Contrasting Classification with Generalisation

Thomas Kühne

School of Mathematics, Statistics and Computer Science
Victoria University of Wellington
PO Box 600, Wellington, New Zealand,
Email: Thomas.Kuehne@mcs.vuw.ac.nz

Abstract

Classification and Generalisation are two of the most important abstraction mechanisms in modelling, and while they share a number of similarities, they are unmistakably different with respect to their properties. Recently, a number of (meta-) modelling language design approaches de-emphasised the differences between classification and generalisation in order to gain various advantages. This paper aims to demonstrate the loss in precision and the loss of sanity checks such approaches entail. After a careful comparison between classification and generalisation, I identify problems associated with the above mentioned approaches and offer alternatives that retain a strong distinction between classification and generalisation.

Keywords: classification, generalisation, strict metamodelling

1 Introduction

The main purpose of modelling is to reduce complexity. Sometimes it suffices to simply reduce the information per element in the universe of discourse but otherwise retain a one-to-one correspondence between these elements and model elements (i.e., creating a token model (Kühne (2006))). However, often there is a need to use more abstract views, in particular, to disregard particularities of individual elements and only capture the relevant universal properties, creating a many-to-one correspondence between elements from the universe of discourse and modeling elements. One thus obtains a way to characterise many individuals by referring to one representative only. In particular, classification is used to create types from instances, giving rise to type models (Kühne (2006)), abstracting away from the identity of instances and their different property values. On the other hand, generalisation is used to create a *genus* (hypernym) from *species* (hyponyms) (Rayside & Campbell (2000)), i.e., to create supertypes from subtypes, thus abstracting away from a number of subtypes and their respective differences.

Today, the differences between classification and generalisation are well understood, but this has not always been the case. Before Frege laid the foundation for a modern axiomatic logic with his "Begriffsschrift" (Concept Script) in 1879, there was no systematic way to avoid mistakes arising from confusing classification with generalisation and the logical fallacies that inevitably follow (see section 3). The *arbor porphyriana* (aka, "tree of knowledge", a concept tree as inspired by Porphyrios) in the version of Purchotius from 1730 relates a supertype (e.g., "Animal") with a type (e.g, "Homo") in the same way as a type ("Homo") with its instances (e.g., "Petrus") (Purchotius (1730)). This lack of proper distinction between classification and generalisation could still be observed in 1890 (see Frege (1892, page 193, footnote 4)). As such, this relatively modern attainment should not be given up lightly, even when apparent advantages seem to suggest this from a pragmatic point of view.

A number of (meta-)modelling language design approaches have been proposed that can be regarded as emphasising the similarities between classification and generalisation and de-emphasising their differences in order to yield

- a simplified language design (Jackson (2006)),
- more flexibility in metamodelling (Varró & Pataricza (2003), Gitzel & Merz (2004)), and
- an increased utility of a language specification (OMG (2004)).

In this paper, I argue that there is value in maintaining a clear distinction between classification and generalisation, and that alternatives to the above mentioned approaches exist that maintain a clear distinction.

Section 2 compares classification with generalisation pointing out similarities and fundamental differences. Section 3 analyses some of the aforementioned approaches and suggests alternative solutions. Section 4 concludes.

2 Comparison

The characterisation of classification and generalisation in the introduction, as typically using instances and types as their domains respectively, suggests that these abstraction mechanisms serve very different purposes and indeed this is the case for most common usage scenarios. However, note that classification may also be performed on types (metamodelling, see Kühne (2006)) and it is possible to generalise at the instance level as well, leading to so-called *abstract objects* (Mittelstraß (1995)). In the following comparison, I am hence careful to compare classification with generalisation by applying them to the same domain in order to avoid observing discrepancies that only exist due to the application to different domains.

Copyright ©2009, Australian Computer Society, Inc. This paper appeared at the The Sixth Asia-Pacific Conference on Conceptual Modelling (APCCM 2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology, Vol. 96, Markus Kirchberg and Sebastian Link, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

2.1 Formalisation

For the following comparison it is useful to introduce the notion of a "concept" as conceived by Frege. Using the terminology of his pupil Carnap (Carnap (1947)), a concept C has an extension $\varepsilon(C)$, all instances falling under the concept C, and an intension $\iota(C)$, a predicate characterizing whether an element belongs to the concept or not, so that

$$\varepsilon(C) = \{x \mid \iota(C)(x)\} \quad (1)$$

We can now state whether a concept C *classifies* an instance e, i.e. $e \,\Lambda\, C$, using an extensional or an intensional viewpoint. Here's the extensional variant:

$$e \,\Lambda\, C \iff e \in \varepsilon(C). \quad (2)$$

Using the usual interpretation of generalisation, a concept C_g is more general than another concept C_s, i.e., $C_s < C_g$, if it includes all the instances falling under C_s:

$$C_s < C_g \iff \varepsilon(C_s) \subseteq \varepsilon(C_g). \quad (3)$$

Equations 2 & 3 make it obvious that a direct comparison between classification and generalisation is hindered by the fact that the former's domain[1] typically consists of instances and the latter's domain typically consists of concepts.

One way to enable an adequate comparison would be to look at concepts C_g and C_s as instances, i.e., instead of considering their *type facets* (e.g., their attributes), by which they define the shape of their instances, one could consider their *instance facets* (e.g., their property values), that is, properties that are associated with the concepts themselves (Atkinson & Kühne (2003)). For example, for the purpose of modelling a pet store, the instance facet of the concept "Dog" could have a tax rate property with the value "16%" whereas the instance facet of concept "DogFood" could have the value "7%" for the same property.

However, this way of looking at concepts is unfamiliar to most and would also imply that we had to use meta-concepts to classify concepts. Therefore, I perform the comparison at the instance level and use generalisation at the instance level by using the notion of an *abstract object* (Kamlah & Lorenzen (1996)). An abstract object represents all instances that are considered to be equivalent to each other for a certain purpose, e.g.,

$$P(|x|_\sim) \iff \forall\, y : y \sim x \to P(y)$$

The abstraction operator $|\cdot|$ gives us a way to make a statement about all instances that are considered equivalent to each other. For example, while

$$HasFourLegs(Lassie)$$

is true if the instance "Lassie" has four legs, the expression

$$HasFourLegs(|Lassie|_{\sim_{dog}}) \quad (4)$$

is only true if *all* instances considered equivalent to "Lassie" have four legs (by means of $\sim_{dog}$, which here is meant to regard all instances of subspecies "Dog" to be equivalent with each other).

Note that $|Lassie|_{\sim_{dog}}$ is not a type/concept. What we assert of $|Lassie|_{\sim_{dog}}$ is asserted of an instance (and all other instances that are equivalent to it). The expression

$$NamedByLinnaeusIn1758(|Lassie|_{\sim_{dog}}) \quad (5)$$

[1] The type of the left hand side element in a relation.

is false since Linnaeus did not name all dogs, but the concept "Dog" (Canis lupus familiaris), i.e., the subspecies subordinate to species "Canis lupus".

It is of course true, though, that $|Lassie|_{\sim_{dog}}$ implicitly defines a set of instances (an equivalence class), which could be the extension of a concept—in particular, if the equivalence relation $\sim_{dog}$ is defined by referring to a predicate $Dog(X)$, i.e., instances are considered equivalent with each other if they satisfy predicate $Dog(X)$. Yet, $|Lassie|_{\sim_{dog}}$ refers to *all instances* of that set, not the *set* itself. In other words, the concept "Dog" is not a collection of dogs.

If one wants to introduce an alternative name to the notion of an abstract object like $|Lassie|_{\sim_{dog}}$ then *prototype* would best describe its nature. An abstract object captures what is universal about a set of instances but resides at the same logical level as the instances, much like an object in a prototype-based language from which other objects can be cloned (Ungar & Smith (1987)). It thus has the quality of a type but is not (yet) a type, hence "*proto*type".

Finally, note that the way we generalise from *Lassie* to $|Lassie|_{\sim_{dog}}$ conforms to how a number of special concepts may be generalised to a general one (see equation 3). A general concept can be regarded as capturing what is universal about its subconcepts. This is true with respect to instance facet properties, e.g., tax rate values, but also with respect to type facet properties, e.g., attributes that require certain properties for instances, such as "age : Integer". If an equivalence relation $\sim_{named}$ considers all subconcepts to be equivalent that feature a "name" attribute and is used on a number of subconcepts, such as "Collie", "Poodle", and "Beagle", then $|Collie|_{\sim_{named}}$ is the generalisation of these subconcepts and could be labelled "NamedDog".

Because a general concept $C_g = |C_i|_\sim$ is derived from more specialised concepts C_i $(i \in [1..n])$ by disregarding differences in the C_i, every C_i will at least have the requirements on instances that C_g has, which means that if an instance satisfies the requirements of a C_i, it will also satisfy the requirements of C_g:

$$\forall\, i, x : \iota(C_i)(x) \to \iota(C_g)(x) \quad (6)$$

Here, I am referring to the intensions of concepts since I want to emphasise the fact that a concept can be viewed as being independent from the instances it describes. A concept resides at a higher logical language level than the instances within its extension, and its intension can be used as a judge with respect to instances that may or may not fall under the concept. This view of concepts is particularly important if one wants to deal with dynamic extensions that may shrink/grow over time and it is the prevalent one in modelling languages such as the UML ((OMG (2007))) where a class / type is regarded as an intensional description of its instances.

From equations 1 & 6 it follows that the extension of the superconcept is a superset of the union of the extensions of its subconcepts.

$$\varepsilon(C_g) = \varepsilon(|C_1|_\sim) \supseteq \bigcup_i \varepsilon(C_i). \quad (7)$$

In equation 7, "$\supseteq$" can be replaced with "$=$", if

$$\forall\, e : e \in C_g \to \exists\, i : e \in C_i,$$

i.e., if there are no elements which are classified by C_g but not by any C_i. The latter holds for all natural objects which always have proper type which is more specific than a generalised type, but may not be true in modelling or programming, where instances of generalised types may be created unless they are declared as being "abstract".

2.2 Similarities

When taking a sufficiently broad view, it is indeed possible to identify a number of similarities between classification and generalisation.

2.2.1 Abstraction

Classification and generalisation can both be regarded as abstraction mechanisms. By abstracting away from individual detail they give rise to relationships that are typically many-to-one,[2] i.e., many elements are abstracted from to yield one representative. By using the representative, i.e. the type or the generalisation,[3] one is able to assert facts about a large number of elements, e.g. what relationships they may engage in, without referring to individual elements. Hence, both classification and generalisation help to reduce the complexity of specifications.

2.2.2 Membership

Instances and subtypes can both be viewed as belonging to or being members of their respective representative. Each instance is a member of the set characterised by its type and each subtype is a member of the subtype hierarchy of which the generalisation forms the top.

2.2.3 Description

Obviously, the representatives can be regarded as *describing* their members, i.e., the members are at least partially defined by virtue of their membership. Consider aggregation, as an example for another many-to-one relationship, that does not have such a descriptive flavour. The *whole* can be used as a representative of its *parts*, but does not describe the latter.

2.2.4 Reuse

Finally, both types and generalisations can be usefully kept in libraries as they allow modellers to derive new elements from existing ones, as instances or subtypes respectively. They, therefore, both support reuse and incremental development in the sense that a modeller may reuse such library elements and only needs to specify what is different about the new element.

2.3 Differences

While the previous section appears to suggest that classification and generalisation have a lot in common, it actually refers to rather superficial similarities which distract from the fundamental differences between them.

2.3.1 Abstraction

Everything that has been stated in section 2.2.1 regarding the nature of classification and generalisation as abstraction mechanisms regarding the reduction of complexity can also be stated about aggregation, leaving only their descriptive nature and utility in libraries as differences. The vast differences between, say generalisation and aggregation, highlight what little significance a commonality in terms of "supporting abstraction" actually has.

[2] Many-to-many forms, known as multiple inheritance and multiple classification, exist but are not very commonly supported.

[3] I refrain from using "supertype" as the latter term implies that the element obtained by generalising has a type role.

2.3.2 Membership

While types and generalisations may both be regarded as representatives, they are in fact at different logical language levels with respect to each other. In section 2.1, I used expressions 4 & 5 to demonstrate the difference between properties at the instance level and the type level. Note that when using an abstract object (a generalisation) we can directly assert a property as in expression 4. To achieve the same with a type, we actually need to use universal quantification as in

$$\forall x : \iota(Dog)(x) \rightarrow HasFourLegs(x).$$

This universal quantification is often left implicit, using the pragmatic assumption that assertions are made about instances of a type, rather than the type itself (expression 5 being an example for the latter).

Yet, the above must not detract from that fact that $|C|_\sim$ refers to *all elements* within the equivalence class implied by $\sim$, whereas $\varepsilon(C)$ refers to the *set* of all elements, i.e., the equivalence class *itself* (given a corresponding $\iota(C)$). Hence, when associating meaning to "$|C|_\sim$" and "C" by using a mapping "μ", "$\mu(|C|_\sim)$" is multi-valued, i.e., here "μ" is a relation, whereas "$\mu(C)$" has a single result, i.e., here "μ" is functional. Assuming a stratification of values in which sets of objects rank higher than the objects they contain (see Russell's Theory of Types (Whitehead & Russell (1910))) clearly, types are at a higher logical level than their instances and hence also at a higher level than generalisations of their instances.

From the fact that the result of classification is a set (rather than all the members of the set), it follows that classification gives rise to a relation which is not transitive. While we have equation 6 for generalisation, and hence transitivity, i.e.,

$$C_1 < C_2 \land C_2 < C_3 \rightarrow C_1 < C_3,$$

for classification, obviously an element C_1 with $C_1 \in \varepsilon(C_2)$, need not be in an element in a set C_3, even if $\varepsilon(C_2) \in C_3$. Figure 1 uses a 3D variant of a Venn diagram to illustrate the fact that an element (Lassie) is automatically also an (indirect) member of the supertype of its type, but is not automatically a member of the type of its type.

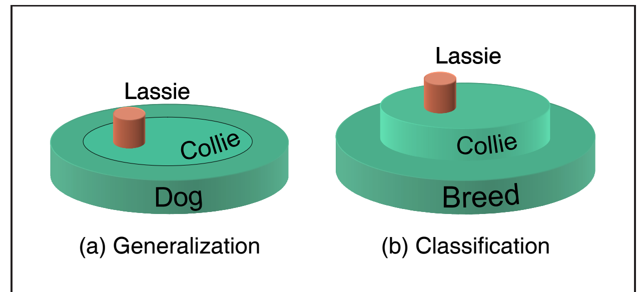

Figure 1: Differences in transitivity

In (Kühne (2006)), I argued that in contrast to classification, generalisation cannot be used to erect a *meta*level hierarchy because of the transitivity of the relation it implies.

2.3.3 Description

Referring to the previous section 2.2 again, it is true that types and generalisations both have a descriptive role. However, note that while a type typically only shapes the *instance facet* of its instances, i.e.,

controls its instances' properties, a generalisation is typically only used to shape the *type facet* of its subordinated elements, i.e., supertypes are typically used to make subtypes inherit type facet features (such as the attribute "age : Integer"). It is, however, possible to influence the type facet with types (see "deep instantiation" (Kühne & Schreiber (2007))) and influence instance facets with specialisation (see Smalltalk "class variables" (Goldberg & Robson (1983))).

2.3.4 Reuse

As a result of their different descriptive roles, types and generalisations have rather different purposes when used as library elements. Types provide a vocabulary that is used without being refined. There is no specification of a "difference" to the derivable element, but one simply provides values for the schema made available through the type. Generalisations, on the other hand, are refined when used by specifying what has to be added to derive a specialised element from a general one.

In summary, although classification and generalisation share some superficial properties, they are intrinsically and unmistakeably different.

3 Analysis of Approaches

In the following I examine a number of approaches that see benefits in de-emphasising the differences between classification and generalisation in one way or the other.

3.1 In the Name of Simplicity

Alloy is a language for the specification of software systems (Jackson (2006)). One of the tutorials on Alloy contains the following statement:

> "*Set membership and subsets are both denoted* `in`." (Seater & Dennis (2008)).

In other words, Alloy uses one "`in`" operator for both "$\in$" and "$\subseteq$". This is surprising at first because of the fundamental differences between these two relations. As discussed earlier on, "$\subseteq$" (corresponding to generalisation) is transitive, whereas "$\in$" (corresponding to classification) is not.

This apparent puzzle is easily resolved by observing that Alloy does not fully support modelling at the instance level. The modeller is rather required to model instances as singleton sets, as in

```
one sig Lassie extends Collie {}
```

Here the set of all collies is specialised by a singleton set[4] `Lassie` which is used to uniquely reference the intended instance "Lassie". This way the "`in`" operator can be considered to only support the "$\subseteq$" interpretation. Checking

```
Lassie in Collie
```

yields true, because the set `Lassie` indeed contains a (unique) element which is also a member of the set `Collie`.

The good news is that, hence, "`in`" does not really confound the set membership and subset relations as quoted above. This confusion may still be claimed regarding a modeller's intention but technically "`in`" always corresponds to "$\subseteq$".

The bad news is, however, that representing instances as singleton sets

- can be very confusing for novices, and
- denies the modeller the ability to distinguish between a singleton set and its instance.

[4] A set with exactly one and thus unique instance.

Novices will read "`in`" to mean "$\in$" in situations like this one:

```
Lassie in House
```

when trying to check whether Lassie is at home, i.e., test whether "Lassie" is among the elements of the set "House" while they are actually checking whether the (singleton) set `Lassie` is a subset of the set `House`.

Strangely, the below expression, using `some` ("$\exists$") to extract an element `x` from the set `Lassie`,

```
some x : Lassie | x in House
```

also yields true, although there is no element `x` in `Lassie` which is a subset of `House`. Although this may suggest, that here "`in`" is interpreted as "$\in$" after all, Alloy interprets the unique instance within `Lassie` as the singleton set containing "Lassie". This also takes place when referring to generated instances, such as "`Collie$0`", which are converted into singleton sets, e.g., "`{Collie$0}`".

Furthermore,

```
some x : House | x in House
```

yields true, if the house is not empty, again strongly suggesting that "`in`" is interpreted as "$\in$". Yet again, however elements in `House` are converted to their singleton sets before the `in` test, so that the actual "$\subseteq$" test yields the expected result.

In total, even experienced modellers need to be very wary in order to avoid misreading Alloy specifications that involve instances. Sometimes an Alloy expression (e.g., `Lassie`) appears to denote an instance, as it is used to uniquely reference a certain element, and sometimes it clearly is used as a set (as in `some x : Lassie | ...`). While there is always a consistent technical reading of "`in`" as "$\subseteq$", some of its usages are highly suggestive of an "$\in$" interpretation. Understanding Alloy's results thus requires an understanding of its implicit conversion of elements to their respective singleton sets. Of course, it could be argued that the latter is not necessary and that Alloy manages to always associate the intended meaning of either "$\in$" or "$\subseteq$" to `in`, but this implies that one concedes to a blurring between classification and generalisation which is potentially dangerous (the intention could deviate from the actual meaning) and inappropriate for novices that have not yet been sufficiently exposed to a proper distinction between classification and generalisation. The latter is a problem when using Alloy in first year courses such as devised by Noble et al. (Noble et al. (2008)).

In section 2.3, I already pointed out the difference between properties at the instance level and at the type level (see expression 4 versus expression 5). Due to Alloy's approach to representing instances with singleton sets, the modeller loses the ability to separate these two levels of properties. Technically, it is an error to ask an instance for its member count since it is not a container type like a set, but an Alloy representation of an instance will happily answer "1". This may be regarded as a feature rather than a bug since it enables specifications which are agnostic as to whether they are dealing with instances or sets. However, this convenience comes with a price because one loses the ability to detect (i.e, type check for) erroneous data flows which lead instances to appear in places where only sets should occur and vice versa.

The rationale given for Alloy's treatment of instances as singleton sets, and the corresponding apparent unification of "$\in$" and "$\subseteq$" to "`in`", is the desire to uniformly allow the application of a single operator

to both scalars (instances) and sets (Jackson (2006)). Overall, Alloy represents a highly elegant language design which enables very concise and readable specifications. However, I believe that prioritising simplicity (just one "in" operator) and uniformity (applicable to both instances and sets) does not justify the loss in expressiveness and clarity as discussed above.

One could maintain the uniformity requirement of having just one "in" operator by overloading it, i.e., using the same syntax for "$\in$" and "$\subseteq$" but retaining their different meaning depending on the types of the arguments. Yet, this would exclude the possibility of detecting type errors resulting from an unintended usage of a set in place of an instance and vice versa.

I claim that the difference between an element and a set, including the set that only contains said element, is big enough in order to abandon simplicity and uniformity in favour of improved type checking capabilities. With respect to Alloy's treatment of instances, I therefore propose to allow the direct modelling of instances and to use two different operators for "$\in$" and "$\subseteq$".

3.2 In the Name of Flexibility

Motivated by the necessity to define the meaning of metalevel boundaries in metalevel hierarchies and in order to support sanity checks for the integrety of such hierarchies, Atkinson and Kühne have proposed a *strict metamodelling* doctrine. According to the latter it is possible to fully understand each level in a metamodelling hierarchy as being instantiated from only the level above it (Atkinson & Kühne (2001)). In order to enforce this property, the only relationships allowed to cross metalevel hierarchy boundaries are "instance-of" relationships.

Subsequently, this doctrine has sometimes been criticised as leading to inflexible infrastructures, and approaches have been developed that relax the strictness requirement in order to provide more flexible metamodelling infrastructures (Gitzel & Merz (2004), Varró & Pataricza (2003)). In particular, Varró and Pataricza take issue with the strict four-layer architecture of the OMG in that it leads to scenarios in which "*...concepts are replicated both on meta-level and model-level...*" (Varró & Pataricza (2003, p. 191)). As a remedy they advocate the introduction of "refinement" as a unification of the notions of instantiation and specialisation, regarding the latter as being highly compatible with each other:

> "*As a result, two model elements can simultaneously be in subtype and instance-of relations...*" (Varró & Pataricza (2003, p. 194)).

Varró and Pataricza even provide a proof for this proposition (Varró & Pataricza (2003, p.195–196)). Their proof relies on the fact that classification and generalisation both give rise to many-to-one relations and that there are pairs of models which *can* be viewed as being in a classification *or* a generalisation relation. However, while Varró and Pataricza would read the aforementioned "*or*" as a logical "or", I maintain that it must be read as a logical "xor". Note that their example using "Graph" and "BipartiteGraph" (Varró & Pataricza (2003, Fig. 6)) excludes attributes. If elements of "Graph" defined attributes—e.g., "Node" could have the attribute "outDegree"—one could clearly see that in the instantiation case the nodes of "BipartiteGraph" would have *values* for "outDegree", whereas in the specialisation case the nodes of "BipartiteGraph" would inherit the "outDegree" attribute. An obvious solution to the "attribute" dilemma is that nodes of "BipartiteGraph" have both "outDegree" values and attributes, but this is most certainly not the intended structure.

Furthermore, if one considered not only model pairs, but deeper derivation structures, the difference in transitivity between classification and generalisation would become apparent.

For these reasons, I consider it inappropriate to view instantiation and specialisation as incarnations of a unified refinement notion that may occur simultaneously between two models.

Gitzel and Merz also aim to reduce the number of concepts in metamodelling hierarchies (Gitzel & Merz (2004)). They model "JavaAccount" as an instance of "Account" using a new form of "instance-of" (classification) relationship which "*...is used in a similar fashion to inheritance relationships...*" (Gitzel & Korthaus (2004, p. 72)). In essence, they are relaxing the strictness doctrine (Atkinson & Kühne (2001)) to allow instantiation across several metalevel boundaries. However, as is apparent from the "Account" / "JavaAccount" example, their new form of "instance-of" relationship in fact has specialisation semantics as opposed to instantiation semantics. Intuitively, every instance of "JavaAccount" should also (indirectly) be an instance of "Account". Also, having elements at one metamodelling level that are instantiated from several different metamodelling levels higher up is incompatible with the requirements for a metamodel hierarchy erecting relation (Kühne (2006)), and as a matter of fact, with Russell's Theory of Types (Whitehead & Russell (1910)).

In ontological metalevel hierarchies (Atkinson & Kühne (2003)), it is obvious that instantiation has to be anti-transitive and instantiation may only occur from one level to an adjacent one. Here is an ill-formed syllogism that violates this rule, representing a logical fallacy:

Man is a species
Socrates is a man
$\therefore$ Socrates is a species

If "species" is replaced with "mammal" then the syllogism works as intended because then the first "is a" corresponds to generalisation as opposed to classification. The above ill-formed syllogism illustrates the inappropriateness of assuming that an instance (Socrates) could be classified by an element that is two levels higher up in the metalevel hierarchy (species).

Gitzel et al. appear to require certain concepts at more than one metamodelling level since there are metamodelling hierarchies which cannot be aligned with each other (Atkinson & Kühne (2001)). In such cases, which include primitive types like "Integer", strictness can be maintained for the hierarchies individually, and a single hierarchy may be used multiple times in conjunction with another one. I believe that this is an acceptable form of replication since it corresponds to "multiple usage" rather than "multiple definition".

Gonzalez-Perez and Henderson-Sellers also appear to treat classification and generalisation as closely related relationships because they let both cross metalayer boundaries in parallel Gonzalez-Perez & Henderson-Sellers (2006, p. 88, Fig. 20), but note that their meta*layers* are not aligned with meta*levels*, the latter being defined by classification relationships only. Although their usage-oriented *layering* appears to blur the differences between classification and inheritance, the underlying *level hierarchy* does not suffer from any such conceptual difficulties.

Summarising, with respect to attempts to relax the strictness doctrine, I argue that there is no value

in weakening the significance of metalevel boundaries. On the contrary, abandoning strictness opens the door for errors that no longer can be detected as such. If there is value in partitioning a metamodelling hierarchy along boundaries other than the metalevel boundaries, such structures should be overlayed, or offered as alternative views, but not undermine the integrity of the metalevel boundaries themselves.

3.3 In the Name of Utility

In order to promote consistency and parsimony, the OMG introduced a "Core" model from which both the Meta-Object Facility (OMG (2006)) and the UML definition (OMG (2007)) are derived. Regarding the UML definition, note that it is both specialised[5] from the Core (OMG (2004, p. 12, Fig. 7.2)) and also instantiated from the Core[6] (OMG (2004, p. 14, Fig. 7.4)).

Formally, we both have UML Λ Core (Core classifies the UML definition) and UML $<$ Core (Core generalises the UML definition). In section 3.2, I have already argued that this is inappropriate, i.e., impossible to maintain in a sound manner. However, I was making the assumption that both models are used with an ontological interpretation, which was appropriate regarding Varró and Pataricza's examples. As observed by Atkinson and Kühne, however, the OMG uses the Meta-Object Facility (MOF) and hence by implication the Core, both as an ontological type model and as a linguistic type model (Atkinson & Kühne (2005, p. 409, Fig. 15(b))). In the following, I will investigate under which circumstances it is possible for a linguistic type model to be both the type model and a supermodel for another model.

Formally, we are looking for elements M (UML) and MM (Core) which fulfil the following constraint:

$$M \in \varepsilon(MM) \wedge \varepsilon(M) \subseteq \varepsilon(MM). \tag{8}$$

Assuming an element m (a UML model), the above constraint with $\varepsilon(M) = \{m\}$ and $\varepsilon(MM) = \{m, M\}$, yields

$$M \in \{m, M\} \wedge \{m\} \subseteq \{m, M\} \tag{9}$$

which fulfils the constraint. The following observations are noteworthy:

- The only way in which constraint 8 may (non-trivially) be true is for an M that has a type role. If M were an instance without a type role, i.e., $\varepsilon(M) = \emptyset$, it could not meaningfully take the place of M in constraint 8 since it would not have an extension whose elements may also appear in the extension of MM; its extension could not be non-trivially considered to be a subset of MM's extension.

- From constraint 8 it follows that $\varepsilon(MM)$ must contain two elements which can be considered as being in a classification relation. Formally,

  $$M \in \varepsilon(MM) \wedge \varepsilon(M) \subseteq \varepsilon(MM) \wedge \varepsilon(M) \neq \emptyset \rightarrow$$
  $$\exists\, m : M \in \varepsilon(MM) \wedge m \in \varepsilon(MM) \wedge m \in \varepsilon(M).$$

  Hence, MM must (at least implicitly) define a notion of instantiation between (some of) its elements. Indeed, the Core/MOF defines instantiation between its elements.

- The above point implies that MM cannot only instantiate M—as well as being M's supermodel—but also m, an instance of M. In fact, this is the case in our example, as the MOF/Core can (linguistically) classify UML models. This ability is used to provide a common repository/interchange format for UML models.

- If M and MM are chosen to be equivalent, i.e., $\iota(M) \sim \iota(MM)$, and hence describe the same models, ignoring the implied different logical levels in a stratified scheme, then MM can be said to be *self-describing*. This is useful to obtain a self-terminating metamodelling hierarchy, such as the OMG's four layer architecture.

With respect to the last point, note that choosing $M = MM$, would lead to a set that contains itself, i.e. an infinite regression, by expressions 8 & 9. This is not possible assuming ZFC set theory (Kunen (1980))), but operationally one could of course construct a metamodelling hierarchy which features as many copies of the top-level as desired. One could use an *MM-quine*, i.e., an MM that contains a quoted version of itself. Upon instantiation one would unquote this version and supplement it with a quoted version again. This is the principle with which one can write programs that replicate/output themselves.

Note that Core is used as a linguistic type model and can instantiate itself. This makes it possible to avoid the logical stratification problems which were mentioned in section 3.2 as follows: The linguistic Core type model can instantiate both the UML definition and another Core instance, the latter acting as a supermodel for the UML definition. This way the UML definition can simultaneously be an instance and a subtype of Core, thus being shaped in two ways, albeit referring to two (identical) instances of Core. In contrast to the "BipartiteGraph" example of section 3.2, it does make sense for the UML definition to have features for its elements—such as "name"—that exist both as properties (with values) and as attributes (shaping UML instances, i.e., user models).

As a (surprising!) result, it is indeed possible to increase the utility of Core/MOF by using it both as a repository format (linguistic type) and a language definition supermodel (language definition supertype) while maintaining a clean structure free of inconsistencies. I propose to use the above developed interpretation of the dual role of the Core/MOF as an underpinning to its intended dual purpose.

4 Conclusion

Classification and generalisation can be regarded as sharing a number of properties. The aim of this paper was to demonstrate that claiming any resemblance between the two abstraction mechanisms is only meaningful when taking a very broad view. Since Frege's pioneering work in 1879, there should be no doubt regarding the fundamental differences between classification and generalisation and the resulting incompatibility regarding a simultaneous usage.

Apart from recalling this important lesson learned, this paper makes a number of contributions:

First, I used the notion of an abstract object to define generalisation on instances, i.e., avoiding the usual view of specialisation as extending type facets. This provided an unusual interpretation of supertypes as generalising the instance facets of their subtypes as well as their type facets. First and foremost, however, it allowed an adequate comparison between classification and generalisation which focussed on intrinsic differences rather than on discrepancies that result from different usage scenarios.

[5] Actually, the term "dependent on" is used which refers to package usage which can be appropriately regarded as specialisation.

[6] Actually, it is instantiated from the MOF which contains the Core.

Second, I pointed out some conceptual problems and inconsistencies of approaches which use a unifying view on classification and generalisation, deemphasising their differences. Whenever needed and possible, I offered a logically consistent view, restoring an adequate separation of classification versus generalisation. To the best of my knowledge, in particular the formal investigation into whether the OMG's view of the Core/MOF as both a repository format and a language definition supermodel has a sound interpretation, is novel.

Third, I argued that acknowledging the differences between classification and generalisation—e.g., by complying with the "strict metamodelling" doctrine—one gains an opportunity for sanity checks regarding the integrity of metamodelling hierarchies that otherwise would not exist. Detecting incorrect dataflow in specifications becomes possible if instances and types are not substitutable for each other. Logical fallacies that are introduced by crossing multiple metalevel boundaries at once or introducing circular definitions are impossible, if metalevels are stratified according to Russell's Theory of Types. Finally, a metamodel can be defined to have a dual purpose in a sound manner, if it features elements that can classify each other.

Ultimately, there is no single correct way of designing languages and in particular Alloy's prioritisation of flexibility over safety can certainly be defended. Also, I am aware that the authors of the work which I subjected to some critical remarks may take different definitions for classification and generalisation as a basis and hence arrive at different conclusions regarding their compatibility, maintaining internal consistency.

However, I hope that the observations made in this paper may be of use for future language designers. While they may not choose to subscribe to the most rigorous treatment I have suggested, they will at least be in a position to consciously deviate from it, explicitly rationalising as to why a non-strict treatment should be preferred and whether it is worth accepting the resulting loss in precision and loss of sanity checks.

Acknowledgements

I would like to thank Lindsay Groves for his very helpful comments on a draft of this paper.

References

Atkinson, C. & Kühne, T. (2001), 'Processes and products in a multi-level metamodeling architecture', *International Journal of Software Engineering and Knowledge Engineering* **11**(6), 761–783.

Atkinson, C. & Kühne, T. (2003), 'Rearchitecting the UML infrastructure', *ACM Transactions on Modeling and Computer Simulation* **12**(4), 290–321.

Atkinson, C. & Kühne, T. (2005), Concepts for comparing modeling tool architectures, *in* L. Briand, ed., 'Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, MoDELS / UML', Springer Verlag, pp. 398–413.

Carnap, R. (1947), *Meaning and Necessity: A Study in Semantics and Modal Logic*, University of Chicago Press.

Frege, G. (1892), Über Begriff und Gegenstand (On Concept and Object), *in* 'Vierteljahrsschrift für wissenschaftliche Philosophie', Vol. XVI, Fues's Verlag, pp. 192–205.

Gitzel, R. & Korthaus, A. (2004), The role of metamodeling in model-driven development, *in* 'Proceedings of the 8th World MultiConference on Systemics, Cybernetics and Informatics', Vol. IV, pp. 68–73.

Gitzel, R. & Merz, M. (2004), How a relaxation of the strictness definition can benefit MDD approaches with meta model hierarchies, *in* 'Proceedings of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics', Vol. IV, pp. 62–67.

Goldberg, A. & Robson, D. (1983), *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA.

Gonzalez-Perez, C. & Henderson-Sellers, B. (2006), 'A powertype-based metamodelling framework', *Software and Systems Modeling* **5**(1), 72–90.

Jackson, D. (2006), *Software Abstractions: Logic, Language, and Analysis*, The MIT Press, Cambridge, Mass.

Kamlah, W. & Lorenzen, P. (1996), *Logische Propädeutik*, Metzler.

Kühne, T. (2006), 'Matters of (meta-) modeling', *Software and Systems Modeling* **5**(4), 369–385.

Kühne, T. & Schreiber, D. (2007), Can programming be liberated from the two-level style? – Multi-level programming with DeepJava, *in* 'Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications', ACM, NY, USA, pp. 229–244.

Kunen, K. (1980), *Set Theory: An Introduction to Independence Proofs*, Elsevier. ISBN 0-444-86839-9.

Mittelstraß, J., ed. (1995), *Enzyklopädie Philosophie und Wissenschaftstheorie*, Verlag J. B. Metzler.

Noble, J., Pearce, D. J. & Groves, L. (2008), Introducing Alloy in a software modelling course, *in* 'ETAPS 2008 Workshop on Formal Methods in Computer Science Education (FORMED)'.

OMG (2004), *Unified Modeling Language Infrastructure, Version 2.1.2*. OMG document formal/2007-11-04.

OMG (2006), *Meta Object Facility (MOF) 2.0 Core Specification*. OMG document formal/2006-01-01.

OMG (2007), *Unified Modeling Language Superstructure Specification, Version 2.1.1*. OMG document formal/07-02-05.

Purchotius, E. (1730), Institutiones philosophicae I, *in* 'Tomus primus, Complectens Logicam & Metaphysicam', Apud Joannem Manfrè.

Rayside, D. & Campbell, G. T. (2000), An Aristotelian understanding of object-oriented programming, *in* 'Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications', ACM Press, pp. 337–353.

Seater, R. & Dennis, G. (2008), 'Tutorial for Alloy analyzer 4.0', http://alloy.mit.edu/alloy4/tutorial4/.

Ungar, D. & Smith, R. B. (1987), Self: The power of simplicity, *in* 'Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications', ACM press, pp. 227–242.

Varró, D. & Pataricza, A. (2003), 'VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML', *Journal of Software and Systems Modelling* **2**(3), 1–24.

Whitehead, A. N. & Russell, B. (1910), *Principia Mathematica*, Suhrkamp, Frankfurt.