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 confusioning 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 Puchotius from 1730 relates a supertype (e.g., “Animal”) with a type (e.g., “Homino”) in the same way as a type (“Homino”) with its instances (e.g., “Petrus”) (Puchotius (1730)). This lack of proper distinction between classification and generalisation could still be observed in 1890 (see Frege (1889, 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 & Patarica (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.
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 \( \{ \text{Lassie} \}_{\sim_{\text{dog}}} \) implicitly defines a set of instances (an equivalence class), which could be the extension of a concept—in particular, if the equivalence relation \( \sim_{\text{dog}} \) is defined by referring to a predicate \( \text{Dog}(X) \), i.e., instances are considered equivalent with each other if they satisfy predicate \( \text{Dog}(X) \). Yet, \( \{ \text{Lassie} \}_{\sim_{\text{dog}}} \) 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 \( \{ \text{Lassie} \}_{\sim_{\text{dog}}} \), 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 “prototype”.

Finally, note that the way we generalise from Lassie to \( \{ \text{Lassie} \}_{\sim_{\text{dog}}} \) 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 \( \sim_{\text{named}} \) 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 \( \{ \text{Collie} \}_{\sim_{\text{named}}} \) is the generalisation of these subconcepts and could be labelled “NamedDog”.

Because a general concept \( C_g = |C|_{\sim} \) 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 \( C_g \).

\[ \forall i, x : i(C_i)(x) \rightarrow \epsilon(C_g)(x) \] (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.

\[ \epsilon(C_g) = \epsilon(|C|_{\sim}) \supseteq \bigcup_i \epsilon(C_i). \] (7)

In equation 7, “\( \supseteq \)” can be replaced with “\( \sim \)”, if

\[ \forall e : e \in C_g \rightarrow \exists i : e \in C_i, \]

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”.

2.1 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 \( \epsilon(C) \), all instances falling under the concept \( C \), and an intension \( \epsilon(C) \), a predicate characterising whether an element belongs to the concept or not, so that

\[ \epsilon(C) = \{ x | \epsilon(C)(x) \} \] (1)

We can now state whether a concept \( C \) classifies an instance \( e \), i.e., \( e \in \epsilon(C) \), using an extensional or an intensional viewpoint. Here’s the extensional variant:

\[ e \in \epsilon(C) \iff e \in \epsilon(C). \] (2)

Using the usual interpretation of generalisation, a concept \( C_i \) is more general than another concept \( C_g \), i.e., \( C_i \subset C_g \), if it includes all the instances falling under \( C_i \):

\[ C_i \subset C_g \iff \epsilon(C_i) \subseteq \epsilon(C_g). \] (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 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 \( C_i \) and \( C_g \) 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 “10%” 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.,

\[ P(x_i) \iff \forall y : y \sim x \rightarrow P(y) \]

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

\[ \text{HasFourLegs}(\text{Lassie}) \]

is true if the instance “Lassie” has four legs, the expression

\[ \text{HasFourLegs}(\{ \text{Lassie} \}_{\sim_{\text{dog}}} \) (4)

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

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

\[ \text{NamedByLinnaeusIn1758}(\{ \text{Lassie} \}_{\sim_{\text{dog}}} \) (5)

\footnote{The type of the left hand side element in a relation.}
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, i.e., many elements are abstracted from to yield one representative. By using the representative, i.e. the type or the generalisation, 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. Considering 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.

2.3.2 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 the 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 \(\iota(C)\) and \(\sim\) by using a mapping \(\mu\), \(\mu(|C|_\sim)\) is multi-valued, i.e., here “\(\mu\)” is a relation, whereas \(\mu(C)\) has a single result, i.e., here “\(\mu\)” 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 (?) ) 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.

\[ C_1 \subset C_2 \land C_2 \subset C_3 \rightarrow C_1 \subset C_3, \]

For classification, obviously an element \(C_1\) with \(C_1 \subset C_2\), need not be in an element in a set \(C_3\), even if \(C_2 \subset C_3\).

Figure 1: Differences in transitivity

In (Kühne (2006)), I argued that in contrast to classification, generalisation cannot be used to erect a metalevel 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 “∈” and “⊆”. This is surprising at first because of the fundamental differences between these two relations. As discussed earlier on, “⊆” (corresponding to generalisation) is transitive, whereas “∈” (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 \( Lassie \) which is used to uniquely reference the intended instance “Lassie”. This way the “in” operator can be considered to only support the “⊆” 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 “⊆”.

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.

Novices will read “in” to mean “∈” 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 \( \text{some} \) (“∃”) to extract an element \( x \) from the set \( Lassie \),

\[ \text{some } x : \text{Lassie } | \text{ x in House} \]

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 “∈” 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,

\[ \text{some } x : \text{House } | \text{ x in House} \]

yields true, if the house is not empty, again strongly suggesting that “in” is interpreted as “∈”. Yet again, however elements in \( House \) are converted to their singleton sets before the test, so that the actual “⊆” 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 \( \text{some } x : \text{Lassie } | \ldots \)). While there is always a consistent technical reading of “in” as “⊆”, some of its usages are highly suggestive of an “∈” 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 “∈” or “⊆” to “in”, but this implies that one conceives 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 2.3.1, 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 “∈” and “⊆” to “in”, is the desire to uniformly allow the application of a single operator

\[ A \subseteq B \]

\[ A \in B \]

A set with exactly one and thus unique instance.
to both scalars (instances) and sets (Jackson (2006)). Overall, Alloy represents a highly elegant language design which enables very complex 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 “ε” and “⊆” but retaining their different meaning. This approach would have profound effects on 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 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 “ε” and “⊆”.

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 integrity 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 metalevel and model-level.…” (Varró & Pataricza (2003, p. 191)). As a remedy they advocate the introduction of “reification” 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 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 metamodeling level that are instantiated from several different metamodeling 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 (?).

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:

\[
\text{Man is a species}\]
\[
\text{Socrates is a man} \quad .\quad .\quad \text{Socrates is a species}
\]

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 metamodeling 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 metalevel boundaries in parallel (Gonzalez-Perez & Henderson-Sellers (2006, p. 88, Fig. 20), but note that their metalevels are not aligned with metalevels, 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 overlaid, 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 OMG introduced a “Core” model from which both the UML definition, note that it is both specialised5 from the Core (OMG (2004, p. 12, Fig. 7.2)) and also instantiated from the Core6 (OMG (2004, p. 14, Fig. 7.4)).

Formally, we both have UML $\Delta$ Core (Core classifies the UML definition) and UML $\prec$ 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:

$$M \in \varepsilon(MM) \land \forall m \in M \subseteq \varepsilon(MM).$$

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

$$M \in \{m, M\} \land \forall m \subseteq \{m, M\}$$

which fulfils the constraint. The following observations are noteworthy:

- The only way in which constraint 8 may (non-trivially) be true is for $M$ 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,

$$M \in \varepsilon(MM) \land \forall m \in M \subseteq \varepsilon(MM) \land \forall M \neq \emptyset \rightarrow \exists m \colon M \in \varepsilon(MM) \land m \in \varepsilon(M).$$

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

5Actually, the term “dependent on” is used which refers to packaging which can be appropriately regarded as specialisation.

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


