Reflective Constraint Writing
A Symbolic Viewpoint of Modeling Languages

Dirk Draheim

University of Innsbruck
draheim@acm.org

Abstract. In this article we show how to extend object constraint languages by reflection. We choose OCL (Object Constraint Language) and extend it by operators for reification and reflection. We show how to give precise semantics to the extended language OCL_R by elaborating the necessary type derivation rules and value specifications. A driving force for the introduction of reflection capabilities into a constraint language is the investigation of semantics and pragmatics of modeling constructs. We exploit the resulting reflective constraint language in modeling domains including sets of sets of domain objects. We give precise semantics to UML power types. We carve out the notion of sustainable constraint writing which is about making models robust against unwanted updates. Reflective constraints are an enabler for sustainable constraint writing. We discuss the potential of sustainable constraint writing for emerging tools and technologies. For this purpose, we need to introduce a symbolic viewpoint of information system modeling.

Keywords: Meta modeling, multi-level modeling, object constraint languages, generative programming, database migration, schema evolution, clabjects, modeling tools, UML, OCL, Z, Genoupe

1 Introduction

An object-constraint language is a logical language that is embedded into a modeling framework and offers language constructs specific to object-oriented modeling. In this article we show how to add reflection to object-oriented constraint languages. Reflection is about access to the meta level, both introspective as well as manipulative. We need a reflective constraint language to analyze issues and express results in the semantics and pragmatics of information system modeling. Reflective constraints are an enabler for sustainable constraint writing, which is about making models robust against unwanted updates [29]. More specifically, we can exploit a reflective constraint language for:

- *Semantics of modeling languages.* Given meta-level access you can give precise semantics to existing modeling language constructs. We do this for UML power types in this article. Furthermore, you can use a reflective constraint language to extend an existing modeling language with new well-defined modeling constructs.

- *More adequate system analysis.* With today's technologies, i.e., databases, third-generation programming languages and modeling tools, we encounter a model-object divide. This model-object divide is not accidental; it is just a property of current mainstream information system technology, which is established and mature. Nevertheless, the model-object divide sometimes hinders us from stating fully adequate models of domain knowledge. This is so, because a model and its objects together intend domain objects and together encapsulate domain knowledge. For example, you might have some classes $A_1, \ldots, A_n$ and have found some constraints for these classes. Now, you might encounter that these constraints are instances of a general constraint pattern that must hold for an arbitrary number of classes. Without appropriate reflective features, you can only state such constraint patterns in the informal comments. A reflective constraint language is the solution for this. In general, we need full reflective support – limited forms of reflection, like generic types, are not sufficient.
- *Quality assurance for system design.*
  - Ensuring that class names follow a given style guide.
  - Ensuring that each attribute has correctly typed setter- and getter-methods.
  - Ensuring a complex design pattern.
  - etc.

All of the above items are practically motivated [20], i.e., reflective constraint writing is to constraint writing what generative programming is to programming. However, reflective constraint writing is also of importance beyond immediate practical exploitation. It can help in mitigating gaps between different information system paradigms. It can help in mitigating gaps between different viewpoints in information system modeling. We proceed as follows. We choose the OMG standard meta-level architecture as the backbone for our efforts. We extend the OCL (Object Constraint Language) with reification and reflection, resulting in the so-called OCL_R in Sect. 2. We show how to give a declarative semantics for OCL_R in Sect. 3. We review some OCL_R examples in Sect. 4 and also provide a comparison with generative programming, based on the concrete programming language Genoupe .

In Sect. 5, we exploit OCL_R to specify constraints needed in modeling of sets of sets of domain objects. We streamline the discussion by showing how usual class diagrams, i.e., without multilevel modeling constructs, are sufficient to adequately model sets of sets of domain objects if appropriate constraints are provided and if and only if these are made robust against M1-level updates. We further streamline the discussion by considering sustainable constraint writing in the specification language Z in Sect. 6. Then, we generalize the found constraints further to give a precise semantics for UML power types 7. From these discussions, we extract more general notions like sustainable constraint writing and a symbolic viewpoint on modeling languages. In Sect. 8 we discuss model evolution, notions of constraints and viewpoints onto modeling languages. We discuss

related work throughout the paper and summarize related work in Sect. 9. We end the article with a conclusion in Sect. 10.

In the Appendices A, B, C we provide overviews of the abstract syntax of the UML core language, the OCL v2.0 types and the OCL expression language.

2 OCL_R – A Reflective Extension of OCL

The OCL (Object Constraint Language) is syntactically and semantically embedded into the UML meta-level architecture. The aim of this section is to extend OCL with full reflection. Note, that we use the 2006 version of OCL, i.e. OCL v2.0 [71] as the basis for our language extension. We do neither use the current version OCL v2.4 [74] nor the version OCL v2.3.1 [73], which has been released as ISO standard ISO/IEC:19507 [52]. The reason for this is the particularly mature and precise definition of the OCL type system in the former version OCL v2.0 – see Appendix B for a discussion of this issue. If you need to delve into some of the concepts used in the upcoming sections, e.g., the OCL type *OclType* and its generating class *TypeType*, it is important that the standard v2.0 [71] is the authoritative reference for this article and not the newer standards. The choice of standard is for technical reasons only and not due to essential differences. For example, the abstract syntax of the OCL versions v2.0 and versions v2.3.1 and v2.4 are exactly the same. All crucial arguments and statements on OCL in this article, e.g., with respect to expressive power, are independent of the chosen standard.

- Properties of all objects, i.e., objects o : *OclAny*:
  - o.oclIsTypeOf(t:OclType):Boolean – *true iff* $o : t \wedge \nexists t'.t < t'$)
  - o.oclIsKindOf(t:OclType):Boolean – *true iff* $o : t$
  - o.oclInState(s:OclState):Boolean – *Test for state machine state.*
  - o.oclIsNew():Boolean – *Postcondition test for object creation.*
  - o.oclassType(t:OclType):instance of Classifier – *Type casting operation.*
- Properties of meta objects t : *OclType* representing user-defined types:
  - t.name:String – *The name of the type t.*
  - t.attributes:Set(String) – *The set of names of the attributes of t.*
  - t.associationEnds:Set(String) – *Names of association ends navigable from t.*
  - t.operations : Set(String) – *The names of the operations of t.*
  - t.supertypes : Set(OclType) – *The set of all direct supertypes of t*
  - t.allSupertypes : Set(OclType) – *The set of all supertypes of t*
  - t.allInstances : Set(type) – *The set of all instances of type t.*

Fig. 1: OCL inbuilt meta-level access.

2.1 Meta-Object Access in OCL

Standard OCL offers only limited access to the meta-level. The complete list of these OCL meta-level access operations is given in Fig. 1. The OCL meta-level

access is restricted to introspection, i.e., no constraint generation is supported. Even the introspective features are limited. First, way not all meta-relationships that are established by the UML meta model have a counterpart in the OCL language. Second, and this is actually the crucial point, the entry to the introspection is only in terms of the current context of an OCL expression and therefore in terms of only a fixed number of constantly defined user-types. This means, OCL's meta access capabilities yield no functional abstraction over user-defined types and therefore do not add to the expressive power of OCL. The meta access of OCL shows in properties for meta objects representing user-defined types, i.e., objects of type *OclType* and properties that expect a parameter of type *OclType*.

With respect to properties for meta objects, the property *allInstances* is the only one that is specified in the OCL standards since version v2.0. All the other stem from the first version v1.1 [70]. The following list of example constraint expressions cannot be expressed with the OCL inbuilt meta-object access capabilities – please compare the list also to Fig. 1:

1. Names of subclasses of a given type t.
2. The subclasses of a given type t.
3. Attribute names of classes navigable via associations from a given type t.
4. All classes of the user model.
5. The number of classes in the user model.
6. All classes of the user model that have no subclasses.
7. The sum of all Integer attributes of all objects of all classes.
8. Test, whether all attributes of all objects of all classes are initialized.
9. Test, whether all attributes of all classes have setter- and a getter-methods.

All of the above constraint expressions (1) through (9) can be expressed by the OCL-extension OCL_R. The several constraint expressions express different levels of sophistication. The first two constraints (1) and (2) could be made possible by augmenting the list of inbuilt OCL expressions in Fig. 1 by appropriate properties. However, in order to enable all the other constraints a more conceptual refactoring of OCL is necessary, because they long not only for introspective access but also for reflection.

The reflective programming language community distinguishes between *reification* and *reflection* – see also Table 1. Reification turns information about a program, i.e., meta-data, into data and makes it accessible to the programming level. Then, reflection can be understood as the exploitation of reified data. We then also talk about reflection in the wider sense. Reified data can be exploited in two ways. First, it can be exploited for introspective access. Second, it can be used to manipulate program structures. The reified data can be turned into program code itself, we then say that reified data is materialized or re-materialized. We then also talk about reflection in the narrow sense. The usual word for the materialization step of turning reified data into code is generation. We feel that generation somehow stresses more the operational facet of this mechanism. We use both generation and materialization as equal terminology. This terminology works also with respect to meta-level access in modeling languages and also

with respect to constraint writing. Here, reification is about making meta-data accessible to the modeler. Again, reification allows for access to the meta-level and can be exploited for introspection and reflection in the narrow sense, i.e., materialization of reified data into modeling elements.

Reflective programming. (Reflective Constraint Writing). Reflective Languages.	• Reification	
	• Reflection (in the wide sense): Exploitation of Reified Data	• Introspection. • Reflection (in the narrow sense): materialization (generation) of reified data into code (constraints).

Table 1: Attempt to summarize some important reflective programming terminology and its application to reflective constraint writing.

The OCL meta-level access offers only a limited form of reification. The OCL standards [73] explicitly state that OCL does not support the reflection capabilities of the MOF (Meta Object Facility) [77]. Note, that it is not sufficient to add syntactical constructs to a language like OCL to support reflective features. The real work lays in the elaboration of the semantics of such reflective capabilities as, e.g., provided by OCL_R. Shallow statements of the intended meaning of syntactical constructs would not be sufficient as semantic elaboration.

2.2 On The Chosen Declarative Approach for OCL_R Specification

Without loss of generality, we will define OCL_R as an M1-level language, i.e., we define the reification operators Φ and Ψ as well as the concrete syntax $\langle \_ \rangle\downarrow$ and $\langle \_ \rangle\uparrow$ used for them against the background of writing M1-level constraints. Similarly, we specify the well-formedness rules and the semantics of OCL_R from the perspective of writing M1-level constraints. Writing M1-level constraints is the major use case of OCL_R. Writing M1-level constraints is about adding constraint expressions at level M1. For OCL this means that writing M1-level constraints is about writing constraints for M0-level objects. With OCL_R it is possible to write meta object constraints, in particular, constraints on user-defined types at level M1 and therefore extend the semantics of meta models. We will see the specification of the UML power types semantics in Sect. 7 as an example for this. Therefore, there is no need to explicitly generalize the current definitions from a M2-level perspective.

2.3 On the Preciseness of the Chosen Specification Approach

We show how to provide a precise semantics description of OCL_R in this article. In the given OCL_R definitions we rely on the existing UML and OCL semantics

defined in [71, 75] as the foundation for our semantic extensions. Note that our definitions are *free over* the semantic definitions yielded by the OMG specification. This means that even if semantic definitions in the OMG stack might be ambiguous or underspecified for some points, our semantics does not suffer. Furthermore, our specification varies in the way semantic decisions are made for the UML stack. There is no need for us to re-formalize or to fix UML semantics. We can simply assume the semantics as completely specified. The OMG stack forms a sweet spot between preciseness and convenience; at least, the core of it has a widely known and accepted semantics. See also [86] for a discussion of UML 2.0 semantics.

2.4 On Abstract Syntax Oriented Reflection

A reflection mechanism can have a design that is oriented throughout towards abstract syntax or, what we call, an ASCII-based design. In an ASCII-based design meta data is reified as text, i.e., 'String' data. Then reflection operators craft model elements from 'String' data input. In a thoroughly abstract syntax oriented design the data type of reified data is kept abstract and reflection is also realized by operations on this abstract data type. An abstract-syntax oriented design offers an important advantage. It makes it much easier to give precise semantics to the reflection mechanism, in particular, with respect to level-crossing type safety. With an ASCII-oriented design it is easier to provide ad-hoc implementations for a reflection mechanism, in particular, it the implementation has to be provided for an existing platform. The design of OCL_R is thoroughly oriented towards abstract syntax.

Fig. 2: Class diagram.

2.5 Notational Issues of OCL Contexts and OCL Meta Objects

As a minor issue we sometimes want to get rid of context notation in constraint writing in the sequel. The concepts of contexts and meta objects, i.e., objects that represent types, are completely exchangeable. First, consider the following OCL invariants, which are written against the tiny class diagram in Fig. 2:

$$\textbf{context } \mathit{Person} \textbf{ inv}\text{: } \mathit{age} \geqslant 40 \tag{1}$$

$$\textbf{context } \mathit{Foo} \textbf{ inv}\text{: } \mathit{Person.allInstances} \rightarrow \mathit{forAll}(\mathit{age} \geqslant 40) \tag{2}$$

It is easy to see that the constraints (1) and (2) have the same semantics. Now, we can see that the role of *Person* in (1) and (2) are exactly the same. On

the on hand, in (1), you can consider *Person* a meta object. One the other hand, in (2), you can consider $Person.allInstances \rightarrow forAll(\_)$ to provide context for the evaluation of $age \geqslant 40$. Consider the following constraint:

$$Person.allInstances \rightarrow forAll(age \geqslant 40) \tag{3}$$

The information in constraint (3) is complete. The type *Foo* in (2) is not needed in the subsequent expression, it is merely a wildcard, so it can be dropped to yield (3) without loss. Henceforth, we will often write constraints without explicitly given context, in the style of (3). Though the constraint in (3) is clumsier than its version in (1) it is easier to handle in formal argumentations like type derivation or value specifications. Note, that the concrete syntax in (1) and (2) is official OCL syntax; although it is rather used in standard documents as opposed to the more embellished concrete syntax usually found in textbooks.

2.6 Terminology for the OMG Meta-Level Architecture

We need to introduce some notation and terminology for issues in meta modeling architectures to be used in the sequel. The introduction of these notations must not be misunderstood as an attempt to specify, or let's say better, to re-specify the UML meta level architecture and its languages. We take the standard OMG four-level meta model hierarchy, see [75]§7.12, as background architecture, see also Fig. 3. Syntax and semantics of the UML meta meta model, the UML meta model and OCL are taken as granted as defined in [76, 75, 77, 71, 73].

However, the concepts introduced in this section go beyond mere notational issues. We also define important terminology, hand in hand, with notation for it. This way we define the value identity for objects in the meta-level architecture. This value identity is defined across the levels of the meta-level architecture, i.e., it is introduced to make objects at different levels of the architecture comparable. Based on the value identity we will define the meta model reification operator Φ and the model reification operator Ψ.

UML Meta Model Notation We denote the UML meta model by $\mathfrak{M}_2$. Similarly, we denote the UML meta meta model by $\mathfrak{M}_3$.

Object Notation We denote the set of all primitive values by P. The set P is flat, i.e., it is the union of all interpretations of UML's primitive types. We introduce a set of object identifiers and denote it by OID. We denote the set of all attribute values by $V = \mathbb{P}(P \cup OID)$. The power set in the definition of V is necessary, because the UML attributes are, in general, many-valued. We denote the set of attribute names or labels by L. We denote the set of finite subsets of a set M by $\mathbb{F}(M)$. Conceptually, in our notation, an object consists of an object identifier, a finite set of labels from L and an attribute value for each of these selected labels. We define the set of all objects O as Cartesian product of object identifiers OID and finitely L'-indexed sets of attribute values, for all possible subsets of labels, i.e.:

$$O = OID \times \bigcup_{L' \in \mathbb{F}(L)} (V_l)_{l \in L'} \tag{4}$$

The way we defined O, objects are denoted as records [1, 17], or to be precise, object values are denoted as records, and objects are formed as an object reference to record. We use the usual notation for records, i.e., $\langle oid \mapsto \langle l \mapsto x_l \rangle_{l \in L} \rangle$. O contains many objects that are impossible, i.e., objects that can never be instances in the UML meta level hierarchy. This is so because our objects are completely untyped assemblies. They are based on the value set V which is completely flat. This does not harm, because the definitions in Sect. 2.6 are not about semantics, but about notation. We will ensure the well-formedness of object and models of OCL_R later by the definition of the typing relationship $\_ : \_$ and the instantiation relationship $\_ :: \_$ for the corresponding extensions to OCL.

Note that O is the full extension of the meta level hierarchy, i.e., the collection of all potential objects that can be materialized in system states. The set O is a forgetful viewport. It only models aspects that are needed in the upcoming semantic definitions. For example, it forgets ordered association ends. In O we combine information on primitive-typed attributes with object references into a record. Another possibility would have been to denote meta level elements as records of merely primitive-typed values plus explicit object links as second kind of instances. Note, by the way, that in the UML semantics both styles of element presentations redundantly co-exist – see Fig. 4, diagram (v). The third option is to represent elements as pure nets of object identifiers with primitive values as leaves. By the way, we have discussed the latter option in form-oriented analysis as so-called parsimonious data model [37, 26]. Once more note, that the purpose of Sect. 2.6 is not to formalize UML semantics. It is merely about establishing notation for the existing meta level framework to be exploited in upcoming sections.

We have designed the value space as $V = \mathbb{P}(P \cup OID)$. As we have said, in the UML an attribute is, in general many-valued. Only, in the special case that the cardinality of an attribute is 1..1, an attribute is single-valued. The standard evaluation of an attribute in UML yields a bag, not a set. We have not designed our values in V as bags but as plain sets. This does not pose a problem, because bags can be formed by exploitation of object references. In the UML, properties can also evaluate to sequences. We assume that this sequencing can be modeled by an indexing mechanism on labels. We are interested in keeping our notation as reductionist as possible.

Meta-Object Levels We use $O_i \subset O$ to denote the set of all objects at meta-level M_i, the M_i-level objects for short.

Instances Given an M_i-level object o and an M_{i+1}-level object C, we use $o :: C$ to denote the fact that o is an instance of C as defined by the UML specification. Given an object $o \in O$ and a set of objects $M \subset O$, we use $o :: M$ to denote the

fact that there exists a $C \in M$ so that $o :: C$. Given sets of objects $M, N \subset O$, we use $M :: N$ to denote the fact that $o :: N$ for all $o \in M$. In case that $M :: N$ we also say that M is a instantiation of N.

Models We call a subset $\boldsymbol{m} \subset O$ of objects a model **iff** $\boldsymbol{m}$ is a partial function, i.e.:

$$\boldsymbol{m} \in OID \rightarrow \bigcup_{L' \in \mathbb{F}(\mathfrak{L})} (V_l)_{l \in L'} \tag{5}$$

Given a model $\boldsymbol{m} \subset O$ we say that $\boldsymbol{m}$ is a model at level i, or M_i-level model for short **iff** for all $o \in \boldsymbol{m}$ we have that $o \in O_i$. Given models $M, N \subset O$ we say that M is a model of N **iff** $M :: N$.

Value Identity Next, we define *value identity* of objects with respect to given models. Given models $\boldsymbol{m}$, $\boldsymbol{n}$, an object $o \in \boldsymbol{m}$ with $o = \langle oid \mapsto \langle i \mapsto x_i \rangle_{i \in I} \rangle$ and an object $p \in \boldsymbol{n}$ with $p = \langle pid \mapsto \langle j \mapsto y_j \rangle_{i \in I} \rangle$, we define o and p to be value-identical, denoted by $o \equiv p$ **iff** for all attribute labels $i \in I$ we have that:

$$\begin{array}{ll} (i) & x_i \not\subseteq OID \Rightarrow (x_i = y_i) \\ (ii) & x_i \subseteq OID \Rightarrow \big(\exists\, \beta : x_i \leftrightarrow y_i \;.\; \forall\, x' \in x_i \;.\; \boldsymbol{m}(x') \equiv \boldsymbol{n}(\beta(x'))\big) \end{array} \tag{6}$$

The definition of $\equiv$ is a partial specification only. It is only defined for objects that share the same set of labels I. It is only complete for well-typed and at the same time identically typed pairs of objects. This does not harm, because in the sequel we only work with well-formed models. Value identity can be characterized as identity up to exploited object references. The abstraction from concrete object references is exactly what is achieved by the bijection β in (6). In terms of programming languages, e.g., in Java terminology, value identity results from deep copying or cloning an object net.

Meta Meta Model Embedding We define the *embedding* of the UML meta meta model into the UML meta model $\iota : \mathfrak{M}_3 \hookrightarrow \mathfrak{M}_2$ by $\iota = \{(x, y) \mid x \equiv y\}$ – see also Fig. 3.

Standard Notation for Functions For the sake of completeness, we recap some standard notation for functions. Given a function $f : A \rightarrow B$, we denote the *lift* of f by $f^\dagger : \mathbb{P}(A) \rightarrow \mathbb{P}(B)$, which is defined as usual. Given a function $f : A \rightarrow B$ we denote the reversal, as usual, by $f^{-1} : B \rightarrow \mathbb{P}(A)$.

Meta model reification Next, we introduce the meta model reification operator Φ. First, we define the set of all meta model reification operators $\boldsymbol{\Phi}$ as the set of embeddings $\phi : \mathfrak{M}_2 \hookrightarrow O_1$ for which it holds true that (i) $\phi(\mathfrak{M}_2)^\dagger :: \mathfrak{M}_2$ and (ii) for all $m \in \mathfrak{M}_2$ it holds true that $\phi(m) \equiv m$. Then, we define Φ as an

arbitrary but fixed element of $\boldsymbol{\Phi}$, i.e., $\Phi \in \boldsymbol{\Phi}$. In the sequel, we refer to Φ as *the* meta model reification operator. Note, that $\phi(m) \equiv \phi'(m)$ for all $\phi, \phi' \in \boldsymbol{\Phi}$ and $m \in \mathfrak{M}_2$. Furthermore note, that $\mid \mathfrak{M}_2 \mid = \mid \phi^\dagger(\mathfrak{M}_2) \mid$, because ϕ is an embedding. This means, that all $\phi, \phi' \in \boldsymbol{\Phi}$ can be characterized as identical up to exploitation of object references. This explains, why it makes sense to define ϕ as an arbitrary but fixed selected element of $\boldsymbol{\Phi}$. In any case, formally, the definition based on a selection is well-defined.

Model reification On the basis of the meta model reification operator Φ we introduce the model reification operator Ψ. We define the set of model reification operators $\boldsymbol{\Psi}$ as the set of embeddings $\psi : O_1 \hookrightarrow O_0$ for which it holds true that, given any M_1-level model $\boldsymbol{m} \subset O_1$, it holds that (i) $\psi(\boldsymbol{m})^\dagger :: \Phi(\mathfrak{M}_2)$ and (ii) for all $m \in \boldsymbol{m}$ it holds true that $\psi(m) \equiv m$. Again, we define Ψ as an arbitrary but fixed element of $\boldsymbol{\Psi}$, i.e., $\Psi \in \boldsymbol{\Psi}$. In the sequel, we refer to Ψ as *the* model reification operator. Informally, the effect of the meta model reification Φ is to copy the meta model to the model level, whereas the effect of the model reification Ψ is to copy the *user user* model to the object level. Take a look at Figs. 3 and 4 for a visualization of how this actually works.

Further Notational Issues We model bags as functions to the ordinals, i.e., given a set T, we model the bags $Bags(T)$ of T as $Bags(T) = T \to \mathbb{N}_0$. Given a set T, we model the sequences $Seq(T)$ of T as indexed sets $(s_i)_{i \in \{1,..,n\}}$ over T with respect to a starting fragment $1, .., n$ of the ordinals. We define the length of a sequence as $\#((s_i)_{i \in \{1,..,n\}}) = n$. We use also $\lambda\, i \in \{1, .., n\}.s(i)$ to denote a sequence in $Seq(T)$.

2.7 Reification for Constraint Languages

A straight-forward approach to extend OCL by introspective and reflective features was to rewrite its syntax and semantics by doubling terms for the different levels of the meta-level architecture. Instead, we choose an economically approach that allows us to let the semantics of OCL almost untouched. We will have to give well-formedness rules and semantic specifications only for the newly added, genuine OCL_R reflection expressions. We achieve this by preparing the M1-level with a reified version of the UML meta model and the M0-level with a reified version of the user model – see Fig. 3.

The operator Φ reifies the UML meta model at level M1. Basically, this reification amounts simply to copying the UML language specification as a class diagram to the user level. This is immediately possible because of the bootstrap approach of the UML specification, i.e., because the UML meta model is specified in a core language that is itself a part of the UML language. Intuitively, we can say that we use the operator Φ to copy the UML meta model and add it to the user-defined model at level M1. Actually, the definition of Φ as provided in Sect. 2.6 is completely declarative. We have defined the set of object references O as an abstract data type. We keep O completely opaque, i.e., we do not define

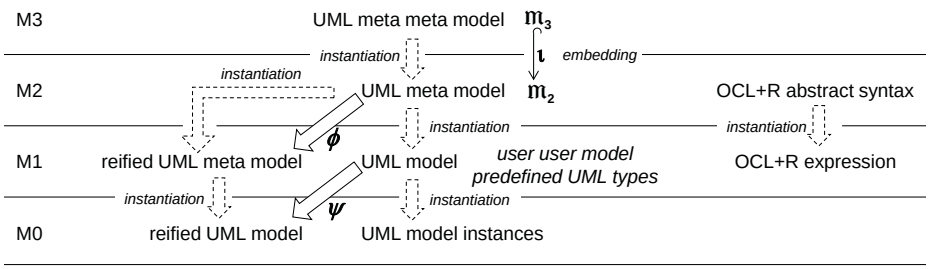

Fig. 3: Extending OCL with reification and reflection.

operations for the creation of object handles or the construction of objects. The value identity $\equiv$ that we have defined in (6) is a structural equality of object nets up to object references. Now, also the definition of Φ is free from concrete object construction mechanisms. We can assume the existence of Φ and therefore all semantics definition in this section are founded, in particular, the typing rules. If you find it helpful, you can think of the act of selecting and fixing an arbitrary ϕ from Φ as the act of copying model $\mathfrak{M}_2$ to level M1.

Each UML meta model expression is therefore immediately a correct M1-level model expression. This fact is also indicated by the embedding $\iota : \mathfrak{M}_3 \hookrightarrow \mathfrak{M}_2$ of the UML meta meta model into the UML meta model. Figure 3 shows the overall scenario of reification and reflection with OCL, whereas Fig. 4 gives a concrete example, based on a small cutout of the UML superstructure specification and a tiny user model. After the addition of the reified meta model to level M1 it is actually really a part of the user model. This fact eases the introduction of new reflective features to the OCL. However, usually we want to distinguish the reified meta model from the model that is actually created by the M1-level user modeler for its genuine purpose, e.g., domain modeling, system analysis, system design, and so forth. Henceforth, we call this part of the user model the *user user* model in cases where disambiguation seems to be important – see Fig. 4. The reification of the meta model data has to be understood as a semantic device, i.e., a means to declare the semantics of the extended language OCL_R. Therefore, by definition, there is no conflict with other software artifacts. The target of this article is not to achieve a particularly smart constraint language – whatsoever the criteria might be with respect to this. We add reflection to a constraint language for conceptual purposes. Ease and preciseness of the semantics are the rationales of the proposal. We are interested in the possibility of introducing reflection to object constraint languages in general. The resulting reflective language is interesting in its own right, but is not the ultimate goal.

With the reification of the UML meta model at level $M1$ we are prepared for introspective access. Given a meta model type, i.e., an $\mathfrak{M}_2$ type T, we use the concrete syntax $\langle T \rangle\!\downarrow$ to denote its reification at level M1. The $\langle \_ \rangle\!\downarrow$ notation is needed to distinguish user-defined types from reified meta model types. For example, if you want to model the national school system you might want to

have a class *Class* in your model, and this class must not come into conflict with the reified meta model type *Class*. With UML, this disambiguation of types is not only an issue of the concrete syntax but also an issue of the abstract syntax. According to the UML superstructure, the name of a named element allows to identify the element unambiguously – see [76] §7.3.34. This means that the UML offers not a completely abstract modeling backbone. Therefore, the concrete syntax $\langle T \rangle\downarrow$ stands for opening of a namespace. In practice, we can get rid of the extra notation. We can simply assume that the namespaces of user-defined and reified types are separated and use names T of reified types $\langle T \rangle\downarrow$ without harm. Nevertheless, in this article we stay with the notation $\langle \_ \rangle\downarrow$ for reasons of preciseness and clarity.

Now, we step further by reifying the *user user* model at level M0. Because of Φ, the appropriate classes of the reified UML meta model are available for this purpose at level M1. The operator Ψ re-instantiates each model element $e_1 :: e_2$ as the value identical model element $\Psi(e_1) :: \Phi(e_2)$. What we have achieved now is full introspective access onto the user-defined types, even without extension of the OCL syntax. Again, we use the notation $\langle T \rangle\downarrow$ to denote the reification of a user-defined type T, i.e., a type of the *user user* model. Note, that we overload the notation $\langle \_ \rangle\downarrow$ to denote both Φ- and Ψ-reifications.

See once more, how the copying mechanism of the reification operators Φ and Ψ work in Fig. 4. The copying step from sub diagram (*iii*) to sub diagram (*v*) seems to unfold the diagram (*iii*). However, it does not. Diagrams (*iii*) and (*v*) are just alternative visualizations of the same, i.e., value identical, object net, where diagram (*v*) is of course more detailed. The usual class diagram notation of (*iii*) is convenient for us, in particular, if we want to conceive it in its role as an O_1-level model for the instantiation of M_0 level models like the sub diagram (*vi*). However, diagram (*iii*) is an object net, and in its role as an instance of the UML meta model $\mathfrak{M}_2$ we would perhaps want to perceive it rather as detailed as visualized in (*v*).

2.8 Reflection for Constraint Languages

This section provides some examples of OCL_R expressions and an informal description of their semantics. In Sect. 3 we show how to give precise semantics in terms of type derivation rules and value specifications. With the meta model and model reification operators Φ and Ψ in Sect. 2.7 we have already achieved full introspective access to the *user user* model. However, yet the crucial step is missing, i.e., gaining fully reflective access onto model elements of the *user user* model via the reified data. What is missing is a means of materialization or re-materialization of modeling elements, i.e., of reflection in the narrow sense – see Table 1 once more. Many interesting constraints are yet not possible to write. To get the point, we will look at a series of example constraints. The examples serve merely as demonstration of the OCL and OCL_R mechanics. They are not meant to present examples of domain knowledge. In later sections we will see and discuss many exploitations of the reflective features that we have added to the constraint language. For the purpose of easy reference, we have

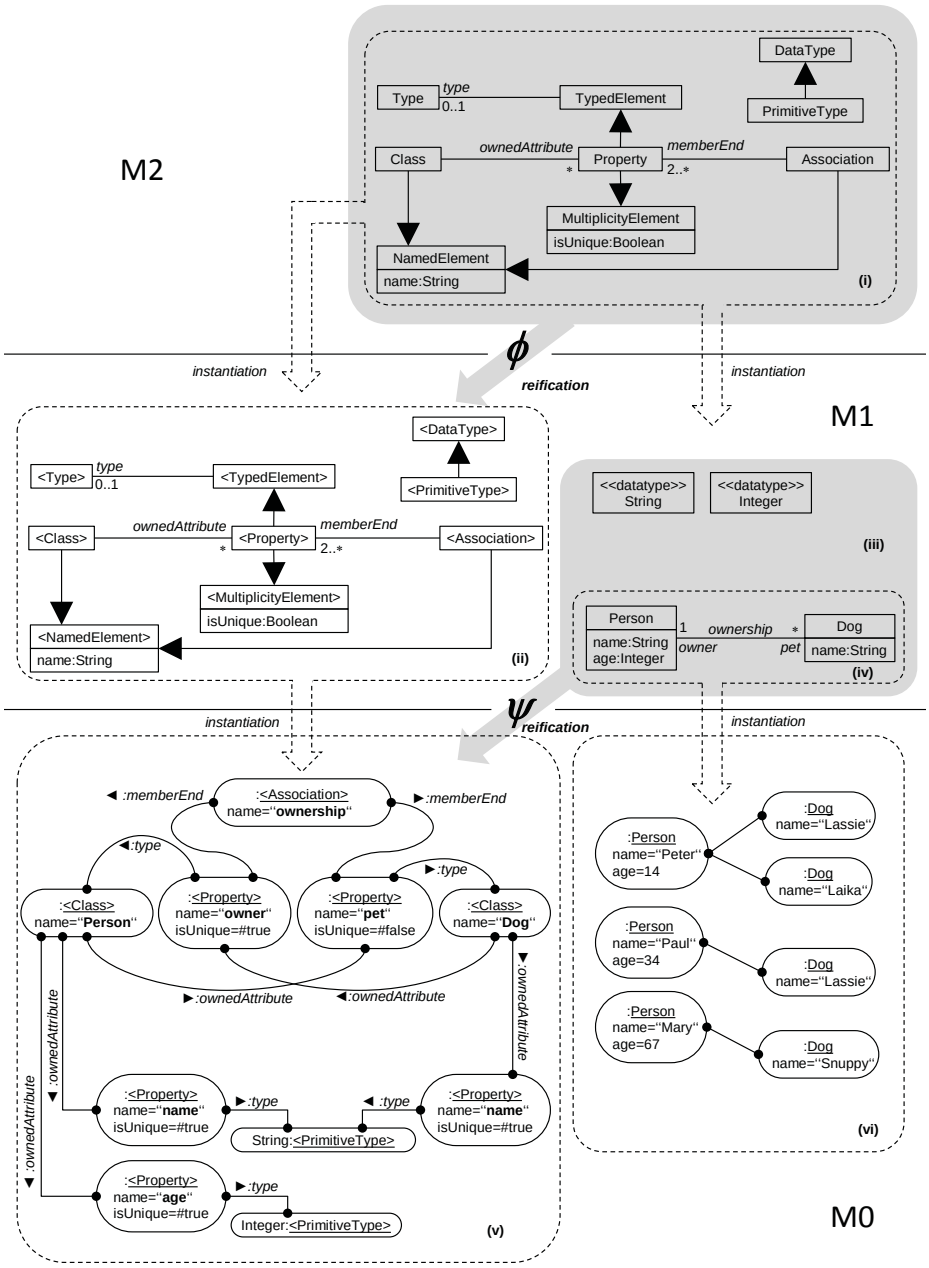

Fig. 4: The OCL_R reification mechanism

added a crucial chunk of the UML meta model in Appendix A, a specification of the OCL type system in Appendix B and a crucial chunk of the OCL abstract syntax specification in Appendix C. For many of the upcoming examples it will be helpful to switch between the text and these Appendices back and forth. We start with the following, correct constraint that exploits the reified data[1]:

$$\langle Class \rangle\downarrow .allInstances \rightarrow forAll(ownedProperty \rightarrow asSet \rightarrow size \leqslant 20) \tag{7}$$

Constraint (7) evaluates to true if all *user user* model classes have at most twenty properties. Against of the background of today's established modeling practice it is fair to say that (7) is an example of a real meta level constraint. It does not constrain the object during system evolvement time, but the modeler during modeling time.

Let us have a look at another constraint example:

$$\begin{array}{l} \langle Class \rangle\downarrow .allInstances \\ \rightarrow select(name = \texttt{"}Person\texttt{"}).ownedProperty \\ \rightarrow select(name = \texttt{"}pet\texttt{"}).type \\ \rightarrow includes(name = \texttt{"}Dog\texttt{"}) \end{array} \tag{8}$$

The constraint in (8) checks whether the class *Person* is associated, via a role *pet* to a class *Dog*. The semantics of the constraint (8) is equal to the semantics of the following usual OCL constraint that work without the new reification capabilities:

$$\textbf{context } Person \textbf{ inv}: pet.oclIsTypeOf(Dog) \tag{9}$$

See, how constraint (9) immediately queries the property *pet*, whereas (8) must navigate the two additional links *ownedProperty* and *type* of UML meta model to reach the target *Dog*. Now, let us have a look at the following invalid, i.e., ill-typed, constraint expression:

$$\begin{array}{l} \langle Class \rangle\downarrow .allInstances \rightarrow select(name = \texttt{"}Person\texttt{"}). \\ allInstances \rightarrow forAll(age \geqslant 40) \end{array} \tag{10}$$

Intuitively, constraint (10) has the following semantics as (1), i.e.:

$$\textbf{context } Person \textbf{ inv}: age \geqslant 40 \tag{11}$$

In OCL_R we will be able to write constraints like (10) in due course, after the introduction of an appropriate reflection notation. However, for the time being, constraint (10) is ill-typed. The problem is the second *allInstances*-property. To see why, consider the following type derivation. The expression

[1] Note, that we feel free to drop brackets from OCL operation calls whenever the paramter list is empty, e.g., we write $s \rightarrow asSet \rightarrow size$ instead of $s \rightarrow asSet() \rightarrow size()$.

$\langle Class\rangle\downarrow$ has type *OclType*. As part of the limited meta data access capabilities of OCL, it is possible to apply the method *allInstances* to this expression. The expression $\langle Class\rangle\downarrow .allInstances$ has type $Set(\langle Class\rangle\downarrow)$. The expression $\langle Class\rangle\downarrow .allInstances \rightarrow select(name = \texttt{"}Person\texttt{"})$ again has type $Set(\langle Class\rangle\downarrow)$, actually, it evaluates to the one-set element consisting of exactly the reified *Person* object. Now, when we try to invoke the *allInstances* method to this expression, we provoke a type error, because *allInstances* can only be applied to terms of type *OclType*. The constraint in (10) simply does not adhere to the well-formedness rules of OCL. As an even simpler counter example, it is not possible to apply *allInstances* twice in a path expression like the following:

$$\langle Class\rangle\downarrow .allInstances.allInstances \tag{12}$$

Again, the expression in (12) is not well-typed. Again, intuitively, constraint expressions (12) has a semantics. It is intended to mean the set of all instances of all classes of the *user user* model. Again, in OCL_R we will be able to write constraint expressions like (12) in due course.

2.9 Full Reflection Capabilities

Now, we introduce a reflection construct $\langle\_\rangle\uparrow$ as the crucial extension to OCL. Based on this notation, we can give a correct version of constraint (10) as follows:

$$\begin{aligned}&(\!|\langle\langle Class\rangle\downarrow .allInstances \rightarrow select(name = \texttt{"}Person\texttt{"})\rangle\uparrow|\!).\\&allInstances \rightarrow forAll(age \geqslant 40)\end{aligned} \tag{13}$$

Constraints (13) checks whether each instance of the class *Person* has an attribute value over 40 for its attribute *age*, i.e., it is equal to constraint (11). Informally, the semantics of the reflection construct is the reversal of reification. In (13) the reflection receives the one-element set containing the reified *Person* object and yields the one-element set containing the *Person* meta-object it has been reified from. More precisely, the reflection construct in (13) turns an expression of type $Set(\langle Class\rangle\downarrow)$ into an expression of type $Set(OclType)$. See how this works in the following example type derivation. Then, as usual, $e : T$ means that a sub expression e has type T – see also Appendix B as a reference for OCL types:

$$\underbrace{\underbrace{(\!|\underbrace{\langle\underbrace{\underbrace{\underbrace{\langle Class\rangle\downarrow}_{(i):OclType} .allInstances}_{(vi):Set(\langle Class\rangle\downarrow)} \rightarrow select(\underbrace{\underbrace{\underbrace{self}_{(ii):\langle Class\rangle\downarrow} .name}_{(vii):String} = \underbrace{\texttt{"}Person\texttt{"}}_{(iii):String}}_{(ix):Boolean})}_{(x):Set(\langle Class\rangle\downarrow}\rangle\uparrow}_{(xi):Set(OclType)}|\!)}_{(xii):OclType} .\ allInstances}_{(xiii):Set(Person)} \tag{14}$$

The result of the reflection (xi) in (14) has type $Set(OclType)$. Unfortunately, with standard OCL this result cannot be immediately exploited in a property call $o.p$. OCL requires that a property can only be applied to an object of a single classifier – the OCL specification states [71]: *A PropertyCallExpression is a reference to an Attribute of a Classifier defined in a UML model. It evaluates to the value of the attribute.* We have two options to deal with this. We can extend OCL so that it can also deal with the application of a property to object of several classes and we will see in due course that this is easily possibly. As the second option, and this is what we see in the current example, is to introduce a new operator to OCL that turns a one-element set into the contained element. With $(\!|M|\!)$ we denote exactly this operation. Note, that $(\!|M|\!)$ is only partially defined, i.e., it is defined only for one-element sets. With respect to semantics, the solution based on $(\!|M|\!)$ is conservative, i.e., it can be added to OCL without changing the existing semantics of the OCL.

With the reflection operator so far, we have added substantially to the expressive power to OCL. However, to earn the full potential reflective power, we need to develop a means to apply a property to a set of objects that are instances of completely arbitrary classes. In OCL we are already used to apply an attribute to a set of objects yielding an object set as result. In general, there is no reason why we should not apply an attribute to objects of more than one class. Have a look at the following example that introduces some ad-hoc concrete syntax for the special case of a fixed number of classes:

$$\textbf{context } \{Person, Dog\} \textbf{ inv}: self.age \geqslant 5 \tag{15}$$

Please note, that it is not our intention to introduce new concrete syntax. There is no need for us to do so. However, possibilities to address properties possessed by objects of more than one arbitrary type arise in OCL_R as a result of its design. Concrete syntax like the one in (15) has only the purpose to analyze the semantics of such scenarios for us. Of course, in (15) we assume that both the class *Person* and the class *Dog* define Integer attributes *age*. Note, however, that the Integer attributes *age* in *Dog* and *Person* are not required to be inherited from a common supertype of *Person* and *Dog* – this is what we meant with *completely arbitrary* classes above. Fortunately, we do not need to change the syntax of OCL to make expression like (15) possible. A property call expression has another OCL expression as its source. In general, it is possible that OCL expressions have type Set(OclType) – see Appendix B. So it is a self-restriction of the OCL semantics [71] to allow only for the application to a single type. The semantics of an expressions like (15) is immediately clear. It collects the values of attributes of all the objects of different type, not only of the objects of a single type. We will show how to give precise semantics to this later. Now, we generalize OCL_R property calls further to cases, in which an arbitrary number of classes is dynamically determined. For example, the following constraint evaluates the *age* attribute for all objects of a class model. Again, the value of such an expression

is defined only, if all classes in the current user-model possess an *age* attribute of correct type:

$$\langle\langle Class\rangle\downarrow .allInstances\rangle\uparrow .allInstances \rightarrow forAll(age \geqslant 40) \tag{16}$$

Next, we introduce the reification notation $\langle\_\rangle\downarrow$ also for *user user* model types. It hands over the reification operator Ψ to the modeler. The following example constraint, which is equivalent to (11), including the crucial type derivation shows how this works:

$$\begin{array}{l}\langle\langle Class\rangle\downarrow .allInstances\\ \rightarrow select(\underbrace{\underbrace{self}_{(i):\langle Class\rangle\downarrow} = \underbrace{\langle Person\rangle\downarrow}_{(ii):\langle Class\rangle\downarrow}}_{(iii):Boolean})\rangle\uparrow .allInstances \rightarrow forAll(age \geqslant 40)\end{array} \tag{17}$$

So far, each OCL expression of type $\langle Class\rangle\downarrow$ or collection type $Set(\langle Class\rangle\downarrow)$ can be made subject to reflection. In general, we will expand reflection to all kind of reified data. Let us have a look at the follwing example, which is again equivalent to (11):

$$\begin{array}{l}Person.allInstances \rightarrow forAll(\\ \quad self.\langle\\ \quad\quad (\!|\\ \quad\quad\quad \langle Class\rangle\downarrow .allInstances\\ \quad\quad\quad \rightarrow select(\langle Person\rangle\downarrow).ownedAttributes\\ \quad\quad\quad \rightarrow select(name = \texttt{"}age\texttt{"})\\ \quad\quad |\!)\\ \quad \rangle\uparrow\\ \quad \geqslant 40\\ )\end{array} \tag{18}$$

In (18), the constraint expression inside the reflection construct yields a reified property. After application of the reflection, this property can then be called. See the following type derivation for the crucial part of (18):

$$\boldsymbol{P} \equiv_{DEF} \underbrace{\langle Class\rangle\downarrow .allInstances \rightarrow select(\langle Person\rangle\downarrow)}_{(ii):Set(\langle Class\rangle\downarrow)}$$

$$\underbrace{\underbrace{self}_{(i):Person} .\langle \overbrace{(\!| \underbrace{\boldsymbol{P}.ownedAttributes}_{(iii):Set(\langle Property\rangle\downarrow)} \rightarrow select(name = \texttt{"}age\texttt{"}) |\!)}^{(vi)}_{(\boldsymbol{v}):\langle \boldsymbol{Property}\rangle\downarrow} \rangle\uparrow}_{(iv):Set(\langle Property\rangle\downarrow)}}_{(vii):Integer} \tag{19}$$

Technically, the property name p in a property call expression $o.p$ itself is not a proper OCL expression, in the sense that it does not have a type. This does not harm in the type derivation of (19). It is exactly the reflection construct that opens a context for typed expressions. See how the type of (v) in (19) is immediately consumed by the type derivation with rule (36) from Sect. 3, i.e., how the typing of (vi) is not needed in the type derivation.

As a next step, we can also generalize the semantics of property call expressions further, so that the application of a set of properties to a class or a set of classes becomes possible. See the following example showing again some ad-hoc syntax, with obvious semantics:

$$\textbf{context}\ \{Person, Dog\}\ \textbf{inv}\colon self.\{age, weight\} \geqslant 0 \tag{20}$$

In OCL_R we can exploit such an extension to the OCL semantics in an expression like the following:

$$\begin{array}{l} \langle\langle Class\rangle\downarrow .allInstances\rangle\uparrow .allInstances.\langle \\ \quad \langle Class\rangle\downarrow .allInstances.ownedAttributes \\ \quad \rightarrow select(type = Integer) \\ \rangle\uparrow .sum \end{array} \tag{21}$$

The constraint (21) is well-formed with respect to all *user user* models. This is ensured by the clause $select(type = Integer)$ which ensures that only type-correct property calls occur. The expression in (21) is a solution to expression (7) in the example list in Sect. 2.1, i.e., the sum of all Integer attributes of all objects of all classes. As we have mentioned before, we are free to omit all the special syntax for reification, reflection and also element picking, i.e., $\langle\_\rangle\uparrow$, $\langle\_\rangle\downarrow$ and $(\!|\_|\!)$ in the application of the OCL_R , unless we do not need it to for the disambiguation of expressions. See how this simplifies expressions, e.g., for the expression (21):

$$\begin{array}{l} Class.allInstances.allInstances.(\\ \quad Class.allInstances.ownedAttributes \\ \quad \rightarrow select(type = Integer) \\ ).sum \end{array} \tag{22}$$

Nevertheless, we stay with the explicit notation throughout the rest of the article, as we have said before, for the reason of preciseness.

3 On The Precise Semantics of OCL_R Reflection

The purpose of this section is to show how to give precise semantics to an object-oriented constraint language. In the definitions of this section we make extensive use of notation introduced in Sect. 2.6 and heavily rely on the concepts defined earlier, e.g., the reification and reflection operators Φ and Ψ. We define the necessary well-formedness rules as strict augmentations to the existing notion of UML and OCL type correctness.

3.1 Typing Notation and Semantic Bracketing

Given a UML, OCL or OCL_R expression e and type T, the *typing* $e : T$ expresses that e is well-typed and has type T. We use further usual notation from the type system community [17, 80, 59] to express well-typing. The statement $\vdash\ e : T$ holds if the typing $e : T$ has been derived, i.e., has been proven. Typing rules are expressed in the following manner:

$$\frac{\vdash e_1 : T_1 \ldots \vdash e_n : T_n \quad C_1 \ldots C_m}{\vdash e' : T'} \tag{23}$$

Given that we have already derived typings $e_i : T_i$ and further conditions C_i hold true, a typing rule of kind (23) allows to derive typing $\vdash e' : T'$. There are no other typings than those that can be derived by typing rules. Typing rules are instances of well-formedness rules.

Furthermore, we use so-called semantic bracketing to define the value of expressions. Given an expression M we use $[\![M]\!]$ to denote its value. With semantic bracketing we mean the natural declarative technique to define the semantics as a recursive function along the structure of abstract syntax trees, i.e., the semantics of an expression $[\![e\ e_1 \ldots e_n]\!]$ as a value $\boldsymbol{E}([\![e_1]\!], \ldots, [\![e_n]\!])$ with $\boldsymbol{E}$ being a sufficiently precise semantic description. It is important to understand, that all definitions in this section, including typing rules and semantic equations are always in terms of abstract syntax trees, even if we use concrete syntax to denote them. Here, we again rely into the semantics of OCL. We assume a sufficiently precise semantics for OCL expressions that is available in our semantics specification, i.e., we assume that $[\![e]\!]$ is defined if e is a pure OCL expression.

3.2 Typing Rules and Values

Fig. 5 shows the OCL_R meta model. Elements of the OCL are shown in grey, whereas the new language constructs are shown in black – compare to the OCL specification in Appendix C. The meta model elements of all of the three new language constructs implement the *OCLExpression* interface, i.e., they are proper OCL expressions that also receive types. A reflection expression refers to another OCL expression as its reflected expression. A reification expression refers to a type expression as its reified type. The types of OCL_R are the same as the types for OCL v.2.0 – see Appendix B Fig. 12.

We consider the semantics for three kinds of expressions that cover the full range of OCL_R semantics, i.e., type expressions, property call expressions and enumeration literal expressions. As explained in Sect. 2.8 the reification of the crucial $\mathfrak{M}_2$-model element *Class*, i.e., $\langle \mathit{Class} \rangle\!\downarrow$ has the type *OclType*. We repeat this as the following rule:

$$\frac{}{\vdash \langle \mathit{Class} \rangle\!\downarrow : \mathit{OclType}} \tag{24}$$

Furthermore we now, by the definition of Φ and Ψ that for all types t of the *user user* model, i.e., the data types of an M1-level model, we have that:

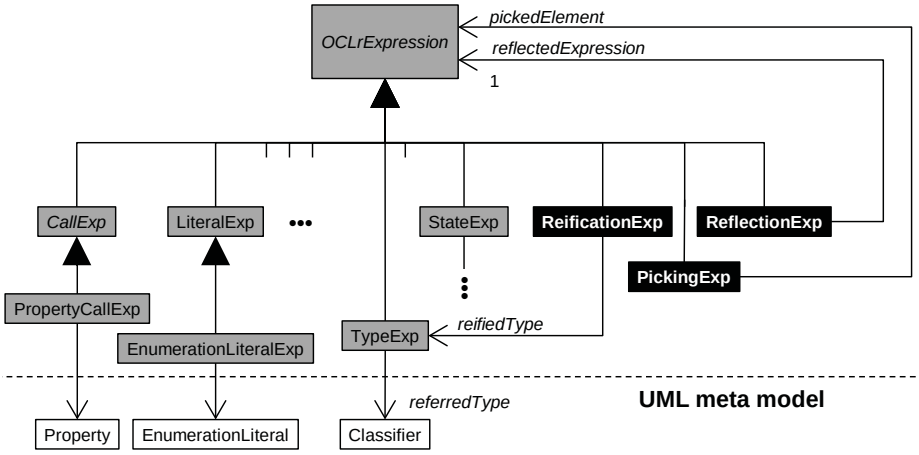

Fig. 5: Abstract Syntax of OCL_R

$$\frac{\vdash t : \mathit{OclType} \quad t :: \mathit{Class}}{\vdash \langle t \rangle\downarrow : \Phi(\mathit{Class})} \tag{25}$$

Now, an expression of a reified data type, i.e., a reified data type expression, yields an *OclType*-expression after reflection. Reflection recovers the M1-level value of reified data:

$$\frac{\vdash e : \Phi(\mathit{Class})}{\vdash \langle e \rangle\uparrow : \mathit{OclType}} \tag{26}$$

The value of a reified data type expression $\langle e \rangle\uparrow$ is defined as:

$$[\![\langle e \rangle\uparrow]\!] = [\![\Psi^{-1}([\![e]\!])]\!] \tag{27}$$

In general, expressions rather have a a type of the form $C(\Phi(\mathit{Class}))$ for a collection C than merely the type $\Phi(\mathit{Class})$. Examples for expressions that have $\Phi(\mathit{Class})$ as type are the iterator variables in OCL loop expressions. A further example is the result of correctly applying an element picking operation. The element picking operation can be applied to a collection of any type, including reified M2-level types. We define the typing and the value of the picking operator $(\!|\_|\!)$ for all kinds of collection C, i.e., *Set*, *Bag* and *Sequence* as follows:

$$\frac{\vdash e : C(T) \quad |[\![e]\!]| = 1}{\vdash (\!| e |\!) \ : T} \tag{28}$$

$$[\![(\!| m |\!)]\!] = \begin{cases} [\![x]\!] & , \exists\, x.m = \{x\} \\ \bot & , \mathit{else} \end{cases} \tag{29}$$

Now, we generalize typing and values of reified data type expressions for the case of collection types. For all kinds of collections C, i.e., *Set*, *Bag* and *Sequence*, we have that:

$$\frac{\vdash e : C(\Phi(Class))}{\vdash \langle e \rangle\uparrow : C(OclType)} \tag{30}$$

With respect to the value of reified data type expressions we need to distinguish three cases now, i.e., sets, bags, and sequences. In case that $e : Set(\Phi(Class))$, we define the value of $\langle e \rangle\uparrow$ as follows:

$$[\![\langle e \rangle\uparrow]\!] = [\![\ (\Psi^{-1})^{\dagger}\big([\![e]\!]\big)\ ]\!] \tag{31}$$

We know that $[\![\langle e \rangle\uparrow]\!]$ can be written differently as $\{t : OclTpye \mid \Psi(t) \in [\![e]\!]\}$. In case that $e : Bag(\Phi(Class))$ we can define the value of $\langle e \rangle\uparrow$ as follows:

$$[\![\langle e \rangle\uparrow]\!] = \lambda\, t : OclTpye\ .\ [\![e]\!](\Psi(t)) \tag{32}$$

In case that $e : Sequence(\Phi(Class))$ we can define the value of $\langle e \rangle\uparrow$ as follows:

$$[\![\langle e \rangle\uparrow]\!] = \lambda\, i \in \{1, .., \#([\![\langle e \rangle\uparrow]\!])\}\ .\ \Psi^{-1}\big([\![e]\!](i)\big) \tag{33}$$

As a next step we specify OCL_R in case of enumeration types and enumeration literals. We define that, for all types t of the *user user* model, we have that:

$$\frac{\vdash t : OclType \quad t :: Enumeration}{\vdash \langle t \rangle\downarrow : \Phi(Enumeration)} \tag{34}$$

$$\frac{\vdash e : \Phi(EnumerationLiteral)}{\vdash \langle e \rangle\uparrow : EnumerationLiteral} \tag{35}$$

The value specification for enumeration literal expressions is identical to the one in case of type expression, i.e., equation (27).

3.3 Semantics of Property Call Expression

We turn to property call expressions now. First, we consider one of the simplest cases that (i) a reflected property is called on a single object, (ii) the reified property results into a single object and (iii) the property call results into a single object of user-defined type. In this case typing is defined for all user defined types T_2 as follows:

$$\frac{\vdash o : T_1 :: Class \quad p : \Phi(Property) \quad p.class = \Psi(T_1) \quad p.type = T_2}{\vdash o.\langle p \rangle\uparrow : \Psi^{-1}(T_2)} \tag{36}$$

In the scenario prescribed by (36) we define the value of a property call expression $o.\langle p \rangle\uparrow$ as follows:

$$[\![o.\langle p \rangle\uparrow]\!] = [\![\ o.\Psi^{-1}([\![p]\!])\ ]\!] \tag{37}$$

For a full specification of OCL_R expressions we had to define a combinatorial number of different cases, depending on the result type of the property, the result type of reified operations and the question of whether the operation is applied to a single object or a collection of objects. Then, each of the involved types can be, combinatorial, a primitive type or a collection, and again, each collection, also the collection of objects, can be a set, a bag, or a sequence. We look at only one further case, which is a particular complex one, i.e., the case that all of the aforementioned components can be collections. For all kinds of collections C_1, C_1, C_3, i.e., *Set*, *Bag* or *Sequence*, we establish the following typing rule:

$$\frac{\vdash o : T_1 :: C_1(\mathit{Class}) \quad p : C_2(\Phi(\mathit{Property})) \quad p.\mathit{class} = \Psi(T_1) \quad p.\mathit{type} = C_3(T_2)}{\vdash o.\langle p \rangle\uparrow : (C_1 \oplus C_2 \oplus C_3)(\Psi^{-1}(T_2))} \tag{38}$$

The definition of the typing rule (38) relies on a combinator $\oplus$ for collection constructors. This combinator is defined in Table 2 in Appendix B.2. The OCL approach is that nested collections are always and automatically flattened. For example, a set of sets is turned into a set, a bag of bags is turned into bag and so forth. The definition of the $\oplus$ combinator fulfills the standard definition of OCL collection flattening in [71, 74]. First, we handle the case that all of the involved components yields sets, i.e., $C_1(\mathit{Class}) = \mathit{Set}(\mathit{Class})$, $C_2(\Phi(\mathit{Property})) = \mathit{Set}(\Phi(\mathit{Property}))$ and $C_3(T_2) = \mathit{Set}(T_2)$. In this case, we know that $o.\langle p \rangle\uparrow$ has the type $\mathit{Set}(T_2)$ and we define the value of $o.\langle p \rangle\uparrow$ as follows:

$$[\![o.\langle p \rangle\uparrow]\!] = \{v \in T_2 \mid v : [\![o'.p']\!],\ o' \in o,\ \Psi(p') \in [\![p]\!]\} \tag{39}$$

Next, we handle the case that all of the involved components yield bags, i.e., $C_1(\mathit{Class}) = \mathit{Bag}(\mathit{Class})$, $C_2(\Phi(\mathit{Property})) = \mathit{Bag}(\Phi(\mathit{Property}))$ and $C_3(T_2) = \mathit{Bag}(T_2)$. In this case, we know that $o.\langle p \rangle\uparrow$ has the type $\mathit{Bag}(T_2)$ and we define the value of $o.\langle p \rangle\uparrow$ as follows:

$$[\![o.\langle p \rangle\uparrow]\!] = \lambda\, v : T_2\ .\ \sum_{o' \in o} \Big(\sum_{p' \in \Psi^{-1}[\![p]\!]} [\![o'.p']\!](v) \Big) \tag{40}$$

Note, that the sums in (40) are all well-defined, because all of the involved collections are finite. The scenario that the involved components yield bags, is the standard scenario in the OCL. We have started with the set scenario in (39) only for instructive purposes. We do not detail out further combinations $C_1 \oplus C_2 \oplus C_3$ of collection constructions.

4 Working with OCL_R

We will see OCL_R at work in Sects. 5 and 7 when we exploit it for the semantic investigation of power types in general and power types in UML in particular.

Before that, let us walk through the informal constraint examples that we have enumerated in Sect. 2.1 as constraints (1) through (9). Again, please have a look at the cutout of the UML meta model as provided by Fig. 11 in Appendix A throughout the examples. Example (1), i.e., the names of subclasses of a given type t can be expressed in OCL_R as follows:

$$\begin{array}{l} \langle \mathit{Generalization} \rangle\!\downarrow .\mathit{allInstances} \\ \rightarrow \mathit{select}(\mathit{general} = \langle t \rangle\!\downarrow).\mathit{specific}.\mathit{name} \end{array} \tag{41}$$

The subclasses of a given type t, i.e., example (2), follows immediately from (41) by dropping the last property call, i.e., the *name* navigation. Example (3), i.e., the attribute names of classes navigable via associations from a given type t can be expressed as follows:

$$\begin{array}{l} \langle \mathit{Class} \rangle\!\downarrow .\mathit{allInstances} \rightarrow \mathit{select}(c \mid \\ \quad \langle \mathit{Association} \rangle\!\downarrow .\mathit{allInstances} \rightarrow \mathit{exists}(\\ \quad\quad \mathit{memberEnd} \rightarrow \mathit{contains}(\langle t \rangle\!\downarrow) \\ \quad\quad \mathit{and} \\ \quad\quad \mathit{memberEnd} \rightarrow \mathit{contains}(c) \\ \quad) \\ ).\mathit{name} \end{array} \tag{42}$$

Example (4), i.e., all classes of the user model, turns out to be a most simple example that has been exploited already in many instances before. It is given by $\langle \mathit{Class} \rangle\!\downarrow .\mathit{allInstances}$. Consequently, example (6), i.e., the number of classes in the user model is given by $\langle \mathit{Class} \rangle\!\downarrow .\mathit{allInstances} \rightarrow \mathit{sum}$. Example (6), i.e., all classes of the user model that have no subclasses, is provided by:

$$\begin{array}{l} \langle \mathit{Class} \rangle\!\downarrow .\mathit{allInstances} \rightarrow \mathit{select}(c \mid \\ \quad \mathit{not} \\ \quad \langle \mathit{Generalization} \rangle\!\downarrow .\mathit{allInstances} \rightarrow \mathit{exists}(\\ \quad\quad \mathit{general} \rightarrow \mathit{contains}(c) \\ \quad) \\ ) \end{array} \tag{43}$$

The example (7) has already been solved as an example by (21) before. A test, whether all attributes of all objects of all classes are initialized, i.e., example (8) can be realized as follows:

$$\begin{array}{l} \langle \mathit{Class} \rangle\!\downarrow .\mathit{allInstances} \rightarrow \mathit{forAll}(c \mid \\ \quad \langle c \rangle\!\uparrow .\mathit{allInstances} \rightarrow \mathit{forAll}(o \mid \\ \quad\quad c.\mathit{ownedAttribute} \rightarrow \mathit{forAll}(a \mid \\ \quad\quad\quad o.\langle a \rangle\!\uparrow \neq \mathit{null} \\ \quad) \\ ) \end{array} \tag{44}$$

Fortunately, in OCL a *null* value of type *OclVoid* is available as defined in [71, 74]. This *null* value is exploited in (44) to test whether an attribute is initialized, where we assume that initialized attributes have a value different from *null*.

4.1 Getter and Setter Method Example

Next, we realize example (9), i.e., a test whether all attributes of all classes have setter- and a getter-methods. The following constraint (45) checks whether for each class and attribute X of type t there exist a setter-method and a getter-method of appropriate parameter signature, i.e., a method $setX(x : t)$ for some arbitrarily named input parameter and a method $getX() : t$:

$$
\begin{array}{l}
\langle Class \rangle\!\downarrow .allInstances \rightarrow forAll(c \mid \\
\quad c.ownedAttribute \rightarrow forAll(a \mid \\
\quad\quad c.ownedOperation \rightarrow exists(m \mid \\
\quad\quad\quad m.name = \texttt{"}set\texttt{"} + \texttt{"}a.name\texttt{"}\ and \\
\quad\quad\quad m.ownedParameter.size = 1\ and \\
\quad\quad\quad m.ownedParameter.direction = \#in\ and \\
\quad\quad\quad m.ownedParameter.type = a.type \\
\quad\quad)\ and \\
\quad\quad c.ownedOperation \rightarrow exists(m \mid \\
\quad\quad\quad m.name = \texttt{"}get\texttt{"} + \texttt{"}a.name\texttt{"}\ and \\
\quad\quad\quad m.ownedParameter.size = 1\ and \\
\quad\quad\quad m.ownedParameter.direction = \#return\ and \\
\quad\quad\quad m.ownedParameter.type = a.type \\
\quad\quad) \\
\quad) \\
)
\end{array}
\tag{45}
$$

Note, that the UML specification fixes that a method can have one and at most one return parameter. It would also be possible to completely specify the correct behavior of the methods based on the signature specification in (45) but we not detail this out here.

4.2 A Comparison with Genoupe Generative Programming

Reflective object-oriented constraint-writing is the natural counterpart to generative programming. Generative programming is another word for reflective programming. Generative programming gets its name from what we have called reflection in the narrow sense in Table 1, i.e., the step of turning reified data into code. Code generation is a particular operational viewpoint on reflection. It hints to a possible implementation strategy based on a pre-compilation phase for reflective features on top of an already existing programming language.

Now, as an instructive example, let us program the counterpart of the getter- and setter-example (45) in Sect. 4 in a reflective programming language. We choose our own reflective programming language Genoupe [33, 34, 58] for this purpose – see Listing 4.1.

Genoupe is an extension of the programming language C# with generative programming features. An important contribution of Genoupe is the definition and implementation of an extended notion of *generator type-safety*, which is,

Listing 4.1 Generation of Getter and Setter Methods with GENOUPE

```
public class GetterSetter (Type T) : @T@{

    @foreach(F in T.GetFields()) {

        public void @"set" + F.Name@ ( x : @F.FieldType@) {
            @F.Name@ = x;
        }

        public  @F.FieldType@ @"get" + F.Name@ {
            return @F.Name@;
        }

    }
}
```

however, not important for the consideration of the current example. For us, it is enough to understand how the generative features in Listing 4.1 work.

GENOUPE is extended by new, concrete syntax for meta programming. The special sign @ is used to introduce or embed some of the new meta-programming syntax. A pre-processing phase takes GENOUPE and generates plain C# code. In Listing 4.1 we implement a class *GetterSetter* parametric on a type parameter (*TypeT*). Then, with :@T@ we achieve that the generated *GetterSetter* class extends the actual parameter class and therefore inherits all of the fields of the actual parameter class. In GENOUPE we have access to all features of the C# reflection API. In Listing 4.1 we exploit the method $T.GetFields()$, that yields all fields of a type T, as well as the properties $F.Name$ and $F.FieldType$ with respective meaning. Then, the GENOUPE meta programming expression @$foreach(i \ldots)\{C(i)\}$ allows us to generate a piece of C# code for each instance of an iterator variable i. In Listing 4.1, we exploit @*foreach* to generate a getter and a setter method for each field of the actual parameter class.

5 Adequately Modeling of Sets of Sets

This section deals with the modeling of sets of sets of domain objects. Modeling sets of sets of objects is important, because it arises naturally in expert domains – see Fig. 6 for an example. Modeling of sets of sets is a classical topic [53] in the modelling community and has been discussed as modeling with power types [68, 69, 61, 47, 46]. It has also been discussed as multilevel modeling in the past [7].

Often, modeling a set of domain objects involves the specification of properties that are common to all objects of the investigated set. This means, that in domain modeling, we are, in general, also interested in the *intension* or *comprehension* of a set of objects rather than merely in its role as an *extension* of

a concept. We could introduce new terminology as has happened in the object-oriented community in the past. For example, we could call a set of objects together with its intension a *class*. There is no single commonly accepted definition of the notion of *intension* in linguistics and ontology. So, let me be more concrete. More concrete, we could say that objects have properties and that we have a special interest into a certain notion, let's call it, e.g., *domain object class*, which is a set of objects together with a specification of which properties are shared by all objects of this set, i.e., are equal for all objects of that set. We could than give this notion a name and *class* has been a usual candidate for this in the past. The problem is that *class* is also used for concrete programming and modeling language constructs which usually have a rather operational semantics. Concrete class constructs in programming and modeling languages have been designed, of course, with a notion of *domain object class* in mind. In order to avoid conflicts with the class terminology of programming and modeling language constructs, we could choose another name for *domain object classes*, e.g. *Class*, *domain class*, or simply *domain object class*. We do not. We simply talk about set of objects, set of sets of objects and so forth and point out, that in domain modeling, we have special interest in properties that are common to all objects of a given set. The treatment of *intensions* of sets of objects have been intensively studied in the in the modeling community and many crucial results has been achievd for : notation, terminology, modeling constructs, patterns, tools, semantical considerations etc. – see also Sect. 9 for examples. Therefore, we have chosen this as an example worth looking at to analyze with a reflective constraint language like OCL_R.

As an example, Fig. 6 shows the mammal hierarchy. A set of sets appears systematically at each level of the hierarchy, grouping objects of classes that reside at a lower level, i.e., breed, sub species, species, genera and so on. The running example in this article will be dogs and breeds from this hierarchy. The different sets of sets in the mammal hierarchy in Fig. 6 all follow the *type-object pattern* [53] of Johnson and Woolf. The set of set is called a type in the type-object pattern. A type in the type object pattern, i.e., an instance of the type class of the pattern, is not a type in the sense of object-oriented subtyping hierarchy; in particular, it is not a type of a concrete modeling language. A type in the type object pattern represents a *kind* or a *group* of objects, i.e., a set of set of objects. Furthermore, a type carries the attributes common to all of the objects that it groups together. In that sense, again, a type of the type-object pattern is the *intension* of a set of objects.

The aim of this section is to discuss ways of adequately modeling sets of sets of domain objects. We show that it is possible to model sets of sets in terms of basic object-oriented modeling constructs plus appropriate constraints. Subtyping and subclassing mechanisms help in modeling sets of sets, but we will see that even most basic notions of object-oriented modeling, i.e., classes and enumeration types, are already sufficient for giving appropriate models of sets of sets, as long as the necessary, generic constraints are provided. There is no necessity to introduce new modeling language features, like a concept of set or

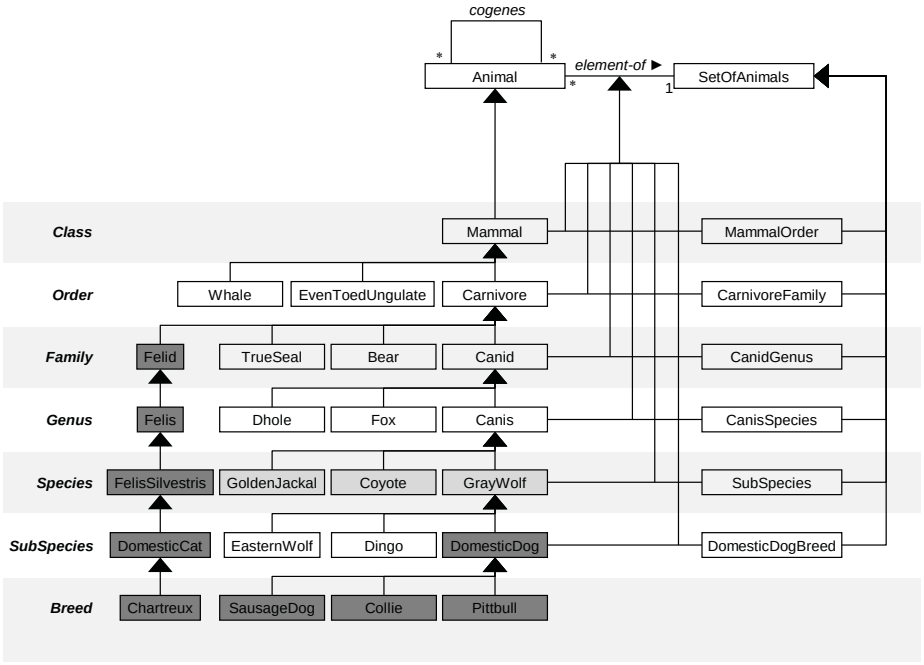

Fig. 6: The mammal hierarchy.

a concept of power set into the tier of class instances. Note, that even in set theory [91] all sets are objects. The class of sets, i.e., the set carrier, consists of opaque objects only. Inner structure of sets is an illusion that emerges in our minds by the application of the set forming operation $\{\_,\_\}$ on the carrier, i.e., the class of sets is an abstract data type. Sets of sets emerge by the axiom of pairing in the finite case and by the axiom of infinity in the infinite case. However, once a set of set is constructed you can also perceive it as a relationship between its members and itself, and equally, it is only a perception or illustration that membership means containment. And this is also true for predicate logic. Signatures in predicate logic can be considered pure, most possible reductionist entity-relationship models.

A credo often somehow stated in object-orientation is: *Everything is treated as an object.* In form-oriented analysis [37, 26] we have expressed doubt in the metaphoric power of such and similar real-world statements. We have said that the value of such a statement is not clear if it is only used as a preamble or eye-catcher and is not exploited anywhere else in the subsequent methodology or its semantic foundation. Now, with the current discussion we have actually found a use case for this real-world statement. If it is a crucial value for object-orientation that everything is treated as an object, then also sets, sets of sets, sets of sets of sets and so forth should be treated as objects.

5.1 Plain Class Modeling for Sets of Sets of Domain Objects

Fig. 7 shows a state of our *dogs and breeds* expert domain and a first simple yet adequate conceptual model for the intended domain. The diagrams (ii) through (iv) in Fig. 8 show further, more elaborate means to model the intended domain. Diagram (i) in Fig. 8 is, basically, the conceptual model copied from Fig. 7. It is included into Fig. 8 for an important presentation issue, i.e., in order to complete the full power type construction diamond.

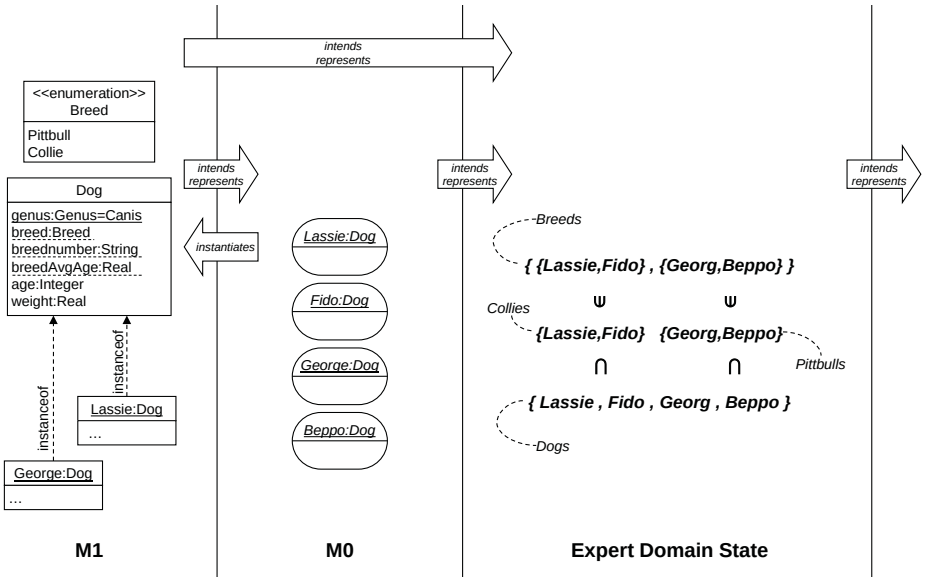

Fig. 7: Adequate constraint-based modeling of sets of sets without power types. The necessary OCL_R constraints are given in the text as (46)–(56)

The M0- and M1-level models in Fig. 7 together with the OCL_R constraints that are given in the sequel adequately represent the domain and the current state of the domain. The class Dog represents the set of dogs in the domain state. The set of dogs is a domain object that owns a genus, in this case Canis, as a property. All the dogs share the same property, therefore, this property is modeled as a class attribute in the M1-model, which is, as usual, indicated by underlining the attribute. Instead of assuming a singleton object as host for the class attribute, we explicitly specify this by the following OCL_R constraint:

$$\begin{aligned} &Dog.allInstances \rightarrow forAll(dog_1, dog_2 \mid \\ &\quad dog_1.genus = dog_2.genus \\ &\quad or\ dog_1.genus \rightarrow asSet \rightarrow size = 0 \\ &\quad or\ dog_2.genus \rightarrow asSet \rightarrow size = 0 \\ &) \end{aligned} \tag{46}$$

We prefer to use the term class attribute over using the UML term static attribute. Because UML static attributes are not really static, but just class-global. A UML static attribute can vary over M0-model editions; however, what is required for a UML static attribute is that it is equal for all objects of its hosting class in a given state. We have chosen the current formalization of the concept of class attribute, because it is amenable in a straightforward manner on basis of OCL.

A reader might say that constraint (46) is superfluous, because the UML specification states that a class attribute belongs to the class rather than to the objects of the class. However, the UML specification is not formal with respect to this, because it neither states the existence of a singleton object hosting the class attributes for each class nor does it mention class attributes in the semantic description of class instantiation. In that sense constraint (46) is one means to make the semantics of class attributes precise. However, the purpose of constraint (46) in this article is different, we want it to be at hand for comparison with the constraints for subset-global attributes like (48) to (51) in the sequel.

The constraint (46) is quite explicit and elaborate, it can be expressed much denser in a different style:

$$Dog.allInstances.genus \rightarrow asSet \rightarrow size \leqslant 1 \tag{47}$$

Second, each dog has an age and a weight. These properties are, without loss of generality, different for each dog. Therefore, they are modeled as ordinary object attributes. Third, each dog has a breed, a breed number and the average age of its breed as a property. Again, these properties are different for each dog, but they are not completely arbitrary. Instead, there is a mutual functional dependency between the breeds and the breed numbers, and a functional dependency between breeds and average ages per breed. We choose the breed itself to identify the respective subset of dogs, i.e., collies or pitbulls. The enumeration type Breed hosts values Pitbull and Collie for this purpose. The properties breed number and breed's average age are not global with respect to the class dog but must be the same for all collies and independently the same for all pitbulls. We can express this by the following constraints:

$$Dog.allInstances \rightarrow select(breed = \#Collie).breednumber \rightarrow asSet \rightarrow size = 1 \tag{48}$$
$$Dog.allInstances \rightarrow select(breed = \#Collie).breedAvgAge \rightarrow asSet \rightarrow size = 1 \tag{49}$$
$$Dog.allInstances \rightarrow select(breed = \#Pitbull).breednumber \rightarrow asSet \rightarrow size = 1 \tag{50}$$
$$Dog.allInstances \rightarrow select(breed = \#Pitbull).breedAvgAge \rightarrow asSet \rightarrow size = 1 \tag{51}$$

Note, that the leading sign # in (48) through (51) is the usual way to denote enumeration literals in OCL. Furthermore, note that the constraints (48) through (51) must not be mixed with constraints of the following form:

$$
\begin{aligned}
&Dog.allInstances \rightarrow select(breed = \#Collie) \rightarrow forAll(breednumber \rightarrow asSet \rightarrow size = 1)\\
&Dog.allInstances \rightarrow select(breed = \#Collie) \rightarrow forAll(breedAvgAge \rightarrow asSet \rightarrow size = 1)\\
&\ldots
\end{aligned}
\tag{52}
$$

This means that the constraints in in (48) through (51) are not merely about multiplicities of properties as one might think at the first sight. Instead, they specify the uniqueness of the properties with respect to each breed. Multiplicities of properties are specified by the constraints of the form (52). They specify a [1..1] cardinality for the properties. In our example, the [0..1] cardinality is implicitly specified for the properties in diagram (i) in Fig. 7, because it can be assumed as the default cardinality of properties.

Later, when we consider subtype externalization in Sect. 5.3 we will discuss that these constraints can be expressed by turning the subset-global attributes into appropriate class attributes. The average age for collies can be the same as the average age of pit bulls. However, the breed number is regarded as the identifier in the domain. It must be different for collies and pitbulls. We can express this by the following constraints:

$$
\begin{aligned}
&Dog.allInstances \rightarrow select(breed = \#Collie) \rightarrow forAll(c \mid\\
&\quad Dog.allInstances \rightarrow select(breed = \#Pitbull) \rightarrow forAll(p \mid\\
&\qquad not(c.breednumber = p.breednumber)\\
&\quad)\\
&)
\end{aligned}
\tag{53}
$$

Now, last but not least, let's have a look at the set of breeds in the expert domain. The set of breeds is represented by the enumeration type Breed at M1-level. The set of collies is represented by the set of M0-level objects that share the value Collie for their breed attribute. It is also represented at M1-level by the value Collie itself.

5.2 Making Constraints Robust against M1-level Model Updates

In Sect. 5.1 we have seen an important aspect of modeling sets of sets, i.e., subset-global attributes. We need to investigate these further and will introduce the notion of subclass attribute. Furthermore we need to discuss auxiliary properties for subsets of objects as well as properties that are global to sets of sets of objects. For this purpose, we investigate several options of modeling in Fig. 8. However, the central theme of this section turns out to be the question of how to make constraints robust against model updates at level M1.

The constraints (48) to (51) work fine to protect the M0-level objects against inadequate updates. However, in general, they are not sufficient for M1-level model updates. For example, if a new value, e.g., Beagle, is introduced by the modeler into the enumeration type, the subset-global attribute for breed numbers is *no longer under the auspices of appropriate constraints*, because the existing

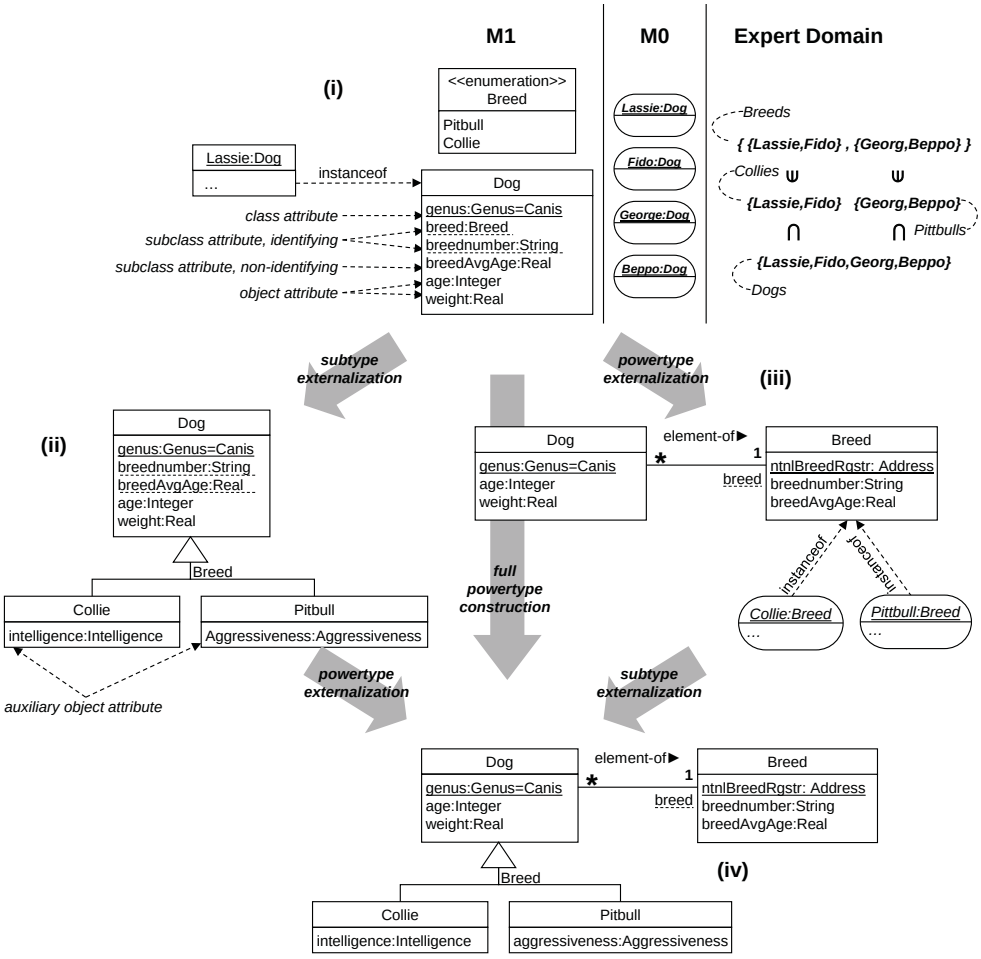

Fig. 8: Constraint-based modeling of sets of sets with UML modeling elements.

constraints work only for the former collection of enumeration type literals Collie and Pitbull, which infringes the intended meaning of the Breed enumeration type as representing the domain set of breeds.

Now, the following OCL_R constraints generalize the constraints (48) to (51) by abstracting from the concrete values in the enumeration type so that the constraints become robust against unwanted M1-level updates:

$$\begin{array}{l}\langle Breed\rangle\downarrow .ownedLiteral \rightarrow forAll(breedId \\ \quad Dog.allInstances \\ \quad \rightarrow select(breed = \langle breedId\rangle\uparrow).breednumber \rightarrow asSet \rightarrow size = 1 \\ )\end{array} \tag{54}$$

$$\begin{array}{l}\langle Breed\rangle\downarrow .ownedLiteral \rightarrow forAll(breedId \\ \quad Dog.allInstances \\ \quad \rightarrow select(breed = \langle breedId\rangle\uparrow).breedAvgAge \rightarrow asSet \rightarrow size = 1 \\ )\end{array} \tag{55}$$

See how, the constraints (54) and (55) make use of reification and reflection. The type *Breed* is a user defined type. It denotes an enumeration. Therefore, after reification of *Breed* we have access to its enumeration literals via introspective access. These can be, after the application of the reflection operator $\langle breedId\rangle\uparrow$, further exploited in M1-level constraint writing.

Let us call a constraint that is made robust against M1-level updates, a *sustainable constraint*. We will further delve into this terminology in Sect. 8. Next, we also want to turn constraint (53) into a sustainable version. This is even possible without OCL_R, i.e., in plain OCL. With constraints (48) and (50) we have specified that breed numbers are unique with respect to each breed. The fact can be exploited to give a less explicit and much more dense version of constraint (53):

$$\begin{array}{l} Dog.allInstances.breed \rightarrow asSet \rightarrow size \\ = Dog.allInstances.breednumber \rightarrow asSet \rightarrow size \end{array} \tag{56}$$

As it turns out, the constraint (56) is independent of concrete breed identifiers and is therefore a robust version of (53).

Now, let us detour a bit and discuss the pragmatics of tool design. A more detailed discussion is provided by the symbolic viewpoints in Sect. 8. Assume that we have a tool that allows for modeling and instantiation of objects in parallel. Imagine that such a tool supports the maintenance of both the model and the data and, in particular, surveils the validity of reflective constraints, e.g., constraints written in OCL_R. Now, given such a tool, how to introduce a new breed into our example model? The answer is: (i) introduce a new value in the Breed enumeration type, (ii) instantiate some dog information objects, (iii) set the attributes of the new objects and care for the equality of the set-global attributes and the uniqueness of the breed number, (iv) submit the model changes as update and (v) expect the modeling tool to do the necessary constraint checking and reject resp. accept the changes based on the result.

5.3 Subtype Externalization

Model (ii) in Fig. 8 shows the result of externalizing the breed attribute into subclasses for each possible value. Having a certain value for an attribute, i.e., having a certain property, characterizes a subset of a set of objects, i.e., the set of objects sharing this property. Therefore, in model (ii) the subclasses Collie and Pitbull represent the domain sets of collies resp. pitbulls. The generalizations of the classes Collie and Pitbull to the class Dog form a UML generalization set which receives the name Breed in model (ii). This generalization set Breed now adequately represents the set of breeds in the expert domain.

Still, we need to enforce the global uniqueness of the breed number and average age with respect to the subsets of collies and pitbulls. We could get this effect by erasing the respective attributes from the class Dog and moving them as class attributes to the subclasses Collie and Pitbull. However, it is better OO-style to keep them in the class Dog so that they are inherited by the subclasses, for example, because we want to introduce further breeds as subclasses in future model editions. A means to override an attribute by a class attribute in a subclass is also no substitute for the given constraints, because without the constraints nothing ensures that the attribute is systematically overridden in all subclasses under consideration. In the current scenario, it is fair to call these attributes *subclass attributes*, because semantically they can be considered class attributes of the subclasses. We have therefore underlined them with a dashed line in the diagrams (i)-(iii) in Fig. 8. We do not want to introduce the concept of subclass attribute with this semantics as a language element here, because although it would work immediately for usual OO programming languages, it is incomplete in UML. In contrast to usual programming languages, generalization in UML can be non-disjoint [68], so in general you would also need to specify the subclasses for which the intended properties are considered as set-global.

The constraints (48) to (51) can now be re-stated for model (ii) as follows – note that the resulting constraints are actually class attribute constraints onto the considered attributes in their role as inherited attributes:

$$Collie.allInstances.breednumber \rightarrow asSet \rightarrow size = 1 \tag{57}$$

$$Collie.allInstances.breedAvgAge \rightarrow asSet \rightarrow size = 1 \tag{58}$$

$$Pitbull.allInstances.breednumber \rightarrow asSet \rightarrow size = 1 \tag{59}$$

$$Pitbull.allInstances.breedAvgAge \rightarrow asSet \rightarrow size = 1 \tag{60}$$

Now, in order to make the constraints (57) to (60) sustainable, i.e., robust against M1-level updates, we can restate the sustainable constraints (54) and (55), again in OCL_R, in terms of the generalization set *Breed*. The involved subclasses are the same with respect to their generalization set as the literals are with respect to their enumeration type:

$$\begin{aligned} &\langle Breed \rangle\downarrow .generalization.specific \rightarrow forAll(\\ &\quad \langle self \rangle\uparrow .allInstances.breednumber \rightarrow asSet \rightarrow size = 1 \\ &) \end{aligned} \tag{61}$$

$$\begin{array}{l}\langle Breed\rangle\downarrow .generalization.specific \rightarrow forAll(\\ \quad \langle self\rangle\uparrow .allInstances.breedAvgAge \rightarrow asSet \rightarrow size = 1\\ )\end{array} \tag{62}$$

Next, we also give an equivalent to constraint (56):

$$\begin{array}{l}\quad \langle Breed\rangle\downarrow .generalization.specific \rightarrow asSet.size\\ = Dog.allInstances.breednumber \rightarrow asSet \rightarrow size\end{array} \tag{63}$$

With model (ii) the introduction of a new breed turns out to correspond to the introduction of a new subtype under the auspices of the necessary constraints. Model (ii) has an important advantage over model (i). The subclasses are the natural host for auxiliary attributes that are specific to a certain subset of domain objects. In the example we have chosen the attribute intelligence for collies and the attribute aggressiveness for pitbulls.

In principle it is possible to turn attributes of a each type into a subtype, i.e., not only attributes of an enumeration type. For enumeration types, which are finite, we simply turn each literal into a type, as we have seen in the current example. For an infinite type we update the model by the introduction of a new subtype representing a value of the type whenever necessary, i.e., whenever an attribute with this value occurs for the first time. This extreme subtype externalization is merely a thought experiment; but it is an instance of the purely symbolic viewpoint of modeling that we will discuss in Sect. 8.3, because it treats the evolving M1/M0-model as a single whole data store.

5.4 Power Type Externalization

Model (iii) shows the result of externalizing all the subset-global attributes in their own class Breed. It is usual to call a class like the class Breed a power type [47, 46, 69]. Now the concept of the set of breeds is made explicit by a class in the model. A concrete breed can now be represented by an M0-level object or an M1-level instance specification. This modeling solution might appear to the reader as particularly natural, because a class can be seen as the natural candidate to represent a set of objects, which are meant to be sets in this case. Actually, because of the 1-multiplicity at the element-of association, the constraints (48) to (51) become obsolete with solution (iii). Now, all we need to do is to generalize this situation to an arbitrary number of breeds is to adopt constraint (56) the following way:

$$\begin{array}{l}\quad Dog.allInstances.breed \rightarrow asSet \rightarrow size\\ = Dog.allInstances.breed.breednumber \rightarrow asSet \rightarrow size\end{array} \tag{64}$$

With the model (iii) there might be empty breeds due to the breed-to-dog association's many-cardinality [$*$]. If we want (63) to effect also empty breeds we need to change it to:

$$\begin{array}{l}\quad Breed.allInstances \rightarrow asSet \rightarrow size\\ = Breed.allInstances.breednumber \rightarrow asSet \rightarrow size\end{array} \tag{65}$$

Note, that both (64) and (65) are no OCL_R constraints, i.e., they are plain OCL constraints. With solution (iii) the membership of a dog in a breed is represented by instances of the element-of association. The model (iv) has an important advantage over the model (i). The class Breed is the natural host for auxiliary properties that are common to all breeds, e.g., the address of the national breed registry. Without the need for subtype-specific attribute extensions solution (iii) actually appears the most natural modeling pattern for the given scenario. This comes at no surprise: solution (iii) is no more, no less than the *type-object pattern* [53] of Johnson and Woolf. Unfortunately, we sometimes might want to model properties that are specific to certain breeds – see also the discussion on the disadvantages of implementation complexity in [53]. This leads us to the next Sect. 5.5.

5.5 Integrated Subtype and Power Type Externalization

Solution (iv) now shows an equivalent to the full UML power type construction [76, 61] for the scenario. It makes explicit *a)* the several breeds as subclasses Collie and Pitbull and *b)* the set of all breeds as a class Breed. These two representations must now be balanced and kept in synch. First, we need an OCL_Rconstraint that expresses that all M0-objects of a given breed subclass are assigned to the same breed M0-object:

$$\begin{array}{l} \langle Breed \rangle\!\downarrow .generalization.specific \rightarrow forAll(\\ \quad \langle self \rangle\!\uparrow .allInstances.breed \rightarrow asSet \rightarrow size = 1 \\ ) \end{array} \tag{66}$$

Second, we also need to express that objects of different subclasses are assigned to different Breed objects:

$$\begin{array}{l} Dog.allInstances.breed \rightarrow asSet.size \\ = \langle Breed \rangle\!\downarrow .generalization.specific \rightarrow asSet \rightarrow size \end{array} \tag{67}$$

The user-defined type name *Breed* is overloaded in diagram (iv). It denotes both the power type call *Breed* as well as the generalization class *Breed*. This does not pose a problem. In the constraints (66) and (67) the type *Breed* is used to denote the generalization set. Together, the constraints (66) and (67) imply that the subclasses of the *Breed* generalization set have no instances in common, i.e., that their sets of instances are disjoint. This is not automatically so. Multiple classification is, as a matter of course, an option with the UML, for example, because of multiple inheritance. UML generalization sets have an attribute *isDisjoint*, that specifies whether the specific classifiers of a generalization set may have instances in common or not – see [76]. We can specify that the subclasses of *Breed* are disjoint as follows:

$$\langle Breed \rangle\!\downarrow .isDisjoint \tag{68}$$

Note, that (66) and (67) together imply (68), but, however, the converse implication does not hold.

The two constraints (66) and (67) capture the essence of the UML power type construct, however, only for a special case. First, there must be exactly one power type association and, moreover, the involved power type association may be a many-to-one association only, see the element-of association in Fig. 8 in our case. In Sect. 7 we will provide general constraints for arbitrary user-defined power type specifications.

6 Z and Sustainable Constraint Writing

In this section we restate the domain model from Fig. 7 in the specification language Z. The aim of this is twofold. First, the specification offers a particularly dense presentation of the crucial domain knowledge discussed throughout the article and is amenable to foster its understanding. In that sense, we will refer to this Z example later in Sect. 7 on the precise semantics of UML power types. Second, and maybe even more important, its discussion can foster the understanding that semantics and pragmatics are concerns in language design that can and should be separated – and this is so also, and in particular, in case of modeling languages.

The specification language Z allows describing system states on the basis of set theory and predicate logic. It offers rich notation for all usual mathematical constructs. It is an advantage to have a standardized means to write mathematical specification. However, Z is more than a neat set notation. It establishes a system model and a system modeling paradigm. A system is modeled as a state evolvement. The approach is to model the state transition as manipulation of declared functions (pre-post-condition specification). It belongs to the large family of Parnas methods [79] with ASMs (abstract state machines) as a most recent member [40]. We use only the data facet of Z in this article. I recommend [88] as a reference, and also [43, 44, 62]. Furthermore, the Z notation is standardized by an ISO standard [50].

Z specifications are not automatically sustainable. However, they can be turned into sustainable specifications. No reflective refactoring of Z is needed for this purpose, because Z allows for quantification over arbitrarily nested sets. It is common, e.g., in text books on Z, to say that the Z notation is a combination of set theory and first-order logic. But take care; this is not a formal statement. Informally, it is a neat explanation. Formally, it can neither be neglected nor approved, because it is not clear what is meant by *combining* set theory and first-order logic. In any case, it is important to understand, that in Z it is possible to quantify over arbitrarily nested sets. So, if Z had a sufficiently formal semantics, it would be in the realm of a typed, higher-order logics, comparable to Isabelle/HOL [65], see also [84, 55] for a discussion. The way we turn a Z specification of our example domain into a sustainable Z specification in Sect. 6.3 is very instructive and gives us yet another viewpoint onto today's object-constraint languages, their expressive power and their pragmatics.

6.1 Types Specification

We introduce the basic sets of dogs, addresses, genera, breed numbers, intelligence degrees and aggressiveness degrees. There are all kept completely opaque in the following:

$$\begin{array}{l}[DOG] \\ [ADDRESS, GENUS] \\ [BREEDNUMBER, INTELLIGENCE, AGRESSIVENESS]\end{array} \tag{69}$$

It would be typical Z style to introduce further, derived types for intended domain concepts. For example it would be typical to introduce the following type for breeds:

$$BREED == \mathbb{P}\, DOG \tag{70}$$

There are two good reasons for auxiliary types as (70). First, they can improve the self-documentation of the specification. Second, they improve re-use. It is typical Z style to make intensive use of such auxiliary types. However, in our case, we stay with the plain types given in (69), because this eases the discussion. Our interest in this section is the discussion of design principles, whereas the artifact quality of the specifications play a minor role.

6.2 Schema Specification

The structure of possible system states manifests in variables and axioms declared in schemas. Mathematical notation is the first class-citizen in Z. For the modeler this means, that he must often specify concepts that would be available as syntactic sugar in other modeling languages. Nevertheless, Z specifications are usually rather dense than bloated. The advantage is that we can hardly deviate from the declarative, mathematical semantics. With the schema *DogDomainData* in (71) we provide a straightforward specification of the system state. With the schema *DogDomainAttributes* in (72) we add the attributes – compare this to the domain model provided in Fig. 7. In our Z specification we model each attribute as a function that yields a value for each given parameter object. This means, we explicitly model an object-mechanism that is implicitly given in each object-oriented modeling language – see also the ephemeral object patterns in [4, 3] for a discussion of this specification style.

Each variable in (71) represents a part of the system state. Therefore each variable holds a subset of its corresponding base type and is typed as power set of this. The set *dogs* stands for the set of dogs at one point in time, whereas the type DOG stands for the set of all possible dogs that may ever exist in any of the system states. Consequently the type of the set *dogs* is modeled as the power set of DOG. Similarly, the type of the set *breeds* is modeled as the power power set of DOG. At each point in time the set *breeds* consists of sets of dogs. Furthermore, we specify that both of the two breeds *collies* and *pitbulls* are subsets of the set of dogs in each system state. Furthermore, we specify that the two breeds *collies* and *pitbulls* are always disjoint sets.

$$
\begin{array}{|l}
\textit{DogDomainData} \\
\hline
dogs : \mathbb{P}\, DOG \\
breeds : \mathbb{P}\mathbb{P}\, DOG \\
collies : \mathbb{P}\, DOG \\
pitbulls : \mathbb{P}\, DOG \\
\hline
collies \subseteq dogs \quad (i) \\
pitbulls \subseteq dogs \quad (ii) \\
collies \in breeds \quad (iii) \\
pitbulls \in breeds \quad (iv) \\
collies \cap pitbulls = \varnothing \quad (v) \\
\hline
\end{array}
\tag{71}
$$

The attributes of the classes of our example are modeled as partial function from the sets of potential objects, i.e., types, into their value ranges in schema (72). A major role of the schema (72) is then, to specify the correct domains of the attribute functions. There is no need to specify the uniqueness for breed numbers and breed average ages for the members of a given breed in the Z solution. This is so, because the corresponding attributes are modeled as functions that have the set of breeds as their domain. The functions assign values to breeds, not to dogs. This solution corresponds to the power type externalization solution in Sect. 5.4, in which these attributes were modeled as properties of power type objects.

$$
\begin{array}{|l}
\textit{DogDomainAttributes} \\
\hline
DogDomainER \\
dogGenus : GENUS \\
dogAge : DOG \pfun \mathbb{N}_0 \\
dogWeight : DOG \pfun \mathbb{R} \\
breedNationalBrgRegistrar : ADDRESS \\
breedNumber : \mathbb{P}\, DOG \pfun BREEDNUMBER \\
breedAvgAge : \mathbb{P}\, DOG \pfun \mathbb{R} \\
collieIntelligence : DOG \pfun INTELLIGENCE \\
pitbullAgressiveness : DOG \pfun AGRESSIVENESS \\
\hline
\mathrm{dom}\, dogAge = dogs \\
\mathrm{dom}\, dogWeight = dogs \\
\mathrm{dom}\, breedNumber = breeds \\
\mathrm{dom}\, breedAvgAge = breeds \\
\mathrm{dom}\, collieIntelligence = collies \\
\mathrm{dom}\, pitbullAgressiveness = pitbulls \\
\hline
\end{array}
\tag{72}
$$

Now, let us compare the Z solution to the power type solution in Sect. 5.5. Constraints (i) and (ii) in the schema *DogDomainData* correspond the introduction of *Collies* and *Pitbulls* as subclasses in the generalization set *Breed* in Sect. 5.5. In the power type solution we had to balance the subclasses of the

full power type construction with the power type objects. It is constraint (66) that enforces a unique power type object for all objects of a given subclass. The constraints (iii) and (iv) for *collies* and *pitbulls* in the schema *DogDomainData* can be considered the counterpart of constraint (66) in Sect. 5.5. The constraints (iii) and (iv) are particularly simple. Let us have a look at the set *collies*. The set *collies* is itself both a subset of the set *dogs* and at the same time an element of the set *breeds*. This is possible, because Z has the expressive power of a typed, higher-order logic. There is no need for an extra object representing the set *collies* as a whole. Attributes that are common to all collies are simply assigned to the whole set *collies*. All this is also true for the set of *pitbulls*. Therefore, there is no need to balance the set of *collies* and *pitbulls* against objects that represent.

Next, the constraint (67) in Sect. 5.5 enforces that the set of instances of *Collies* is disjoint from the set of instances of *Pitbulls*. Therefore, constraint has (67) constraint (v) in the schema *DogDomainData* as its counterpart in the Z solution. The Z constraint (v) is again particularly easy due to the fact that we can exploit mathematical set notation for it.

6.3 Sustainable Schema Specification

The crucial constraints (i), (ii) and (iv) in schema (71) are not sustainable. If we add a new breed, let's say $beagles \in breeds$, it is neither ensured, that the new breed is a sub set of the set of dogs, nor that it is disjoint to the already existing breeds. Let us have a look at schema (73) which is a solution to this problem. Constraint (i) in (73) is an appropriate sustainable generalization of the constraints (i) and (ii) in (71). Constraint (v) in (73) is an appropriate sustainable generalization of its counterpart (v) in in (71).

DogDomainDataSustainable

$$
\begin{array}{l}
dogs : \mathbb{P}\, DOG \\
breeds : \mathbb{P}\mathbb{P}\, DOG \\
collies : \mathbb{P}\, DOG \\
pitbulls : \mathbb{P}\, DOG \\
beagles : \mathbb{P}\, DOG \\
\hline
\forall\, breed : breeds \bullet breed \subseteq dogs \qquad (i) \\
collies \in breeds \qquad (ii) \\
pitbulls \in breeds \qquad (iii) \\
beagles \in breeds \qquad (iv) \\
\forall\, breed_1, breed_2 : breeds \bullet breed_1 \cap breed_2 = \varnothing \qquad (iv)
\end{array}
$$

(73)

It also would have been possible to add explicit constraints for the new beagle breed to the schema *DogDomainData*, i.e.:

$$beagles \subseteq dogs \tag{74}$$

$$beagles \cap collies = \varnothing \tag{75}$$

$$beagles \cap pitbulls = \varnothing \tag{76}$$

But, of course, the solution of schema (73) is better. Explicit constraints are again non-sustainable. They are not generic and therefore explicit constraints do not scale. Already in the current small example, they start to bloat the specification.

7 Precise Semantics of UML Power Types

The current UML superstructure specification contains the following description of the semantics of power types [76]:

> *Formally, a power type is a classifier whose instances are also subclasses of another classifier.* [...] *As established above, the instances of Classifiers can also be Classifiers. This is the stuff that meta models are made of.*

The statement is inconsistent against the background of the rest of the UML specification [76]: an M1-level subclass is an instance of the M2-level class *Class* and cannot be an instance of an M1-level classifier. Instances of an M1-level classifier cannot be classifiers themselves. Instances of an M1-level classifier are M0-level model elements and definitely do not reside at level M1. We must not mix the level-crossing UML instantiation relation with the set membership relation $\in$ in the intended domain. If you model with a power type construct the resulting model is not *per se* a meta model. Furthermore, the above statement is not a formal statement, but this is actually a minor point.

Where does the confusion stem from? One source of misunderstanding of the domain-relation $\in$ as level-crossing instantiation may arise from using the phrase *is instance of* for *is element of* in the domain, which might be natural in many domains. Compare this to the Z specification of the running example in Section 6. In (71) *collies* is an element of *breeds* and a the same time a *subset* of *dogs*. We must not mix modeling with the linguistic modeling framework that we exploit as tool, i.e., we must not mix $\in$ with the instantiation of a sentence of our modeling language which is described by its grammar, i.e., a meta model. We should never forget that meta models are really just kinds of grammars and we should not be confused by the fact that we use a common modeling language as notation and mechanism to write these grammars.

However, it is not appropriate to simply reject the above statement from the UML specification and similar statements in the community as inconsistent. It implicitly contains an important aspect of power types that goes beyond their meaning as constraining states of information objects of the current model. Based on the findings and terminology of this article, we can attempt an informal, yet more precise re-formulation of the above definition. For example, we could state:

Informally, a power type is a class whose instances represent sets of domain objects, where each of these domain objects is represented by an instance of a subclass of another classifier.

Arbitrary subclasses? An arbitrary classifier? No. All the extra information expressed by the constraints (66) and (67) is yet still missing, so that both the UML definition of power type as well as are our re-formulation yield no complete specification. Again compare the above statements with the Z specification in (71).

Now, with OCL_R we can give a general semantics for UML power types. With UML, a concrete user-defined power type consists of a generalization set and a designated power type class for this generalization set. The UML specifies that a generalization set has a property *powertype* of type *Classifier*, which is optional, i.e., has [0..1] cardinality. Obviously, the generalization set specifies a power type construct, whenever this value is present. Concrete power type objects can be assigned to the objects of the generalization set. Our specification needs to decide upon pramatic issues, i.e., we assume that there are associations between the super class of the generalization set and the power type class in order to assign concrete power type objects, which is in accordance to the literature and the examples in the UML specification. So far, in Sects. 5 and 6 we have treated examples of a special case, in which there is exactly one such association, which has to be, furthermore a many-to-one association. This special case is the usual case, e.g., all of the examples in the UML specification follow this pattern – see Figures 7-49, 7-50 and 7-51 in [76]. We treat a most general case here – see Fig. 9. There might be many power type associations of arbitrary cardinalities and, furthermore, the generalization set is not yet required any more to be disjoint.

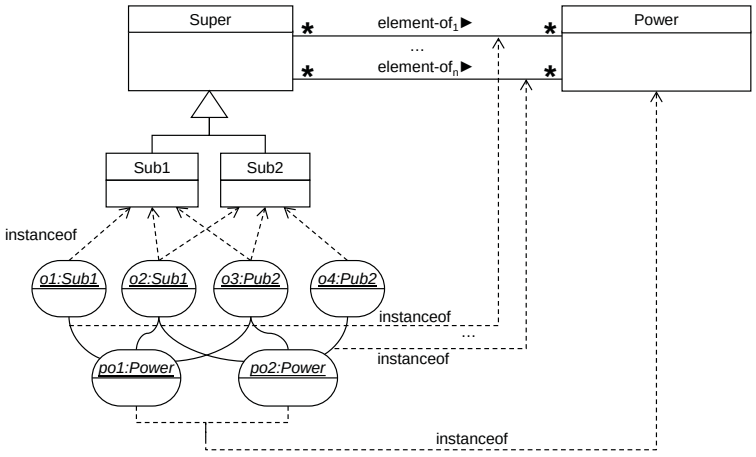

Fig. 9: General UML powertype sepcification pattern.

In the sequel we call an object of one of the subclasses of the generalization set a *generalization set object* for short. Furthermore, we will call a power type object that is assigned to a generalization set object a *representing object*. Now, we need to specify that (i) there is at least one representing object for each generalization set object, (ii) the number of representing objects equals the number of generalization set subclasses, (iii) for each representing object there exists a genereliza tion set subclass, so that the given representing object is assigned to all objects of this suclass. This is achieved by the following OCL_R constraint:

```
01  ⟨Class⟩↓ .allInstances.forAll(Super, Subs, Power |
02    ⟨GeneralizationSet⟩↓ .allInstances→exists(gs |
03      gs.powertpye = Power
04      and   gs.generalization.general→includes(Super)
05      and   gs.generalization.specific = Subs
06    )
07    implies(
08      let ps = Super.ownedAttribute→select(type = Power) in (
09        ⟨Super⟩↑ .allInstances → forAll(o |
10          o.⟨ps⟩↑→ asSet→size ⩾ 1
11        )
12        and
13        ⟨Super⟩↑ .allInstances.⟨ps⟩↑→ asSet→size                    (77)
14        = Subs→asSet→size
15        and
16        ⟨Super⟩↑ .allInstances.⟨ps⟩↑→ forAll(op |
17          Subs → exists(Sub |
18            ⟨Sub⟩↑ .allInstances → forAll(o |
19              o.⟨ps⟩↑→ includes(po)
20            )
21          )
22        )
23      )
24    )
25  )
```

Lines 02 through 06 establish all triples of classes *Super*, *Subs* and *Power* that correspond to a valid and complete user-defined UML power type. Here *Super* stands for *superclass* and means the general class of the generalization set, *Subs* stands for *subclasses* and means the collection of specific classes of the generalization set and *Power* stands for the *powertype* that is assigned to the generalization set – see Fig. 9 once more. Now, the sub constraint in lines 09 through 11 ensures property (i) from above, the sub constraint in lines 13 and 14 ensures property (i) and the sub constraint in lines 16 through 22 ensures property (iii). Altogether, constraint (77) grasps the essential semantics of UML power types. Each sub class is represented by a power type object. A power

type object carries information that is common to all objects of the subclass it represents.

Still, there remain some pragmatic issues that remain open. In case of a complete overlap of the instances of two subclasses of the generalization set, we cannot distinguish between the two involved power type objects representing the subclasses any more. We do not design solutions to issues like that here. Instead, we turn to the special use case of UML power types, in which (i) there exists exactly one power type association, which (ii) then has a many-to-one cardinality. In this case, we need to specify, that (iii) for each generalization set subclass the same representing object is assigned to all objects of the given subclass and (iv) different representing objects are assigned to the objects of different generalization set subclasses. This means that we need to generalize constraints (66) and (67) from Sect. 5.5 to all user-defined power types. This is achieved by the following OCL_R constraint:

```
01  ⟨Class⟩↓ .allInstances.forAll(Super, Subs, Power |
02    ⟨GeneralizationSet⟩↓ .allInstances→exists(gs |
03      gs.powertpye = Power
04      and   gs.generalization.general→includes(Super)
05      and   gs.generalization.specific = Subs
06    )
07    implies(
08      let p = Super.ownedAttribute→select(type = Power) in (
09        p→asSet→size = 1
10        and
11        ⟨Super⟩↑ .allInstances→forAll(o |
12          o.⟨p⟩↑→asSet→size = 1                                    (78)
13        )
14        and
15        Subs→forall(Sub |
16          ⟨Sub⟩↑ .allInstances.⟨p⟩↑→asSet→size=1
17        )
18        and
19        ⟨Super⟩↑ .allInstances.⟨p⟩↑→asSet→size
20        = Subs→asSet→size
21      )
22    )
23  )
```

The sub constraint in line 08 ensures property (i) from above, the sub constraint in lines 11 through 13 ensures property (ii), the sub constraint in lines 15 through 17 ensures property (iii) and the sub constraint in lines 19 and 20 ensures property (iv).

8 A Symbolic Viewpoint of Modeling Languages

Figure 10 shows different viewpoints on model evolution. Note, that they are really only viewpoints on one and the same scenario. Each of the viewpoints grasps important issues in pragmatics of information system design and operations. Furthermore, the distinction between ephemeral versus evolution persistent constraint writing in Fig. 10 is an important concept in its own right.

8.1 The Classic Database Evolution Viewpoint

The first viewpoint (i) in Fig. 10 is the classic database viewpoint, which is also the usual OO programming language viewpoint. The schema is given as an OO class diagram and is cleanly separated from the data. The schema corresponds to the UML M1-level, whereas the data corresponds to the UML M0-level. It is assumed that the schema is fixed, whereas the data is not. The data is continuously manipulated. This viewpoint therefore distinguishes between design time and runtime. The schema shapes the information space. It constraints the structure in which we can capture and maintain data. However, it is also possible to fix more complex domain-related integrity constraints for the data, for example, referential integrity, class-internal functional dependencies, or domain-related integrity constraints, e.g., the rule that a certain integer value must not exceed a maximum value and so forth. A crucial feature of databases is to support the enforcement of these constraints that are considered an integral part of the schema. Whenever you try to update the data in a way that would violate the constraints, the database will reject your update.

We have said that the schema is fixed. But actually it is not. Schema updates can occur. However, it is important to understand that in the viewpoint (i) schema updates are considered to occur seldom and therefore schema updates are considered almost fixed. *Seldom* and *almost* are vague concepts and therefore we will be able to switch to the equal M1/M0 resp. symbolic model evolution viewpoint (ii) later. Furthermore, schema updates are regarded as cost-intensive and are usually controlled by other access rights than those for data updates. Usually, you need to contact your database administrator for this purpose. Whenever a schema update occurs, it triggers a data migration step as indicated by the numbers *1* and *2* in Fig. 10. This data migration step can be very complex, because the existing data must be re-shaped [54, 14, 28, 32, 27]. Similarly, if you change the class structure of your application this at least means that you need to stop, recompile and restart the application program. For an enterprise application this can already be very cost-intensive and risky. Hopefully, the program has been designed for reuse and the change has been foreseen in the applied patterns. If not, and if your changes are really structural, unforeseen changes, this can easily give rise to a cost-intensive code refactoring project.

8.2 The Symbolic Viewpoint

The classic database viewpoint is pervasive. For example, the UML meta level architecture distinguishes between an M1-level and M0-level – note, that the

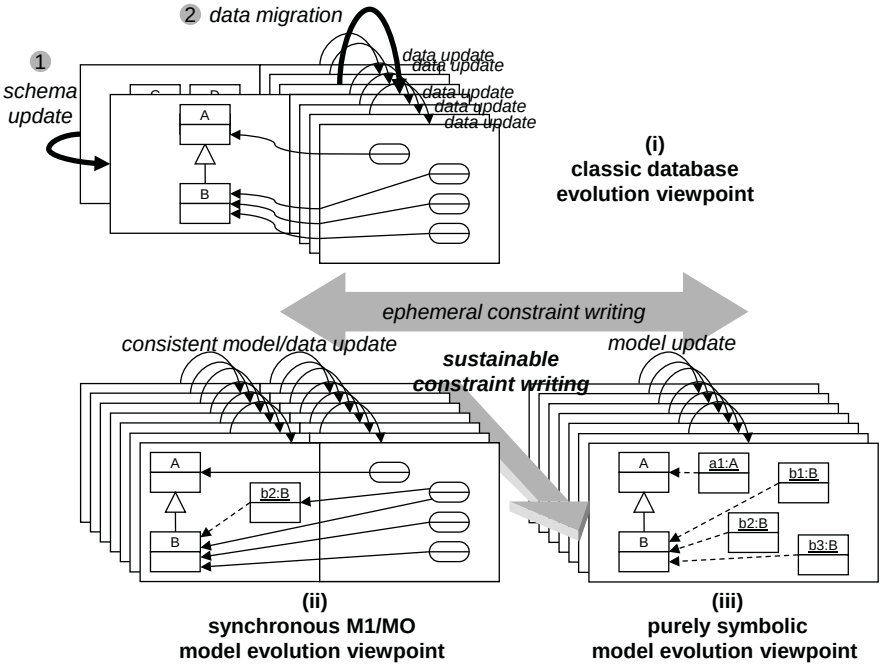

Fig. 10: Viewpoints on model evolution.

M0-level is explicitly called the runtime object level in the UML specification. Nevertheless, the viewpoint is not set in stone. It is simply possible to view schema and data updates as equal. Once we abstract from the differences in frequency, costs and access rights for schema and data updates, the way is free to review the scenario from a different light. First, the M1-level model elements also encapsulate information about the *intended* [15, 16] domain, not only the M0-level objects! In that sense, the M1-level is also a data level. Second, there can be also important constraints on the M1-level model elements with respect to the domain. These can be completely independent from the M0-level. And more importantly, it might be adequate to state them in terms of potential, i.e., not yet instantiated M1-level model elements. For example, you might have a class hierarchy that consists of two trees and might want to ensure that whenever a new subclass is added to one of the trees, a further subclass should also be added at the same position into the other tree. Usual database technology will not support the application of such constraints when updating schemas.

We call a constraint that is written in terms of only a fixed number of concrete M1-level model elements an *ephemeral constraint*, if it may fail to fulfill its intended purpose after an update of some M1-level model elements. Obviously, this description of ephemeral constraints is not a strict definition and the notion of ephemeral constraint is therefore an informal notion. A constraint is ephemeral only with respect to a certain notion of considered model update and a certain

notion of intended purpose. We call a constraint that overcomes the weakness of an ephemeral constraint an M1-level model evolution persistent constraint, evolution persistent constraint, *persistent constraint* or *sustainable constraint* for short. Usually, a sustainable constraint is achieved by generalizing an ephemeral constraint to all possible relevant user-defined types.

Once we have adopted an equal M1/M0 model evolution viewpoint, we are free to think about new tools with innovative modeling features. However, we must not forget about the established database viewpoint, because it incorporates the important aspects of cost-effects and access right management. Furthermore, viewpoint (ii) enables us to rethink the semantics of modeling elements in order to make it more precise.

We call viewpoint (ii) a *symbolic viewpoint*, because it stresses the fact that M1- and M0-level modeling elements can be considered as together intending [15, 16] objects in the expert domain. In terms of symbolic computation the M0-level modeling objects can be regarded as ground terms. For example in the UML, this viewpoint is obfuscated by the existence of instance specifications at level M1, in particular, because instance specifications are optional. Therefore, we introduce the purely symbolic viewpoint (iii) in Fig. 10 as a refinement of viewpoint (ii).

8.3 The Purely Symbolic Viewpoint

In the purely *symbolic viewpoint* (iii) we assume that all M0-level objects are always and only captured and maintained by instance specifications that represent them. For example, as a thought-experiment, we could design a database based on UML class diagrams in which we capture and manipulate database objects always and merely by instance specifications.

The purely symbolic viewpoint can help to avoid certain confusions. In the discussion of OO semantics it can easily happen – and happened in the past – that distinct concepts like the following are thrown together and confused with each other: *a)* instantiations of M1-level elements, *b)* instances of M1-level elements, *b)* set memberships in the expert domain, *c)* representations of instances of M1-level elements at level M1, *d)* instantiations of M2-level elements, *e)* instances of M2-level elements, *f)* representations of set memberships in the expert domain at level M0, *g)* representations of set memberships in the expert domain at level M1, *h)* types at level M1, *i)* classes at level M1, *j)* class constructs of modeling languages, *j)* intensions of sets of domain objects, *k)* domain objects, *l)* the intended meaning of domain objects, and so on and so forth.

You can perceive the achieved M0-level free modeling in two ways. Either practically, as a concrete tool in which the visual modeling canvas is also the data store, or simply as an appropriate formal viewpoint. Because, even if we discuss without M0-level, tools and languages can provide different interfaces or look&feels for the manipulation of the ground terms and the type terms. If we assume that all data is kept and maintained at M1-level this greatly eases and unifies the discussion. Note that in symbolic computation there is also no dedicated grammatical tier for the ground terms. In the reductionist calculi of symbolic computation like the lambda calculus [11] or PCF (Programming With

Computable Functions) [81] the objects resulting from computations are terms of ground type, but still, they are just terms of the language and so it is the same with full-fledged functional programming languages or term rewriting systems. The symbolic viewpoint is a grammatical viewpoint. In a symbolic viewpoint, every object of interest is symbolized as term of the same language. This is the reason, why have chosen to call the viewpoint discussed here a symbolic viewpoint.

8.4 UML Instance Specification

Let us analyze UML instance specifications from the purely symbolic perspective of Sect. 8.3. Instance specifications represent M0-level objects. Let us have a look at our tiny example model in Fig. 7. Here we have M0-level objects *Lassie* : *Dog*, *Fido* : *Dog*, *George* : *Dog* and *Beppo* : *Dog*. Two of them, i.e., *Lassie* : *Dog* and *George* : *Dog* have also a UML-instance specification at M1-level. The UML considers instance specifications as examples only. There is explicitly no need to give an instance specification for each M0-level instance. Furthermore, an instance specification needs not to provide a slot and value specification for each attribute of the corresponding object. However, if we visualize an MO-object by an instance specification at level M1 it would make sense to require that an instance specification should obey to the same rules that we impose as constraints for the M0-level objects. We can do this with appropriate reflective constraints. Let us have a look at a first example, i.e., at the very basic constraint (1):

$$\textbf{context}\ \mathit{Person}\ \textbf{inv}\colon \mathit{age} \geqslant 40 \tag{79}$$

We can turn (79) into an OCL_R constraint that appropriately effects instance specifications the following way:

$$\begin{array}{l}
\langle \mathit{InstanceSpecification}\rangle\!\downarrow .\mathit{allInstances} \\
\rightarrow \mathit{select}(\mathit{classifier} \rightarrow \mathit{includes}(\langle \mathit{Person}\rangle\!\downarrow)) \\
\rightarrow \mathit{select}(\mathit{slot} \rightarrow \mathit{forAll}(\\
\quad \mathit{definingFeature.name} = \texttt{"}\mathit{age}\texttt{"} \\
\quad \mathit{implies} \\
\quad \mathit{value.IntegerValue}() \geqslant 40 \\
\quad) \\
)
\end{array} \tag{80}$$

Now, let us consider a constraint that also contains a navigation expression. Assume that we also have a class *Dog* with property *owner* : *Person*[0..*]. Now, consider the following constraint:

$$\textbf{context}\ \mathit{Dog}\ \textbf{inv}\colon \mathit{owner.age} \geqslant 40 \tag{81}$$

Again, we can turn constraint (81) into an appropriate OCL_R constraint for M1-level instance specifications:

$$
\begin{array}{l}
\langle \mathit{InstanceSpecification}\rangle\!\downarrow .\mathit{allInstances} \\
\rightarrow \mathit{select}(\mathit{classifier} \rightarrow \mathit{includes}(\langle \mathit{Person}\rangle\!\downarrow)) \\
\rightarrow \mathit{select}(\mathit{person} \mid \\
\quad \langle \mathit{InstanceSpecification}\rangle\!\downarrow .\mathit{allInstances} \\
\quad \rightarrow \mathit{select}(\mathit{classifier} \rightarrow \mathit{includes}(\langle \mathit{Dog}\rangle\!\downarrow)) \\
\quad \rightarrow \mathit{exists}(\mathit{slot} \rightarrow \mathit{includes}(\\
\qquad \mathit{definingFeature.name} = \text{"}\mathit{owner}\text{"} \\
\qquad \mathit{and} \\
\qquad \mathit{value} \rightarrow \mathit{includes}(\mathit{person}) \\
) \quad) \quad) \\
\rightarrow \mathit{select}(\mathit{slot} \rightarrow \mathit{forAll}(\\
\quad \mathit{definingFeature.name} = \text{"}\mathit{age}\text{"} \\
\quad \mathit{implies} \\
\quad \mathit{value.IntegerValue}() \geqslant 40 \\
) \quad)
\end{array}
\tag{82}
$$

Note, once more, that the UML allows instance specifications to be partial specifications, i.e., an instance specification does not have to specify a value for each property of the object that is represents. This explains the usage of *implies* in constraint (80). Constraint (80) allows for instances specifications that do not have a slot for the property *age*, however, if such a slot exists, it has to adhere to the given constraints. It is possible to change exactly this partial specification approach. It is possible to give OCL_R constraints that enforce that each instance specification is a full-fledged, consistent object description. Furthermore, the purpose of constraints (80) and (82) has been to demonstrate, that it is, in principle, possible to turn each OCL constraint into an appropriate OCL_R constraint on instance specifications. We do not give the detailed specifications of all this here.

The OCL_R constraints resulting from the described transformation are substantially more complex than the original constraints. Therefore, you might want to think of all the discussion here as a mere thought-experiment. However, it shows that we could get rid of the M0-level to achieve a purely symbolic viewpoint. It is important to understand that the M1- and M0-level together form a language to describe states in the expert domain. The existence of instance specifications merely introduces redundancy. Currently the semantics of UML relies on the notion of M0-objects, and, even more important, its constraint language OCL is designed in terms of M0-objects. We guess that the intention of M0-objects in the UML was to deliberately introduce a degree of freedom in the interpretation of models, i.e., in the sense that M0-objects could be, e.g., data objects in a database, or, run-time objects of an object-oriented programming language and so forth. Merely throwing away the M0-level, without appropriate tool support, is not really an option. However, the discussion also shows that it is actually possible to design appropriate tools for supporting a symbolic modeling viewpoint.

9 Related Work

Programming languages and their type systems, in particular, generative programming languages [20], form a mature field of study that is important for the current discussion. In the programming languages GENOUPE [33, 34, 58] and FACTORY [35, 25] it is possible to implement OCL_R constraints. GENOUPE is a C#-extension, whereas FACTORY is a Java-extension. With GENOUPE and FACTORY generators it is possible to analyze a given class and weave its attributes as class attributes into another class. With these generators the languages are expressive as *DeepJava* [56]. *DeepJava* offers neat clabject-style syntax. The natural candidate for representing sets of sets in C# and Java is, the nested resp. inner classes construct [39]. The problems with nested classes is that subclassing cannot crosscut the nesting structure, which makes impossible a direct, natural transformation of, e.g., model (v) in Fig. 8 into code. This problem is even not overcome by nested inheritance as provide by *Jx* [66] and *J&* [67] or advanced nested composition constructs as provided by DEEPFJIG [19].

Generative programming can be understood in a very concrete, narrow sense. Then, it is about programming languages that offer generative programming language features and establish appropriate type systems for generative programming. Actually, the systematic generation of parts of software systems, in first place code, but also all other kinds of software artifacts, is a practically highly relevant topic and actually a widespread issue in professional projects. It comes along in many faces and flavors: domain-specific languages [22], compiler-compilers [78], rapid-development tools, object-relational mapping tools [14, 28, 32], object-oriented component technologies, enterprise computing frameworks and so on and so forth. The Adaptive Object Model Architecture of Yoder and Johnson [90, 82] is a mature approach to describe self-referential, systematically adoptable software systems, as well as their design patterns and architectural patterns.

In [13] the authors define the conceptual programming language PCF_{DP} as an extension of PCF (Programming with Computable Function) by a quotation mechanism known from LISP that allows for reification and reflection. PCF [81] is the typed lambda-calculus with recursion and can be considered a reductionist functional programming language. Then, the authors give an axiomatic semantics [42] for PCF_{DP} and this way achieve a program logics for generative run-time meta-programming.

Clabject modeling [8, 6] is the established and major multilevel modeling approach [7]. With clabject modeling useful terminology has been created for the distinction between the different kinds of instantiations. In [9] the authors distinguish between so-called linguistic and ontological instantiation. Linguistic instantiation stands for the model-level-crossing instantiation relation. Ontological instantiation stands for the domain-level-crossing relation $\in$ in the intended domain. Clabjects are classes that allow for deep instantiation. The ontological instance of a clabject is itself a clabject and can therefore stand for a set of objects. This way it is possible to adequately represent arbitrarily nested sets,

i.e., the syntactical rules of the clabject frameworks are suitable to guarantee the intended meaning of the model.

In [57] it has been clarified that meta levels must not be confused with the levels of a modeling hierarchy and also, that linguistic instantiation must not be confused with ontological instantiation. In [83] the authors elaborate a formal semantics, based on category theory [12], for terminology that has been created in the multilevel modeling community. *Nivel* [2] is a reductionist multilevel modeling language that supports clabjects, associations, generalization sets, but no power types. A formal description of *Nivel* is provided by translation to the Weight Constraint Language [87], i.e., stable model semantics. This formal description achieves a reformulation of the clabjects rules [10] in type systems notation [17].

Meta modeling tools are the natural candidates for supporting multi-level modeling and clabject modeling. They are also the natural, potential host for pervasive M2/M1/M0-level crossing constraint checking features. The tool MelaniE [6, 8] already offers a clabject-oriented constraint language for this purpose. With an appropriate clabject modeling tool like MelaniE [6, 8] we can assume that all information is represented at M1-level without any M0-level objects, i.e., without linguistic instances of classes. Therefore, clabject modeling tools also establish a modeling viewpoint that is similar to the purely symbolic viewpoint developed in Sect. 8.3. MetaDepth [23, 24] is an implementation of a multilevel modeling language on the basis of the AToM^3 [89] meta modeling tool [89]. It supports clabjects as crucial concept and also checks for adherence to the clabject rules.

The meta modeling tool AMMI [48, 30, 31] defines and realizes the so called *visual reification principle*. Visual reification must not be mixed with the reification operators discussed for the OCL$_R$ semantics here. Visual reification is a kind of bipartite instantiation principle in meta modeling tools that allows for making meta models visually reminiscent of their own instances, which eases meta modeling for domain experts.

The symbolic viewpoints from Sect. 8.3 superficially resemble but must not be confused with the viewpoint of the important strand of research on *models and evolution* [85, 21]. The *models and evolution* viewpoint incorporates potentially many kinds of artifacts with models as centrally important artifacts. It deals with the gaps between these artifact groups. It is a particularly mature but still classical viewpoint. Our symbolic viewpoints deal with models only and deliberately abstract from differences between different kinds of models. Our symbolic viewpoints are merely instructive devices that gain their value only from their tension with classical viewpoints on modeling.

Investigations on the relationship of OO conceptual modeling and ontological modeling are very promising [45, 63] and have impact [51] – see [64] for a sound overview. For the understanding of the arguments in this article, the established mainstream interpretation of OO conceptual modeling as an extended, mature semantic modeling approach [18] is sufficient. In form-oriented analysis [37, 36, 26] we have characterized conceptual modeling as the school of shaping and maintaining information. We have identified real-world metaphors as being merely

guidelines for requirement elicitation. This means, that for the argumentation in this article it is not necessary to understand conceptual modeling in terms of ontological modeling [38], i.e., as construction of an ontological commitment as characterized in [41].

We believe that the viewpoint of *considering models as evolving data storing systems* is also particularly appropriate for the emerging paradigm of cloud-based software engineering [60], which eventually demands, in our opinion, for a more holistic approach to the design of data services and their utilization [4, 3]. For example, in [5] we have coined the concept of *viable software system* which is about systems that are pro-actively designed, implemented and supported in terms of their future versions and releases, and we expressed our opinion that such a concept will be a critical success factor for cloud-based software engineering to take off.

Acknowledgements

I am grateful to Roland Wagner and Josef Küng for the many inspiring discussions on the foundations and, in particular, on the realization of database information systems, e.g., in the context of the DEXA series of conferences and related events. In particular, I am also grateful for the joint endeavors in distributed workflow automation projects.

10 Conclusion

We have shown how to extend an object constraint language with reflection. Reflective constraint writing is to constraint writing what generative programming is to programming. We have extended the concrete object constraint language OCL of the UML modeling language stack for this purpose, resulting in so-called OCL_R. We have shown how to give precise, declarative semantics for OCL_R on the basis of semantical reification operators Φ and Ψ that mitigate between the M2-, M1- and M0-levels of the meta level architecture.

As a by-product, we have shown how to generalize OCL property call expressions by a truly generative version. This means, we have shown how to generalize OCL property call expressions of the form $o.p$ to multi-class, multi-property call expressions of the most general kind $\{o_1 : C_1, .., o_n : C_n\}.\{p_1, .., p_m\}$, i.e., so that the classes C_i can be dynamically generated and properties p_i may be identified merely by name, i.e., may not be inherited from a common supertype.

First, reflective constraint writing can be exploited in quality assurance for system design. Then, a major goal of introducing OCL_R was to support the analysis of semantics and pragmatics of modeling constructs. Another goal of reflective constraint writing is to enable sustainable constraints, which are, typically, constraints involving meta-level access. We have clarified why sustainable constraint writing is important for a robust modeling process. As an example, we have elaborated sustainable constraints, i.e., constraints that persist model evolution, for the modeling of sets of sets.

We have shown how usual class diagrams are sufficient to adequately model sets of sets of domain objects – given that constraints are provided that are appropriately made robust against M1-level updates. We have introduced the concepts of subset-global attribute and subclass attribute. We have introduced and analyzed the subtype externalization pattern. We have introduced and analyzed the power type externalization pattern. The two patterns of subtype externalization and power type externalization open a design space. We have discussed advantages and disadvantages of each of the modeling alternatives. The fact that even basic OO modeling languages are not reductionist as compared to, e.g., the PD (Parsimonious Data) modeling language in form-oriented analysis [36, 26], once more shows the conceptual redundancy of subtype externalization and power type externalization. We have defined and analyzed power type construction as a diamond consisting out of subtype externalization and power type externalization.

We have achieved precise semantics for conceptual models of arbitrarily nested sets. We have argued that the definition of power type in the UML specification is inconsistent. Based on the findings and terminology of this article, a precise re-formulation of the above definition has been possible. We have given a precise specification of the UML power types semantics with OCL_R.

We distinguished three viewpoints onto today's information systems, i.e., the classical viewpoint, the symbolic viewpoint and the purely symbolic viewpoint. It is the purely symbolic viewpoint that has served best to explain the potential of emerging multilevel modeling tools as evolving data storing systems.

Reflective constraint writing adds value. Reflective constraint writing can make constraints robust against model updates. There are many use cases for reflective constraints in different software engineering domains, i.e., both in system design and conceptual modeling. With respect to system design, reflective constraints can be exploited to ensure better artifact quality. They can be used, e.g., to enforce style guides or the correct application of design patterns. Conceptually, reflective constraint writing is about the externalization of important domain knowledge that is otherwise captured in the ephemeral counterparts of sustainable constraints.

A UML Meta Model

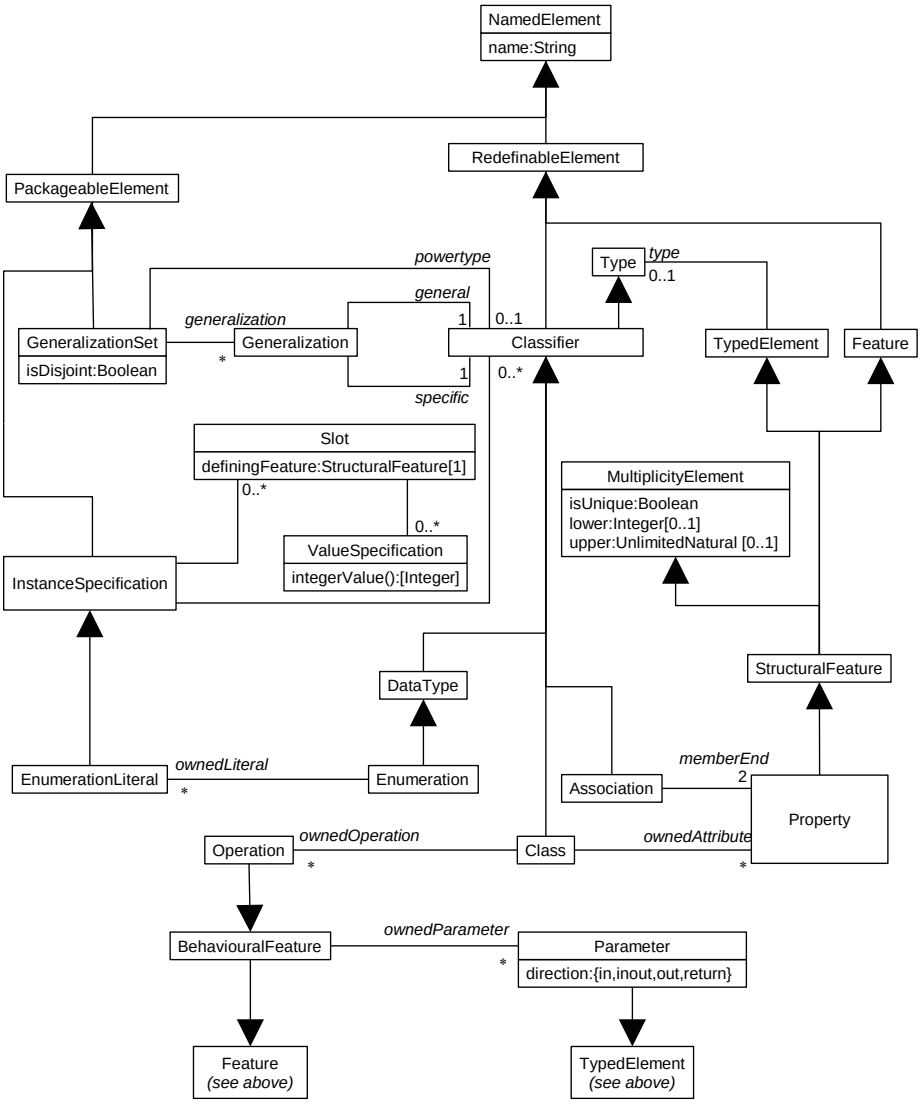

Fig. 11: Cutout of the UML meta model (superstructure) as needed in this article.

The class diagram in Fig. 11 shows a cutout of the UML superstructure specification [76] consisting of all UML meta model elements used in the OCL constraints in this article. We have repeated some of classes, i.e., TypedElement and Feature, for the sake of improving overall readability.

B OCL Types Abstract Syntax

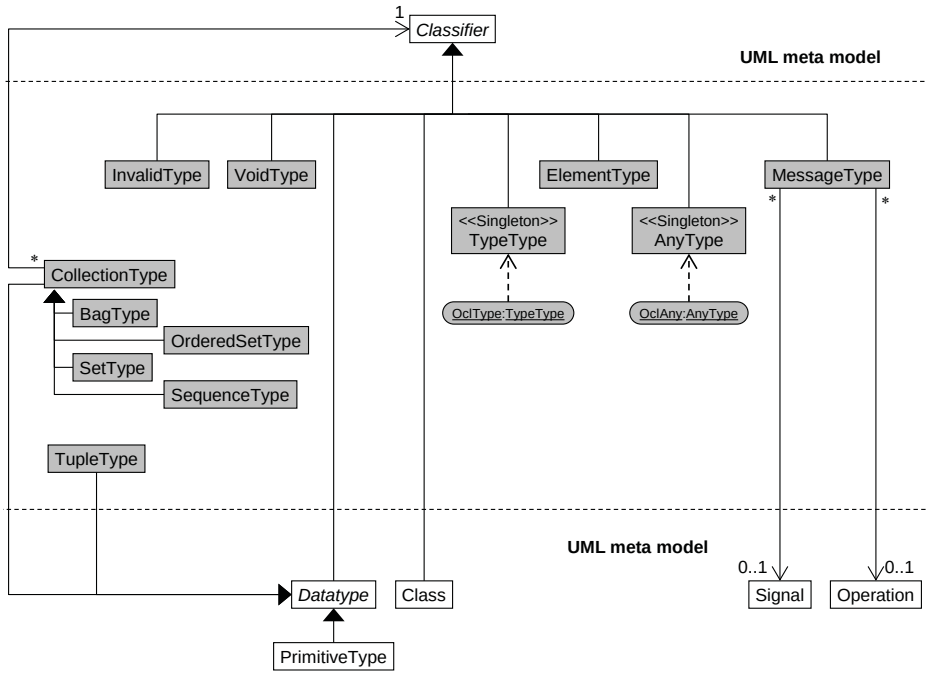

Fig. 12: Meta model of OCL v2.0

Fig. 12 shows the abstract syntax of the OCL v2.0 types. Basically, it shows the types from Figure 8.1. from the OCL v2.0 specification [71]. The singleton *AnyType* meta object *OclAny* serves as most general type for all OCL expressions, i.e., $e :: OCLExpression$ implies $e : AnyType$. The singleton *TypeType* meta object *OclType* serves as type for all OCL type expressions, i.e., $e :: TypeExpr$ implies $e : OclType$. Model elements that are genuine to the OCL type specification are given in gray color, whereas, meta model elements that are reused from the UML superstructure specification [76] have white color.

B.1 On the Choice of OCL version v2.0

We have chosen to take the 2006 version OCL v2.0 [71] instead of the current version v2.4 [74] and the ISO standard version v2.3.1 [73, 52] as the basis for OCL_R. The reason is the type system. The crucial difference is in the existence of the type *OclType* and its corresponding abstract syntax element *TypeType* which are present in the former version v2.0 but absent from the newer versions. The type *OclType* is needed for a complete definition of well-typing. It serves as

type for type expressions, i.e., for expressions $e :: \mathit{TypeExp}$ – see Appendix 13. For example, the problem shows in the definition of the property *oclIsTypeOf*. The property is defined in and OCL v2.0 and the newer OCL version in different ways:

$$\mathit{oclIsTypeOf}(\mathit{type} : \mathit{OclType}) : \mathit{Boolean} \tag{83}$$
$$\mathit{oclIsTypeOf}(\mathit{type} : \mathit{Classifier}) : \mathit{Boolean} \tag{84}$$

The operation applys to all objects, i.e., objects $o : \mathit{OclAny}$. It test whether the object's type equals the type given as parameter. For example, the following constraint evaluates to true:

$$\textbf{context } \mathit{Person} \textbf{ inv}: \mathit{self}.\mathit{oclIsTypeOf}(\mathit{Person}) \tag{85}$$

The OCL v2.0 definition (83) of *oclIsTypeOf* is correct, whereas the v2.0 definition (84) cannot is ill-typed with respect to its described semantics. Even worse, in accordance with its described semantics, the operation $\mathit{oclIsTypeOf}(\mathit{type} : \mathit{Classifier})$ cannot be typed at all with the types avalailble in the newer OCL versions. The expression *Person* is a type expression that denotes a user-defined type. In v2.0 this expression has type *OclType*, so that the definition of *oclIsTypeOf* is correct. Let's turn to the definition of the newer OCL versions. It states that $\mathit{type} : \mathit{Classifier}$. The type *Classifier* can only be a user-defined type, among the pre-defined types there is no type *Classifier*. Here is where the misunderstanding might stem from. The types in the meta model in Fig. 12 are no OCL types themselves. They yield the abstract syntax that describes the OCL types. The existence of the class *Class* in the meta model means that each user-defined type serves as an OCL type, i.e., as a type for OCL expressions. The class *Class* itself is not an OCL type. And so is not the abstract class *Classifier*. Now, the semantic description requires the parameter of *oclIsTypeOf* is a type expression and not an expression of user-defined type. This means that *oclIsTypeOf* is ill-typed in the newer versions of OCL. Furthermore there is no appropriate type available in the newer version of OCL that could be given to the parameter *type*. In v2.0 the type *OclType* serves this purpose. The type *OclType* – yet without a defining abstract syntax and a corresponding meta model element *TypeType* – has been available in OCL since its first 1997 version OCL v1.1 and disappeared from the OCL specification in 2010 with version OCL v2.2 [72].

B.2 Flattening OCL Collections

In the OCL, nested collections are automatically flattened. Each combination of nested collections yields a concrete flattened collection which is defined in [71, 71]. We have turned the definition of this flattening into a combinator $\oplus$ for collection constructors – see the definition in Table 2.

The standard fixes concrete results for the combination of collection into nested structures. Actually, there is a design space. Of course, it is natural to

turn a set of sets into a set and, similarly, to turn a bag of bags into a bag. However, with respect to the combination of bags and sets the OCL has taken a deliberate decision for a symmetric solution, i.e., a bag of sets is turned into a bag, whereas a set of bags is turned into a set. This means, that in the latter case, some information is lost, that is inherent in the encompassed bags. Furthermore, the construction of sequences out of sets and bags is not straightforward, in the OCL it is solved non-deterministically.

_⊕_ (T)	**Set**	**Bag**	**Sequence**
Set	Set(T)	Set(T)	Set(T)
Bag	Bag(T)	Bag(T)	Bag(T)
Sequence	Sequence(T)	Sequence(T)	Sequence(T)

Table 2: The collection type combinator $\oplus$.

C OCL Expressions Abstract Syntax

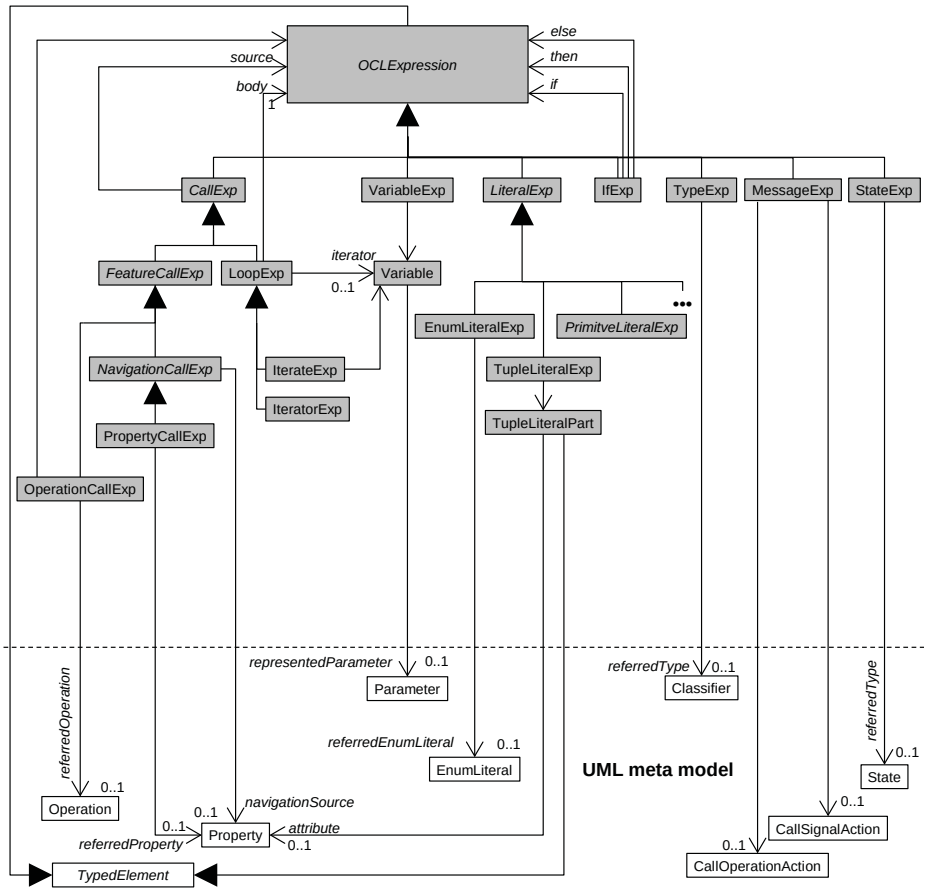

Fig. 13: Meta model of OCL expressions.

Fig. 13 gives a substantial cutout of the OCL abstract syntax as specified in [73]. Basically, it shows, as a single overview, the structure of the OCL syntax kernel as given in Figure 8.2. in [73] plus more elements that are crucial for understanding the syntax and semantics of OCL_R, in particular, Fig. 13 details out the abstract syntax of feature call expressions. Model elements that are genuine to the OCL meta model are given in gray color, whereas, meta model elements that are reused from the UML superstructure specification [76] have white color.

References

1. M. Abadi, L. Cardelli. A Theory Of Objects. Springer, 1996.
2. T. Asikainen, T. Männistö. Nivel – a Metamodeling Language with a Formal Semantics. In: Software and System Modeling, vol. 8, no. 4, 2009.
3. C. Atkinson, P. Bostan, D. Draheim. Foundational MDA Patterns for Service-Oriented Computing. In: The Journal of Object Technology, vol. 13, no. 5, 2015.
4. C. Atkinson, P. Bostan, D. Draheim. A Unified Conceptual Framework for Service-Oriented Computing - Aligning Models of Architecture and Utilization. In (A. Hameurlain, J. Kng, R. Wagner): Transactions on Large-Scale Data- and Knowledge-Centered Systems, vol. 6, Springer, 2012.
5. C. Atkinson, D. Draheim. Cloud Aided-Software Engineering - Evolving Viable Software Systems through a Web of Views. In: Software Engineering Frameworks for the Cloud Computing Paradigm. Springer, 2013.
6. C. Atkinson, R. Gerbig, B. Kennel. Symbiotic General-purpose and Domain-specific Languages. In: Proc. of ICSE 2012 – the 34^{th} Intl. Conf. on Software Engineering, IEEE Press, 2012.
7. C. Atkinson, G. Grossman, T. Kühne, J. de Lara (Eds.). Proc. of MULTI 2014 – the 1^{st} Workshop on Multi-Level Modelling, 2014.
8. C. Atkinson, M. Gutheil, B. Kennel. A Flexible Infrastructure for Multi-level Language Engineering. In: IEEE Transactions on Software Engineering, vol. 35, no. 6, 2009.
9. C. Atkinson, B. Kennel, B. Goß. Supporting Constructive and Exploratory Modes of Modeling in Multi-Level Ontologies. In: Proc. of SWESE'2011 – the 7^{th} Intl. Conf. on Semantic Web-Enabled Software Engineering, 2011.
10. C. Atkinson, T. Kühne. Rearchitecting the UML Infrastructure. In: ACM Transactions on Modeling and Computer Simulation, vol. 12, no. 4, 2002.
11. H.P. Barendregt. The Lambda Calculus – Its Syntax and Semantics. North Holland, Amsterdam, 1984.
12. M. Barr, C. Wells. Category Theory for Computing Science, 2^{nd} edition. Prentice Hall, 1995.
13. M. Berger, L. Tratt. Program Logics for Homogeneous Generative Run-Time Meta-Programming. In: Logic in Computer Science, vol. 11, no. 5, 2015.
14. B. Bordbar, D. Draheim, M. Horn, I. Schulz, G. Weber. Integrated Model-Based Software Development, Data Access and Data Migration. In: Proc. of MODELS 2005 – the 8^{th} ACM/IEEE Conf. on Model Driven Engineering, Languages and Systems, 2005
15. F. Brentano. Psychologie vom empirischen Standpunkt. Duncker & Humblot, 1874.
16. F. Brentano. Psychology from an Empirical Standpoint. Routledge, 1995.
17. L. Cardelli. Type systems. In: Handbook of Computer Science and Engineering. CRC Press, 1997.
18. P.P.-S. Chen. The Entity-Relationship Model – Toward a Unified View of Data. ACM Transactions on Database Systems, vol.1, no.1, 1976.
19. A. Corradi, M. Servetto, E. Zucca. DeepFJig: Modular Composition of Nested Classes. In: Proc. of PPPJ 2011 – the 9^{th} Intl. Conf. on Principles and Practice of Programming in Java. ACM Press, 2011.
20. K. Czarnecki, U. Eisenecker. Generative Programming – Methods, Tools, and Applications. Addison-Wesley, 2000.

21. D. Deridder et.al. (Eds.). Pre.-Proc. of the Intl. Workshop on Models and Evolution at MODELS'2011, 2010.
22. A. van Deursen, P. Klint, J. Visser. Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices, vol. 35, no. 6, 2000.
23. J. de Lara, E. Guerra. Deep Meta-Modelling with MetaDepth. In: Proc. of TOOLS Europe 2010 – 48^{th} Intl. Conf. on Objects, Models, Components, Patterns. Springer, 2010.
24. J. de Lara, E. Guerra. Generic Meta-Modelling with Concepts, Templates and Mixin Layers. In: Proc. of MODELS 2010 – the 13^{th} ACM/IEEE Conf. on Model Driven Engineering, Languages and Systems, Springer, 2010.
25. D. Draheim, G. Weber. Strongly Typed Server Pages. In: Proc. of NGITS'02 – the 5^{th} Workshop on Next Generation Information Technologies and Systems, Springer, 2002.
26. D. Draheim, G. Weber. Modeling Submit/Response Style Systems with Form Charts and Dialogue Constraints. In: Proc. of the Workshop on Human Computer Interface for Semantic Web and Web Applications, Springer, 2003.
27. D. Draheim. Business Process Technology - A Unified View on Business Processes, Workflows and Enterprise Applications. Springer, 2010.
28. D. Draheim, M. Horn, I. Schulz. The Schema Evolution and Data Migration Framework of the Environmental Mass Database IMIS. In: Proc. of the 16^{th} Intl. Conf. on Scientific and Statistical Database Management. IEEE, 2004.
29. D. Draheim. Sustainable Constraint Writing and Symbolic Viewpoints of Modeling Languages. Invited Talk. In: Proc. of DEXA'14 – the 25^{th} Intl. Conf. on Database and Expert Systems Applications, Springer, 2014.
30. D. Draheim, M. Himsl, D. Jabornig, W. Leithner, P. Regner, T. Wiesinger. Intuitive Visualization-Oriented Metamodeling. In: Proc. of DEXA'2009 - the 20^{th} Intl. Conf. on Database and Expert Systems Applications, Springer, 2009.
31. D. Draheim, M. Himsl, D. Jabornig, J. Küng, W. Leithner, P. Regner, T. Wiesinger. Concept and Pragmatics of an Intuitive Visualization-Oriented Metamodeling Tool. In: Journal of Visual Languages and Computing, vol. 21, no. 4, Elsevier, , 2010.
32. D. Draheim, C. Natschlger. A Context-Oriented Synchronization Approach. In: Proc. of the PersDB 2008 – the 2^{nd} Intl. VLDB Workshop in Personalized Access, Profile Management, and Context Awarness, 2008.
33. D. Draheim, C. Lutteroth, G. Weber. Generative Programming for C#. ACM SIGPLAN Notices, vol. 40, no. 8., ACM Press, 2005.
34. D. Draheim, C. Lutteroth, G. Weber. A Type System for Reflective Program Generators. In: Proc. of GPCE 2005 - Generative Programming and Component Engineering, Springer, 2005.
35. D. Draheim, C. Lutteroth, G. Weber. Factory: Statically Type-Safe Integration of Genericity and Reflection. In: Proc. of the ACIS'2003 – the 4^{th} Intl. Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2003.
36. D. Draheim, G. Weber. Modelling Form-Based Interfaces with Bipartite State Machines. Journal Interacting with Computers, vol. 17, no. 2. Elsevier, 2005.
37. D. Draheim, G. Weber. Form-Oriented Analysis – A New Methodology to Model Form-Based Applications. Springer, 2005.
38. T. Froehner, M. Nickles, G. Weiss. Open Ontologies – The need for Modeling Heterogeneous Knowledge. In: Proc. IKE'2004 – the Intl. Conf. on Information and Knowledge Engineering, 2004.

39. J. Gosling, B. Joy, G. Steele, G. Bracha. The Java Language Specification – 3^{rd} editition. Addison Wesley, 2005.
40. Y. Gurevich. Evolving Algebras 1993 – Lipari Guide. Oxford University Press, 1995.
41. N. Guarino. Formal Ontology and Information Systems. In (N. Guarino, Editor): Proc. of FOIS'98 – the 1^{st} Intl. Conf. on Formal Ontology and Information Systems, IOS Press, 1998.
42. C.A.R. Hoare. An Axiomatic Basis for Computer Programming. Communications of the ACM, vol. 12, no. 10, 1969.
43. I.J. Hayes. Applying Formal Specification to Software Development in Industry. IEEE Transactions on Software Engineering, vol. 11, no. 2, 1985.
44. I.J. Hayes. Specification Case Studies. Prentice Hall, 1993.
45. B. Henderson-Sellers. Bridging Metamodels and Ontologies in Software Engineering. In: The Journal of Systems and Software, vol. 84, 2011.
46. B. Henderson-Sellers, C. Gonzalez-Perez. Connecting Powertypes and Stereotypes. In: Journal of Object Technology, vol. 4., no. 7, ETH Zürich, 2005.
47. B. Henderson-Sellers, C. Gonzalez-Perez. The Rationale of Powertype-Based Metamodelling to Underpin Software Development Methodologies. In: Proceeding of APCCM'05 – the 2^{nd} Asia-Pacific Conf. on Conceptual Modelling, vol. 43, Australian Computer Society, 2005
48. M. Himsl, D. Jabornig, W. Leithner, D. Draheim, P. Regner, T. Wiesinger, J. Küng. A Concept of an Adaptive and Iterative Meta- and Instance Modeling Process. In: Proc. of DEXA 2007 - 18^{th} Intl. Conf. on Database and Expert Systems Applications. Springer, 2007.
49. J.E. Hopcroft, R. Motwani, J.D. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 2001.
50. ISO. ISO/IEC 13568:2002. Information technology – Z Formal Specification Notation – Syntax, Type System and Semantics. ISO, 2002.
51. ISO. Intl. Standard ISO/IEC 24744: Software Engineering – Metamodel for Development Methodologies. ISO, 2007.
52. ISO. Information Technology – Object Management Group Object Constraint Language (version 2.3.1), ISO Standard ISO/IEC 19507:2012(E), ISO, 2012.
53. R. Johnson, B. Woolf. Type Object. In: Pattern Languages of Program Design, vol. 3, Addison-Wesley, 1997.
54. G. Kappel, S. Preishuber, E. Prö ll, S. Rausch-Schott, W. Retschitzegger, R. Wagner, C. Gierlinger. COMan – Coexistence of Object-Oriented and Relational Technology. In: Proc. of ER'94 the 13^{th} Intl. Conf. on the Entity-Relationship Approach, Springer, 1994.
55. Kolyang, T. Santen, B. Wolff. A Structure Preserving Encoding of Z in Isabelle/HOL. Springer, 2007.
56. T. Kühne, D. Schreiber. Can Programming be Liberated From the Two-Level Style? Multi-Level Programming with DeepJava. In: Proc. of OOPSLA'2007 – the 22^{th} ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, 2007.
57. T. Kühne: Matters of Metamodeling. In: Software and System Modeling, vol. 5, no. 1, Springer, 2006.
58. C. Lutteroth, D. Draheim, G. Weber. A Type System for Reflective Program Generators. Science of Computer Programming, vol. 76, no. 5, Elsevier, 2011.
59. Per Martin-Löf. Intuistionistic Type-Theory. Bibliopolis, 1984.
60. Z. Mahmood, S. Saeed (Editors). Software Engineering Frameworks for Cloud Computing Paradigm. Springer, 2013.

61. J. Martin, J.J. Odell. Object-Oriented Methods – A Foundation (UML). Prentice Hall, Englewood Cliffs, 1998.
62. C. Morgan. Programming from Specification. Prentice Hall, 1990.
63. B. Neumayr, K. Grün, M. Schrefl. Multi-level Domain Modeling with M-Objects and M-Relationships. In: APCCM'09 – Proc. of the 6^{th} Asia-Pacific Conf. on Conceptual Modeling, Australian Computer Society, 2009.
64. B. Neumayr, M. Schrefl. Comparison Criteria for Ontological Multi-Level Modeling. Presented at: Dagstuhl Seminar on "The Evolution of Conceptual Modeling", Technical Report 08.03., Johannes-Kepler-University Linz, 2008.
65. T. Nipkow, L.C. Paulson, M. Wenzel. Isabelle/HOL – A Proof Assistant for Higher-Order Logic. Springer, 2002.
66. N. Nystrom, X. Qi, A.C. Myers. J& – Nested Intersection for Scalable Software Composition. In: Proc. of OOPSLA'2006 – the 21^{th} ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, 2006.
67. N. Nystrom, S. Chong, A.C. Myers. Scalable Extensibility via Nested Inheritance. In: Proc. of OOPSLA'2004 – the 19^{th} ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, 2004.
68. J.J Odell. Dynamic and Multiple Classification. In: Journal of Object Oriented Programming, vol. 4, no. 9, 1992.
69. J.J Odell. Power Types. In: Journal of Object Oriented Programming, vol. 7, no. 2, 1994.
70. OMG. Object Constraint Language, version 1.1. Rational Software Corporation et al., 1997.
71. OMG. Object Constraint Language, version 2.0, OMG, 2006.
72. OMG. Object Constraint Language, version 2.2, OMG, 2010.
73. OMG. Object Constraint Language, version 2.3.1, OMG, 2012.
74. OMG. Object Constraint Language, version 2.4, OMG, 2014.
75. OMG. OMG Unified Modeling Language – Infrastructure, version 2.4.1. OMG, 2011.
76. OMG. OMG Unified Modeling Language – Superstructure, version 2.4.1. OMG, 2011.
77. OMG. OMG Meta Object Facility – Core Specification, version 2.4.1. OMG, 2011.
78. T. Parr, K. Fisher. LL(*) – the Foundation of the ANTLR Parser Generator. In: Proc. of PLDI'11 – the 32^{nd} ACM SIGPLAN Conf. on Programming Language Design and Implementation, ACM Press, 2011.
79. D.L. Parnas. A Technique for Software Module Specification with Examples. Communications of the ACM, vol. 15, no. 5, 1972.
80. B.C. Pierce. Types and Programming Languages. MIT Press, 2002.
81. G. Plotkin. LCF Considered a Programming Language. Theoretical Computer Science, vol. 5, 1977.
82. R. Razavi, N. Bouraqadi, J.W. Yoder, J.-F. Perrot, R.E. Johnson. Language Support for Adaptive Object-Models Using Metaclasses. In: Computer Languages, Systems & Structures, vol. 31, no. 3–4, 2005.
83. A. Rossini, J. de Lara, E. Guerra, A. Rutle, U. Wolter. A Formalisation of Deep Metamodelling. In: Formal Aspects of Computing, vol. 26, no. 6, 2014.
84. T. Santen. On the Semantic Relation of Z and HOL. In: Proc. of ZUM'98 – The Z Formal Specification Notation. Springer, 1998.

85. B. Schätz et.al. (Eds.). Pre.-Proc. of the Intl. Workshop on Models and Evolution at MoDELS'2010, 2011.
86. , B.V. Selic. On the Semantic Foundation of Standard UML 2.0. In: Formal Methods for the Design of Real-Time Systems, Springer, 2004.
87. P. Simons, I. Niemel, T. Soininen. Extending and Implementing the Stable Model Semantics. In: Artificial Intelligence, vol. 138, no. 1–2, 2002.
88. J.M. Spivey. The Z Notation. Prentice Hall, 1992.
89. J. Lara, H. Vangheluwe. Using AToM as a Meta CASE Tool. In: Proc. of ICEIS'2002 – the 4^{th} Intl. Conf. on Enterprise Information Systems, 2002
90. J. Yoder, R. Johnson. The Adaptive Object Model Architectural Style. In: Proc. of WICSA'02 – the 3^{rd} Working IEEE/IFIP Conf. on Software Architecture, IEEE Press, 2002.
91. E. Zermelo. Untersuchungen über die Grundlagen der Mengenlehre. In: Mathematische Annalen, no. 65, 1908.