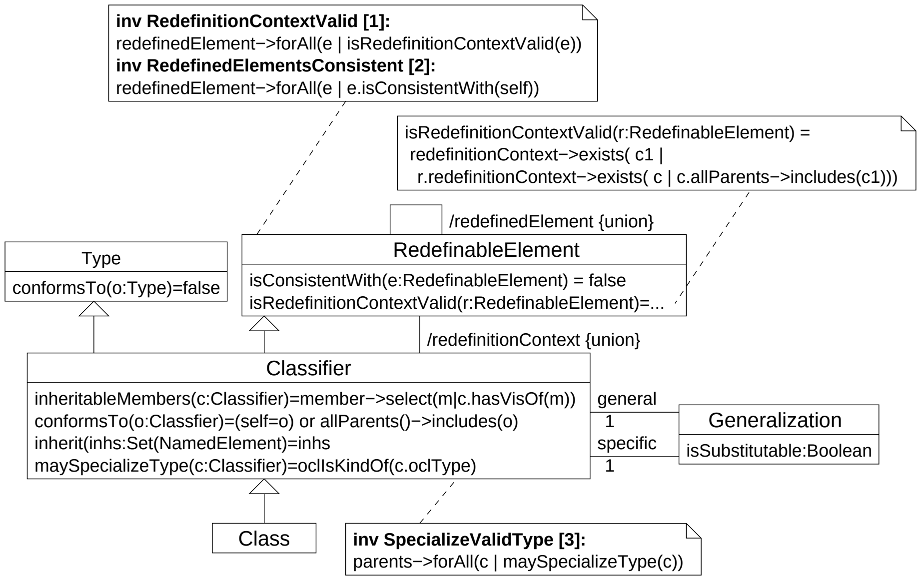

Fig. 3. Condensed UML 2.0 Facts about Generalization and Redefinition - Part A

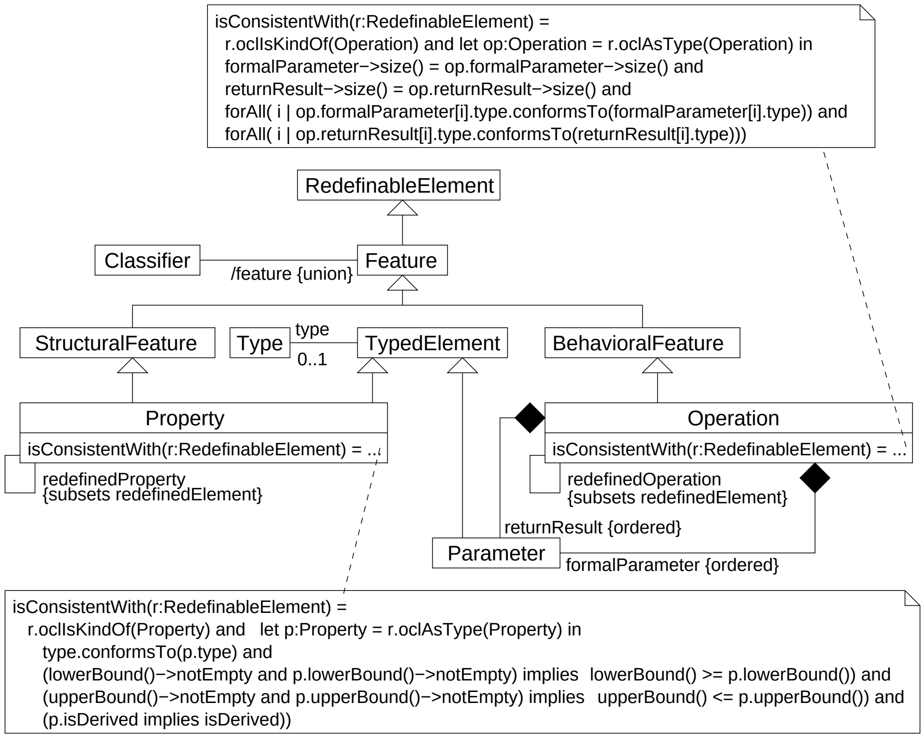

Fig. 4. Condensed UML 2.0 Facts about Generalization and Redefinition - Part B

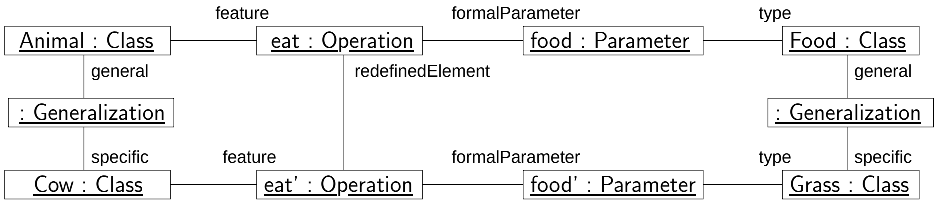

Fig. 5. Example as metamodel object diagram

For this object diagram, the invariant RedefinedElementsConsistent can be written out as follows:

$$\begin{aligned}
&\text{eat}'.\text{redefinedElement}\rightarrow\text{forAll}(\text{e} \mid \text{e.isConsistentWith}(\text{eat}'))\\
=\;&\text{eat.isConsistentWith}(\text{eat}')\\
=\;&\text{eat}'.\text{formalParameter}[1].\text{type.conformsTo}(\text{eat.formalParameter}[1])\\
=\;&\text{food}'.\text{type.conformsTo}(\text{food.type})\\
=\;&\text{Grass.conformsTo}(\text{Food})\\
=\;&(\text{Grass} = \text{Food}) \text{ or Grass.allParents}\rightarrow\text{includes}(\text{Food})
\end{aligned}$$

RedefinedElementsConsistent is responsible for the mentioned covariant overriding rule in UML 2.0. If one operation redefines (i.e., overrides) another, all formal argument and return types of the redefining operation must be specializations of the formal argument and return types of the redefined operation.

Finally, the third invariant SpecializeValidType intends that a classifier may specialize each of its superclasses. We show in Section 4 that the operation *maySpecializeType(c:Classifier)* involved in the constraint depends on the OCL definition of subtyping in an ill-defined way.

3.3 Interpretation of UML Models using Covariant Overriding

Let us consider again our animals class diagram from Fig. 2. The following piece of program (pseudo code) illustrates the problem which arises when *eat* is covariantly overridden in class Cow.

```
declare x : Animal
declare y : Food
x := someAnimal
y := someFood
x.eat(y)
```

If we assume that instances of type Cow can be substituted for instances of type Animal (i.e., by subsumption), then the expression *someAnimal* may evaluate to an object of type Cow and the result of *someAnimal* can be still safely assigned to the variable x. Further, $x.eat(y)$ would be statically type safe, because the static type of x is Animal. Nevertheless, if the expression *someFood* evaluates to an instance which has type Food (i.e., which has not type Grass), the evaluation of $x.eat(y)$ will produce an error because *Cow::eat* is not defined for the parameter type Food. This consideration exemplifies why subclassing cannot be subtyping when the parameter types of a method are covariantly redefined, for the reasons given in Section 2.

However, class based programming languages like Java, C++, or C# have (more or less) sound type systems and prohibit covariant method overriding. Actually, they only support invariant overriding, although covariant overriding would be sound for methods (i.e., parameter types could vary contravariantly and return types could vary covariantly w.r.t. the object type). Newer versions of C++ and Java allow at least covariant redefinition of return types [Str97][BCK+01]. If a class diagram containing covariant overriding is to be translated into such a programming language, the inherent typing problem becomes obvious. A pragmatic solution may look like the one in Fig. 6.

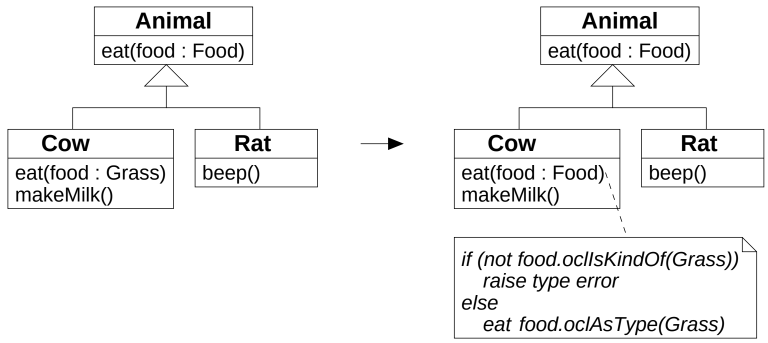

Fig. 6. Transformation to invariant overriding

Although the animals example is now statically type safe, it still produces an error if cows shall eat food which is not grass. We have only deferred the error from compile time to runtime. It is important to see that this (potential) error is *inherent* to the UML class diagram and not a consequence of the implementation. The designer of the class diagram expresses that cows must not eat inappropriate food, for example to avoid mad cow diseases.

Multi-methods [ADL91,BC97,CL95,DCG95] provide an alternative interpretation of covariant specialization. Instead of simply failing a dispatch (i.e., a method call), one of the overridden base class methods may be called instead. Actually, a multi-method based semantics could have been chosen for overriding in UML 2.0. However, we feel that multi-methods are a less intuitive meaning for specialization than 'simple' overriding. Furthermore, multi-method semantics are difficult to realize in common OO languages and, in general, can lead to ambiguities in the context of multiple inheritance.

At least one commonly known OO programming language exists, in which covariant overriding as it is realized in UML 2.0 is directly available: Eiffel [Mey88,Mey97]. Eiffel, aware of the mentioned typing problems, supports covariant overriding as a fundamental design aspect and thus is close to the UML 2.0 understanding of subclassing. Our animals example could be directly implemented in Eiffel. A runtime error would be raised by Eiffel when a cow tries to eat food which is not grass. Although Eiffel is not statically type safe, ongoing work proposes that many runtime type errors could be eliminated by compilers by using more elaborated analysis and type-checking techniques [HBM$^+$03].

4 Concerns with the UML 2.0 Specification

Despite the subclassing resp. subtyping problem explained in the last section, we generally agree with the UML 2.0 way of defining redefinition for operations, attributes, and associations. Why do we agree? Even sound statical type systems such as in Java, cannot guarantee real substitutability for subtypes. A more special object may violate semantic contracts (e.g., invariants) which a more general object fulfills. Type systems guaranteeing real substitutability would require a behavioral notion of subtyping (e.g., the 'Liskov Substitution Principle' [LW94]). However, it seems that very few real world examples [Duc02] exist where objects of one class are generally substitutable for objects of another class. If subclassing must be subtyping when modeling real world aspects, very few subclassing relations can exist. On the other hand, it is a basic desire of designers to model set inclusion by subclassing (i.e., the set of cows is a subset of the set of animals). Hence, although identifying subclassing and subtyping is desirable for programming languages to achieve reasonable statical safe type checking, it is not adequate for intuitive object-oriented modeling on a higher level of abstraction. Therefore, we agree with the covariant way of overriding in UML 2.0.

Nevertheless, we have a couple of concerns with the upcoming specification, which are not fundamental but regard the technical realization of how subclassing and overriding (redefinition) is described.

4.1 Concerns with the Metamodel

We have found two errors in the metamodel parts that model subclassing and overriding. These errors can be fixed with minor efforts.

Ill-defined operation *Classifier::maySpecializeType* Consider the third invariant constraint of the metaclass Classifier (named SpecializeValidType in Fig. 3). For each instance c of Classifier,

$$\mathrm{c.parents} \rightarrow \mathrm{forAll}(\mathrm{c}' | \mathrm{c.maySpecializeType}(\mathrm{c}'))$$

must hold. This can be rewritten using the definition of *maySpecializeType* to

$$\mathrm{c.parents} \rightarrow \mathrm{forAll}(\mathrm{c}' | \mathrm{c.oclIsKindOf}(\mathrm{c}'.\mathrm{oclType}))$$

The built-in OCL operation *o.oclIsKindOf(t)* is defined as follows: 'The oclIsKindOf property determines whether t is either the direct type or one of the supertypes of an object' [OMG03a]. Since subtyping in OCL requires subclassing, the definition of *maySpecializeType* is circular: c may subclass c' only if c is a subtype of c' and if c is a subtype of c' then c must be a subclass of c'. Hence, *maySpecializeType* is not well-defined.

As a proposal for solving this problems we argue for simply omitting SpecializeValidType from the UML 2.0 specification, as it does not make any further restrictions to UML models.

Operation *Property::isConsistentWith* contradicts textual semantics Another inconsistency can be found in *Property::isConsistentWith*. Consider the example in Fig. 7. It is similar to the previous one, except that animals and cows now have a 'food' attribute (i.e., a property) instead of an 'eat' operation. The (textual) specification states, in the covariant specialization manner, that an attribute type must be redefined by the same or a more specific type. However, if we write out the operation *isConsistentWith* we obtain

$$\begin{aligned}
&\text{food}'.\text{redefinedElement}\rightarrow\text{forAll}(\text{e} \mid \text{e.isConsistentWith}(\text{food}'))\\
=\ &\text{food.isConsistentWith}(\text{food}')\\
=\ &\text{food.type.conformsTo}(\text{food}'.\text{type})\\
=\ &\text{Food.conformsTo(Grass)}\\
=\ &(\text{Food} = \text{Grass}) \text{ or Food.allParents}\rightarrow\text{includes(Grass)}
\end{aligned}$$

which obviously yields false. If we used this definition of *maySpecializeType* for Property, we gained a *contravariant* overriding rule for properties, which is not intended according to the explaining text and which does not match the general idea behind redefinition in UML 2.0.

Apparently, the subterm type.conformsTo(p.type) must be flipped (i.e., be rewritten to p.type.conformsTo(self.type)) to achieve the intended covariant overriding rule for properties.

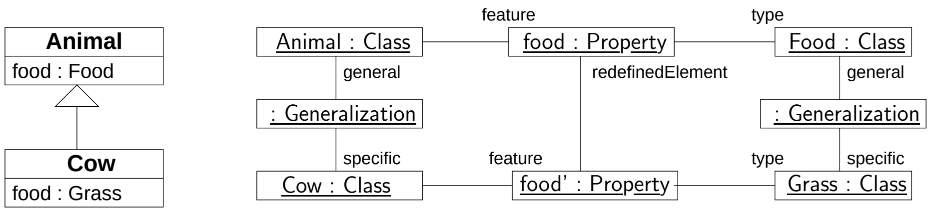

Fig. 7. Example with Properties

4.2 Concerns with the Textual Part of the Specification

Beside corrections to the UML metamodel we suggest some further modifications of the textual description ('Semantics') in the specification.

Provide more explanations and examples The descriptions of Classifier, Generalization, and RedefinableElement should be extended to make it more clear how subclassing is defined in UML. Especially, some explanations how the covariant overriding semantics of UML 2.0 may be carried over to typical statically typed programming languages like Java, C++ and C# should definitely be added. A simple example like our animals example would be nice to help those software developers which are no language lawyers. This also helps in understanding why UML models using covariant overriding may not be translated directly into most implementation languages. The textual semantics should be adjusted to make it clear that subclassing does not produce subtyping in the common sense in all cases.

Avoid using subtyping and subclassing synonymously The term 'subtyping' only occurred once in the 'Foundation' part of the earlier UML specifications. The neutral terms parent and child, with the transitive closures ancestor and descendant, are the preferred terms in that document [OMG03b, p.2-72]. In the UML 2.0 specification, 'subtyping' is heavily used all over the 'Classes' part (the term 'subtype' occurs 26 times in the text). Especially, it is often used where 'subclassing' is meant. There is no need to employ subtyping in most cases, 'subtype' can be safely replaced by 'subclass'.

5 Conclusion

As we have pointed out in the previous sections, UML subclassing cannot be used to define safe subtyping as it is defined in common sense type theory [CW85,AC96], because it allows subclasses to covariantly override superclass operations and attributes. Despite of this fact, we advocate the UML way, because it is more adequate for abstract modeling of real world domains - which are often covariant: cows should really *not* eat all kinds of food (remind BSE). In fact, we have to make a compromise between full substitutability of superclass objects with subclass objects on the one hand (which is not even achieved by most programming languages), and an intuitive way of modeling inclusion of values (e.g., animals include dogs) and grouping of similar concepts.

Nevertheless, the way the new semantics for subclassing and overriding is defined in the UML 2.0 specification is highly complex, scattered over many diagrams, constraints and additional operations sections and is very difficult to understand. The specification should definitely be extended by a section which explains how overriding is handled in UML in a way which even developers who are no language lawyers can understand. Especially, a simple example should illustrate the issues which arise when covariantly specialized models are translated into a statically typed programming language. The specification should make it clear, that it depends on the modeler if her subclassing transforms into subtyping. The UML itself should avoid using the term 'subtyping' where 'subclassing' is much more adequate. This was the case in earlier versions and should be readopted in the final 2.0 specification.

Apart from these general considerations we have proposed two improvements in the important 'Classes' section of the metamodel which could be realized with minor efforts.

Thanks to the anonymous referees for their helpful comments.

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, New York, 1996.

[ADL91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static type checking of multi-methods. In Andreas Paepcke, editor, *OOPSLA '91 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 113–128. ACM Press, 1991.

[BC97] John Boyland and Giuseppe Castagna. Parasitic methods: An implementation of multi-methods for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*, volume 32, 10 of *ACM SIGPLAN Notices*, pages 66–76, New York, October 5–9 1997. ACM Press.

[BCK+01] Gilad Bracha, Norman Cohen, Christian Kemper, Steve Marx, Martin Odersky, Sven-Eric Panitz, David Stoutamire, Kresten Thorup, and Philip Wadler. Add generic types to the java programming language. participant draft specification, 2001. Java Specification Request #14, part of the Java 1.5 specification. `http://jcp.org/en/jsr/detail?id=14`.

[Beu02] Antoine Beugnard. Is MDA achievable without a proper definition of late-binding? In Jean Bezivin and Robert France, editors, *Workshop in Software Model Engineering*, October 2002.

[Bru96] Kim B. Bruce. Typing in object-oriented languages: Achieving expressibility and safety(!), 1996. `http://citeseer.nj.nec.com/bruce96typing.html`.

[Cas95] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135. ACM SIGACT and SIGPLAN, ACM Press, 1990.

[CL95] Craig Chambers and Gary T. Leavens. Typechecking and modules for multimethods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, November 1995.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[DCG95] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 93–102, 1995.

[Duc02] R. Ducournau. "Real World" as an argument for covariant specialization in programming and modeling. In J.-M. Bruel and Z. Bellahsène, editors, *Advances in Object-Oriented Information Systems, OOIS'02 Workshops Proc.*, volume 2426 of *Lecture Notes in Computer Science*, pages 3–12. Springer-Verlag, 2002.

[HBM+03] Mark Howard, Eric Bezault, Bertrand Meyer, Dominique Colnet, Emmanuel Stapf, Karine Arnout, and Markus Keller. Type-safe covariance: Competent compilers can catch all catcalls. Technical report, Swiss Federal Institute for Technology Zürich, April 2003. `http://citeseer.nj.nec.com/howard03typesafe.html`.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained. The Practice and Promise of the Model Driven Architecture*. Addison-Wesley, 2003.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[Mey88] Bertrand Meyer. Eiffel: A language and environment for software engineering. *The Journal of Systems and Software*, 8(3):199–246, June 1988.

[Mey97] B. Meyer. *Object-Oriented Software Construction, Second Edition*. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, 1997.

[OMG02] MOF 2.0 query / views / transformations rfp. Technical report, Object Management Group, 2002.

[OMG03a] UML 2.0 OCL second revised submission. Technical report, Object Management Group, 2003. `http://www.omg.org/cgi-bin/doc?ad/03-01-07`.

[OMG03b] Unified Modeling Language 1.5 Specification. Technical report, Object Management Group, 2003.

[OMG04] UML 2.0 superstructure final adopted specification. Technical report, Object Management Group, 2004. `http://www.omg.org/cgi-bin/doc?ptc/2003-09-15`.

[Pon02] Claudia Pons. Generalization relation in UML model elements. In *Inheritance Workshop at European Conference for Object-Oriented Programming (ECOOP)*, 2002.

[PS92] Jens Palsberg and Michael I. Schwartzbach. Three discussions on object-oriented typing. *ACM OOPS Messenger*, 3(2):31–38, 1992.

[Str97] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, 1997.