Reflective Extension of Object Constraint Languages, Sustainable Constraint Writing and a Symbolic Viewpoint of Modeling Languages

DIRK DRAHEIM, University of Innsbruck

In this technical report we show how to extend object constraint languages by reflection. We do this in terms of the concrete OMG language stack. We extend the OCL (Object Constraint Language) by operators for reification and reflection. We give precise semantics to the extended language OCL\(_R\) by giving the necessary type derivation rules and value specifications. A driving force for the introduced reflection capabilities is the investigation of semantics and pragmatics of modeling constructs. The design of the reflection mechanism of OCL\(_R\) is thoroughly oriented towards abstract syntax. The approach of the OCL\(_R\) semantics specification is declarative. 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.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Design Tools and Techniques; D.2.6. [Software Engineering]: Programming Environments; D.2.11 [Software Engineering]: Software Architectures; D.3.1 [Programming Languages]: Formal Definitions and Theory; D.3.2 [Programming Languages]: Language Classifications constraint and logic languages; D.3.3 [Programming Languages]: Language Constructs and Features; F.3.0 [Logics and Meanings of Programs]: General; F.4.3 [Mathematical Logic and Formal Languages]: Formal Languages; H.1.2 [Models and Principles]: User/Machine Systems; H.2.1 [Database Management]: Logical Design ; H.2.1 [Database Management]: Languages

General Terms: Modeling

Additional Key Words and Phrases: Meta modeling, multi-level modeling, object constraint languages, generative programming, UML, OCL, Z

This technical report has been written on the occasion the 25\textsuperscript{th} DEXA Conference – DEXA'2014 (International Conference on Database and Expert Systems Applications). It is meant as additional material to the talk: D. Draheim. Sustainable Constraint Writing and a Symbolic Viewpoint of Modeling Languages (Invited Talk). In: Proceedings of DEXA'2014, LNCS, Springer, 2014. \textbf{Acknowledgements:} I am deeply grateful to Roland Wagner and Josef Küng for their support and intensive discussions concerning all aspects of theory and practice of information systems.

Author’s address: D. Draheim, University of Innsbruck, Technikerstr. 23, 6020 Innsbruck, Austria, Email: draheim@acm.org, URL: http://draheim.formcharts.org/

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than the author must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission.

The Reflective Extension of Object Constraint Languages

Contents

1 Introduction 5

2 Reflective Extension of OCL 6
   2.1 Meta-Object Access in OCL ................................. 6
   2.2 Declarative Specification of OCL ......................... 8

3 Adequately Modeling for Sets of Sets of Domain Objects 20
   3.1 Plain Class Modeling for Sets of Sets of Domain Objects 20
   3.2 Making Constraints Robust against M1-level Model Updates 22
   3.3 Subtype Externalization ..................................... 22
   3.4 Power Type Externalization .................................. 24
   3.5 Integrated Subtype and Power Type Externalization ........ 25

4 Precise Semantics of UML Power Types 25

5 A Symbolic Viewpoint of Modeling Languages 27
   5.1 The Classic Database Evolution Viewpoint .................. 27
   5.2 The Symbolic Viewpoint .................................... 28
   5.3 The Purely Symbolic Viewpoint .............................. 28

6 Related Work 30

7 Conclusion 31

A UML Meta Model 33

B OCL Types Abstract Syntax 34

C OCL Expressions Abstract Syntax 36

D Z Specification of a Power Types Scenario 37
   D.1 Types Specification ......................................... 37
   D.2 Schema Specification ....................................... 37

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 particular, features for navigating nets of objects. We are interested in adding 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. Furthermore, reflective constraints are an enabler for sustainable constraint writing. Sustainable constraint writing is about making models robust against unwanted updates [19]. More specifically, we can exploit a reflective constraint language, e.g., for:

- **Semantics of modeling languages.** Given meta-level access we can give precise semantics to existing modeling language constructs. We do this for UML power types in this article. Furthermore, we can also 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 for 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 modeled a class A with some n subclasses A1,...,An. Now, you have found some constraints for these n + 1 classes. Imagine that you actually know more. You know that these constraints are an instances of a general constraints pattern that must hold for arbitrary many subclasses, not just n. 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.
  - ...

All of the above items are practically justified. Reflective constraint writing is 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 – the model-oriented, the declarative-axiomatic and the operational viewpoint. 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 so-called OCL_R. We give a declarative semantics for OCL_R. Next, 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. Then, we generalize the found constraints further to give a precise semantics for UML power types. From these discussions, we extract more general notions like sustainable constraint writing and a symbolic viewpoint on modeling languages.

In Sect. 2 we extend the OCL (Object Constraint Language) with reflection to OCL\textsuperscript{R}. In Sect. 3 we exploit OCL\textsuperscript{R} to adequately model sets of sets. In Sect. 4 we exploit OCL\textsuperscript{R} to give precise semantics to UML power types. In Sect. 5 we discuss model evolution, notions of constraints and viewpoints onto modeling languages. We discuss related work throughout the article and summarize further related work in Sect. 6. We end the article with a conclusion in Sect. 7. 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. Appendix D contains a Z specification example.

2. REFLECTIVE EXTENSION OF OCL

The OCL (Object Constraint Language) is syntactically and semantically embedded in 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 [56] as the basis for our language extension. We do neither use the current version OCL v2.4 [60] nor the version OCL v2.3.1 [58], which has been released as ISO standard ISO/IEC:19507 [59]. 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 [56] 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.

2.1. Meta-Object Access in OCL

Standard OCL offers only limited access to the meta-level. This meta-level access is restricted to introspection, no reflection 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. The complete list of these operations is given in Fig. 2.1. 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 [55]. 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. 2.1:

1. Names of subclasses of given type t.
2. Names of attributes of a class navigable via an association from a given type t.
3. The subclasses of a given type t.
4. All classes of the user model.
5. All classes of the user model that have no subclasses.
6. Number of classes in the user model.
7. Test, whether all attributes of all classes have a setter- and a getter-method.
8. Test, whether all attributes of all objects of all classes are initialized.
9. The sum of all Integer attributes of all objects of all classes.
The Reflective Extension of Object Constraint Languages

Properties of all objects, i.e., objects \( o : \text{OclAny} \):
- \( o.\text{oclIsTypeOf}(t:\text{OclType}):\text{Boolean} \) – true iff \( o \) : \( t \)
- \( o.\text{oclIsKindOf}(t:\text{OclType}):\text{Boolean} \) – true iff \( o \) : \( t \)
- \( o.\text{oclInState}(s:\text{OclState}):\text{Boolean} \) – Test for state machine state.
- \( o.\text{oclIsNew}():\text{Boolean} \) – Postcondition test for object creation.
- \( o.\text{oclclassType}(t:\text{OclType}):\text{instance of Classifier} \) – Type casting operation.

Properties of meta objects \( t \) representing user-defined types, i.e., objects \( t : \text{OclType} \):
- \( t.\text{name}:\text{String} \) – The name of the type \( t \).
- \( t.\text{attributes}:\text{Set(String)} \) – The set of names of the attributes of \( t \).
- \( t.\text{associationEnds}:\text{Set(String)} \) – Names of association ends navigable from \( t \).
- \( t.\text{operations}:\text{Set(String)} \) – The names of the operations of \( t \).
- \( t.\text{supertypes}:\text{Set(OclType)} \) – The set of all direct supertypes of \( t \).
- \( t.\text{allSupertypes}:\text{Set(OclType)} \) – The set of all supertypes of \( t \).
- \( t.\text{allInstances}:\text{Set(type)} \) – The set of all instances of type \( t \).

Fig. 1. OCL inbuilt meta-level access.

All of the above constraint expressions (1) through (9) can be expressed by the OCL-extension OCL\(_R\) that we will define in Sect. 2.2. The several constraint expressions express different levels of sophistication. Constraints (1) and (3) can be made possible by augmenting the list of inbuilt OCL expressions in Fig. 2.1. 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 programming language community distinguishes between reification and reflection. Also with respect to meta-level access in modeling languages this distinction is important. Here, reification is about making meta-data accessible to the modeler. Reification allows for introspective access to the meta-level. The concepts of reification and introspective access can be identified. Reflection is more. It is based on reification. Reflection turns data gained from introspective access into modeling elements, i.e., it materializes reified meta data in modeling. You have the options to either consider reflection as consisting of a reification step and a materialization step, or to consider reflection as the materialization step itself. In any case, reification is a necessary precondition for reflection. We freely switch between both viewpoints. We use the latter viewpoint in the definition of the reification and reflection operators \( \Phi \) and \( \Psi \) in Sect. 2.2.

The OCL meta-level access offers only a limited form of reification. The OCL standards [58] explicitly state that OCL does not support the reflection capabilities of the MOF (Meta Object Facility) [63]. 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.

Out the above properties for types, the property allInstances is particularly important. All of the following OCL expressions 1 through 3 have the same semantics:

\[
\text{context Person inv. } \text{age} \geq 16 \quad (1) \\
\text{context Dog inv. } \text{Person.allInstances} \rightarrow \forall \text{allInstances}(\text{age} \geq 16) \quad (2) \\
\text{context Dog inv. } \text{Person.allInstances.age} \geq 16 \quad (3)
\]

In OCL, the user defined type Person in (1) is said to provide the context for the evaluation of the expression \( \text{age} \geq 16 \), which actually is a shortcut for \( \text{self}.\text{age} \geq 16 \). The type Person in (2) is said to be a meta object and the application of the property
allInstances is said to be an example of meta access. Now, the target of this discussion is to create awareness that the concepts of contexts and meta objects are completely exchangeable. First, we need to understand that (3) is a correct OCL expression. The type of the expression Person.allInstances is Set(Person), so it is type-correct to apply the property age to this expression. Now, we can see that the role of Person in (1) and (3) are exactly the same. In (1) you can consider Person a meta object with \( \lambda x. \lambda y.(\text{context } x \text{ inv } y) \) having the same semantics as \( \lambda x. \lambda y.(x.\text{allInstances}(y)) \). In (3) you can consider Person.allInstances to provide context for the evaluation of \( \text{age} \geq 16 \). Henceforth, we will therefore write constraints without explicitly giving a context, even if this is not standard. Consider the following constraint:

\[
\text{Person.allInstances} \geq 16
\]

Constraint (4) is complete. The type Dog in (3) is not needed in the subsequent expression, so it can be dropped to yield (4).

2.2. Declarative Specification of OCL_R

Without loss of generality, we define OCL_R as an M1-level language, i.e., we define the reification operators \( \Phi \) and \( \Psi \) 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. 4 as an example for this. Therefore, there is no need to explicitly generalize the current definitions from a M2-level perspective.

2.2.1. Notation 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 in the definition of the semantics of OCL_R. 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 [61]; 7.12 as background architecture, see also Fig. 2. Syntax and semantics of the UML meta meta model, the UML meta model and OCL are taken as granted as defined in [62; 61; 63; 56; 58].

We denote the UML meta model by \( \mathcal{M}_2 \). Similarly, we denote the UML meta meta model by \( \mathcal{M}_3 \). We denote the set of all primitive values by \( \mathcal{P} \). The set \( \mathcal{P} \) 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 \( \mathcal{O} \). We denote the set of all attribute values by \( \mathcal{V} = \mathcal{P}(\mathcal{P} \cup \mathcal{O}) \). The power set in the definition of \( \mathcal{V} \) is necessary, because in our notation all attributes are many-valued. We denote the set of attribute names or labels by \( \mathcal{L} \). We denote the set of finite subsets of a set \( M \) by \( \mathcal{P}(M) \). We denote the set of all objects \( \mathcal{A} \) as the Cartesian product of object identifiers \( \mathcal{O} \) to \( \mathcal{O} \)-compatible finitely indexed sets of attribute types, i.e:

\[ \mathcal{A} = \mathcal{O} \times \mathcal{L} \]

\[^1\]We have never seen in any of the OMG specifications or OCL text books, that the objects in the result set of a property call to allInstances are directly addressed but always only via collection functions.
\[ \mathfrak{A} = \mathfrak{O} \times \bigcup_{\mathcal{L} \in \mathfrak{F}(\mathfrak{L})} \{ \mathfrak{I}_1 \}_{l \in \mathcal{L}} \]

With \( \mathfrak{A} \) objects are denoted as records [1; 10], or to be precise, object values are denoted as records, and objects are formed as an object reference to an object value. We use the usual notation for records, i.e., \( \langle \text{oid} \rightarrow \langle l \mapsto x_i \rangle \rangle_{l \in \mathcal{L}} \) or also \( \langle \text{oid} \rightarrow \langle l_1 \mapsto x_{i_1}, \ldots, l_n \mapsto x_{i_n} \rangle \rangle^2 \). Note that \( \mathfrak{A} \) is the full extension of the meta level hierarchy, i.e., the collection of all potential objects that can be materialized in system states\(^3\). We use \( \mathfrak{I}_i \) to denote the set of all objects at meta-level \( M_i \), the \( M_i \)-level objects for short. We call a subset \( m \subset \mathfrak{I}_i \) of \( M_i \)-level objects an \( M_i \)-level model or model for short iff \( m \) is a partial function, i.e.:

\[ m \in \mathfrak{O} \longrightarrow \bigcup_{\mathcal{L} \in \mathfrak{F}(\mathfrak{L})} \{ \mathfrak{I}_1 \}_{l \in \mathcal{L}} \]

Next, we define value identity of objects with respect to given models. Given models \( m, n \) and objects \( o \in m, p \in n \), with \( o = \langle \text{oid} \rightarrow \langle l \mapsto x_i \rangle \rangle_{l \in \mathcal{L}} \) and \( p = \langle \text{pid} \rightarrow \langle l \mapsto y_j \rangle \rangle_{l \in \mathcal{L}} \) 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\(^5\):

\[
\begin{align*}
(i) & \quad x_i \notin \mathcal{D} \Rightarrow (x_i = y_i) \\
(ii) & \quad x_i \in \mathcal{D} \Rightarrow (\exists \beta : x_i \leftrightarrow y_i, \forall x' \in x_i.m(x') \equiv n(\beta(x'))) (5)
\end{align*}
\]

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 a set of \( M_i \)-level objects \( O \subset \mathfrak{I}_i \), and a \( M_{i+1} \)-level model \( M \subset \mathfrak{I}_{i+1} \), we use \( O :: M \) to denote the fact, that \( O \) is a well-formed instantiation of model \( M \). Given a function \( f : A \rightarrow B \), we denote the lift of \( f \) by \( f^L : \mathcal{P}(A) \rightarrow \mathcal{P}(B) \), which is defined as usual. If it is clear from the context, we overload a function symbol \( f \) with the denotation of its lift \( f^L \). We define the embedding of the UML meta model into the UML meta model \( e : M_1 \leftarrow M_2 \)

\(^2\)As we have defined \( \mathfrak{A} \) it contains many 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 \( \mathfrak{O} \) which is completely flat. This does not harm, because the definitions in Sect. 2.2.1 are not about semantics, but about notation. We will ensure the well-formedness of object and models later by the introduction and exploitation of two operations \( - \_\_L \) and \( - \_\_ \).

\(^3\)The set \( \mathfrak{A} \) is a forgetful viewport. It models all aspects that are needed in the upcoming semantic definitions. In particular it forgets ordered association ends.

\(^4\)Further note, that with \( \mathfrak{A} \) we have a formed one out of three basic viewports onto the elements of the meta level hierarchy. In \( \mathfrak{A} \) we combine information on primitive-typed attributes with object references into a record. Another viewport is do 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. 3, diagram (v). The third option is to represent elements as pure nets of object identifiers with primitive values as leaves. We have discussed the latter option in form-oriented analysis as so-called parsimonious data model [27]. Once more note, that the purpose of Sect. 2.2.1 is not to formalize UML semantics. It is merely about establishing notation for the existing meta level framework to be exploited in upcoming sections.

\(^5\)The definition of \( \equiv \) is a partial specification only. It is only defined for objects that share the same set of labels \( I \). On the other hand, the definition is only complete for well-typed and at the same time identically typed pairs of objects. This does not harm. In the sequel we only work with well-formed models, so that both of the above conditions are fulfilled and therefore the definition is complete with respect to all considered objects. The advantage of the partial specification is that we become at hand a particularly lightweight notation and specification thereof. This is also the rationale behind the notation characteristics discussed in footnotes 2 and 4.
by \( \iota = \{(x, y) \mid x \equiv y\} \) – see Fig. 3. Next, we introduce the meta model reification operator \( \Phi \). First, we define the set of meta model reification operators \( \Phi \) as the set of embeddings \( \phi : \mathcal{M}_2 \rightarrow \mathcal{A}_1 \) for which it holds true that (i) \( \phi(\mathcal{M}_2)^* = \mathcal{M}_2 \) and (ii) for all \( m \in \mathcal{M}_2 \) it holds true that \( \phi(m) = m \). Then we define \( \Psi \) as an arbitrary but fixed element of \( \Phi \), i.e., \( \Psi \in \Phi \). In the sequel, we refer to \( \Psi \) as the meta model reification operator. On the basis of the meta model reification operator \( \Phi \) we introduce the model reification operator \( \Psi \). We define the set of model reification operators \( \Psi \) as the set of embeddings \( \psi : \mathcal{A}_1 \rightarrow \mathcal{A}_1 \) for which it holds true that, given any \( \mathcal{M}_1 \)-level model \( m \in \mathcal{A}_1 \), it holds that (i) \( \psi(m)^* = \Phi(\mathcal{M}_2) \) and (ii) for all \( m \in \mathcal{M}_2 \) it holds true that \( \psi(m) = m \). Again, we define \( \Psi \) as an arbitrary but fixed element of \( \Psi \), i.e., \( \Psi \in \Psi \). In the sequel, we refer to \( \Psi \) as the model reification operator. Given a function \( f : A \rightarrow B \) we denote the reversal, as usual, as \( f^{-1} : B \rightarrow \mathcal{P}(A) \). Given an embedding \( i : A \rightarrow B \) we use \( \perp \) to denote, as usual, an undefined value i.e. \( i(a) = \perp \) for a \( \notin i^{-1}(B) \). We model bags as functions to the ordinals, i.e., given a set \( T \), we model the bags of \( T \) as \( B(T) = T \rightarrow \mathcal{N}_0 \). We define bag union of bags \( b, c : B(T) \) as \( b \cup^\lambda \ c = \lambda s.(b(s) + c(s)) \). Similarly, we define the big bag union \( \cup \). Given an embedding \( i : S \rightarrow T \), we define the bag lift \( i^* : B(S) \rightarrow B(T) \) for each bag \( b : B(T) \) by \( i^*(b) = (\lambda s.b(i^{-1}(s))) \). Given a set \( T \), we model the sequences of \( T \) as \( S(T) = \mathcal{N}_0 \rightarrow T \). Given an embedding \( i : S \rightarrow T \), we define the sequence lift \( i^* : S(S) \rightarrow S(T) \) for each sequence \( s : B(T) \) by \( i^*(s) = (\lambda n.i(s(n)))s \).

2.2.2. Reification for Constraint Languages

A straightforward approach to extend OCL by introspective and reflective features was to rewrite the semantics of OCL in terms of switching between 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, despite the formulation of new well-formedness rules for a new OCL\_R reflection expression and the semantics definition of the newly introduced 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. 2.

The operator \( \Phi \) 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\(^6\). This is immediately possible because of the bootstrap approach of the UML

\(^6\) Intuitively, we can say that we use the operator \( \Phi \) to copy the UML meta model and add it to the user-defined model at level M1. Actually, the definition of \( \Phi \) as provided in 2.2.1 is completely declarative. We have defined the set of object references \( \mathcal{D} \) as an abstract data type. We keep \( \mathcal{D} \) completely opaque, i.e., we do not define operations for the creation of object handles or the construction of objects. The value identity \( \equiv \) that we have
specification, i.e., the UML meta model is specified in a core language that is itself a part of the UML language. Each UML meta model expression is therefore immediately a correct M1-level model expression. This fact is also indicated by the embedding $\iota: \mathcal{M}_3 \rightarrow \mathcal{M}_2$ of the UML meta meta model into the UML meta model. Figure 2 shows the overall scenario of reification and reflection with OCL, whereas Fig. 3 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. It is important to understand that in the terminology of the UML, see, e.g., Figure 7.8 in [61], the reified meta model is part of the user model. This fact eases, from the perspective of the OCL constraint writing, the introduction and understanding 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 model in cases where disambiguation seems to be important – see Fig. 3. 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 $\text{OCL}_R$. Therefore, by definition, there is no conflict with other software artifacts of tools. The target of this article is not the particularly well design of a new constraint language. 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. We designed the reflective constraint language as an enabling tool for the analysis of important semantic issues in information system modeling.

With the reification of the UML meta model at level $\mathcal{M}_1$ we are prepared for introspective access. Given a M2-level 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 $\text{Class}$ in your model, and this class must not become into conflict with the reified meta model type $\text{Class}$. 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 [62] §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 separate and use names of reified types without $\langle \_ \rangle\downarrow$ without harm. Therefore, in the examples in further sections, we will simply use names of meta model types in M1-level constraints. However, in this section we stay with the $\langle \_ \rangle\downarrow$ for reasons of preciseness and clarity. We step further by reifying the user user model at level M0. Now, the appropriate classes of the reified UML meta model are available for this purpose at level M1. The operator $\Psi$ re-instantiates each model element $e_1::e_2$ as the value identical model element $e_1::\Psi(e_2)$.  

defined in (5) is object net structural equality up to object references. Then, also the definition of $\Phi$ is free from concrete object construction mechanisms. We can assume the existence of $\Phi$ (set theory) and therefore all semantics definition in this section, in particular, the typing rules are founded. If you find it helpful, you can think of the act of selecting and fixing an arbitrary $\phi$ from $\Phi$ as the act of copying model $\mathcal{M}_0$ to level M1.

$^7$Same comment for $\Psi$ as for $\Phi$ in footnote 6

Fig. 3. The OCL\textsubscript{R} reification mechanism
2.2.3. Reflection for Constraint Languages

What we have already achieved now is full introspective access onto the user-defined types, even without extension of the OCL syntax. However, yet the crucial step is missing, i.e., gaining access onto model elements of the user model via the reified data. Therefore, many interesting constraints are yet not possible to write. To get the point, first have a look at a correct constraint example that exploits the reified data:

\[ \langle \text{Class} \rangle . \text{allInstances.ownedProperty} \rightarrow \text{size} \leq 20 \] (6)

For the purpose of easy reference, we have added a specification of the OCL type system in Appendix B and a crucial chunk of the OCL abstract syntax specification in Appendix C. Constraint (6) evaluates to true if all user model classes have at most twenty properties. The examples in this section serve as merely demonstration of the OCL and OCL_R mechanics. They are not meant to present examples of domain knowledge. In later section we will discuss many exploitations of the reflective features added to the constraint language. In that sense, it is not clear, what the constraint (6) should be good for. However, against of the background of today's established modeling practice it is fair to say that (6) 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 – see also Fig. 3:

\[ \langle \text{Class} \rangle . \text{allInstances} \rightarrow \text{select(name = "Person").ownedProperty} \rightarrow \text{select(name = "pet").type} \rightarrow \text{includes(name = "Dog")} \] (7)

The constraint in (7) checks whether the class Person is associated, via a role pet to a class Dog. The semantics of the constraint (7) is equal to the semantics of the following usual OCL constraint that works without the new reification capabilities:

\[ \text{context Person inv: petoclIsTypeOf(Person)} \] (8)
\[ \text{Person.allInstances.petoclIsTypeOf(Person)} \] (9)

See, how constraint (8) immediately queries the property pet, whereas (7) must navigate the two additional links ownership and type of UML meta model to reach the target Person.

Now, let us have a look at the following invalid, i.e., ill-typed, constraint expression:

\[ \langle \text{Class} \rangle . \text{allInstances} \rightarrow \text{select(name = "Person").allInstances.age} \geq 16 \] (10)

The expression \( \langle \text{Class} \rangle \) has type OclType. Therefore, as part of the limited meta data access capabilities of OCL, it is possible to apply the method allInstances to this expression. The expression ... allInstances has type Set(\( \langle \text{Class} \rangle \)). The expression ... \( \text{select(name = "Person")} \) again has type Set(\( \langle \text{Class} \rangle \)), actually, it evaluates to the one-set element consisting of exactly the person object. Now, when we try to invoke the allInstances method to this expression, we provoke a type error. 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:

\[ \langle \text{Class} \rangle . \text{allInstances.allInstances} \] (11)

Again, the expression in (11) is not well-typed. However, intuitively, constraint expressions like (10) and (11) have semantics. For example, we intend (11) to mean the set of all instances of all classes of the user model. In \( \text{OCL}_R \) we will be able to write constraint expressions like (10) and (11) in due course.

### 2.2.4. Informal Description of \( \text{OCL}_R \) Reflection

This section provides some examples of \( \text{OCL}_R \) expressions and an informal description of their semantics. A precise semantics is given in Sect. 2.2.5 in terms of a type derivation rules and value specifications. The following constraints check whether each instance of the class \( \text{Person} \) has an attribute value over \( 16 \) for its attribute \( \text{age} \):

\[
\llparenthesis \langle \text{Class} \rangle \downarrow \text{allInstances} \rightarrow \text{select}(\text{name} = \text{"Person"}) \rrparenthesis \downarrow \text{allInstances.age} \geq 16
\quad (12)
\]

We introduce a reflection construct \( \langle \_ \rangle \uparrow \) as the crucial extension to \( \text{OCL} \). There is a substantial difference between the reification notation \( \langle \_ \rangle \downarrow \) introduced earlier and the reflection construct \( \langle \_ \rangle \uparrow \). Up to now, the reification notation is not an extension to \( \text{OCL} \), whereas the reflection construct \( \langle \_ \rangle \uparrow \) is. The reflection construct is added to the abstract syntax. Instead, the reification notation is merely a convention. It ensures that the namespaces of reified M2-level elements are kept separated from the user-defined types. Practically, the reification notation is added to the concrete syntax of \( \text{OCL} \).

Informally, the semantics of the reflection construct is the reversal of a reification. In (12) the reflection is meant to receive the one-element set containing the reified \( \text{Person} \) object and yields the one-element set containing the \( \text{Person} \) meta-object it has been reified from. More precisely, the reflection construct in (12) turns an expression of type \( \text{Set}(\langle \text{Class} \rangle \downarrow) \) into an expression of type \( \text{OclType} \). See how this works in the following example type derivation. Than, as usual, \( \_ : \text{T} \) meant that a sub expression has type \( \text{T} \) – see also Appendix B as a reference for \( \text{OCL} \) types:

\[
\llparenthesis \langle \text{Class} \rangle \downarrow \text{allInstances} \rightarrow \text{select}(\text{self}.\text{name} = \text{"Person"}) \rrparenthesis \uparrow \text{allInstances}
\quad (13)
\]

The result of the reflection (xi) in (13) has type \( \text{Set}(\text{OclType}) \). With standard \( \text{OCL} \) this result cannot be immediately exploited in a property call \( \text{o.p} \). The \( \text{OCL} \) semantic description of property calls implicitly implies that a property can only be applied to object of a single \( \text{classifier} \)\(^8\). We have two options to deal with this. We can change \( \text{OCL} \) 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 \( \text{OCL} \).

\(^8\)The \( \text{OCL} \) specification states [56]: A \text{PropertyCallExpression} is a reference to an \text{Attribute} of a \text{Classifier} defined in a \text{UML} model. It evaluates to the value of the attribute.
that turns a one-element set into the contained element. With \( \langle M \rangle \) we denote exactly this operation. With respect to semantics, the solution based on \( \langle M \rangle \) 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. All the constraints in Sect. 3 can be expressed with the apparatus that we have established so far. However, to gain full reflective power, we need a means to apply a property to objects of more than one, arbitrary classes. With arbitrary we mean that the property under consideration is not inherited via a shared super type. 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 object of more than class. Have a look at the following example that introduces also some ad-hoc concrete syntax for the special case of a fixed number of classes:

\[
\begin{array}{c}
\text{Person, Dog} \\
\text{self.age} \geq 5
\end{array}
\]  

(14)

Of course, in (14) we assume that both the class Person and the class Dog both define an Integer attribute. Fortunately, we do not need to change the syntax of OCL to make expression like (14) possible. A property call expression has another OCL expression as its source. In general, it is possible that OCL expressions have type \( \text{Set(OclType)} \) – see Appendix B. So it is a self-restriction to allow only for the application to a single type. The semantics of an expressions like (14) 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.

Now, given that the semantics of property call expressions is generalized to many classes, we can use OCL\( \mathcal{R} \) to express even more general cases, in which we apply a property to an arbitrary number of classes. The following constraint evaluates the age attribute for all objects of a class model. Again, the expression can be considered well-formed only, if all classes in the current user-model define an age attribute of correct type:

\[
\langle\langle \text{Class} \rangle\rangle.\text{allInstances}.\text{allInstances}.\text{age} \leq 16
\]  

(15)

Next, we introduce a reification operator that we denote as before with \( \langle\langle \rangle\rangle \). Conceptually, this reification operator is different from the reification notation defined so far. The reification notation defined so far represents the reification operator \( \Phi \) and works between meta levels M2 and M1. In doing so it is a mere notational convention as explained before. The newly introduced \( \langle\langle \rangle\rangle \) is defined for user user model types, and therefore works between level M1 and M0. The semantics of the operator is straightforward. It hands over the reification operator \( \Psi \) to the modeler so that he gains direct introspective access to the user-defined types. The following example constraint including the crucial type derivation shows how this works:

\[
\text{Class.allInstances} \rightarrow \text{select}(\text{self}) = \langle\langle \text{Person} \rangle\rangle.\text{allInstances}.\text{age} \geq 16
\]  

(16)

---

9 The notation \( \langle M \rangle^{-1} \) would be even more straightforward, because element picking is the reversal of element embedding, i.e., \( \langle\{x\} \rangle = x \). Furthermore note, that \( \langle M \rangle \) is only partially defined, i.e., it is defined only for one-element sets.
So far, each OCL expression of type \langle Class \rangle_t or \langle Set \langle Class \rangle_t \rangle 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 following example:

\[
\text{Person}
\text{.allInstances } \rightarrow \text{forall (}
\text{self.}
\langle \text{Class} \rangle_t \text{.allInstances } \\
\rightarrow \text{select(name = "Person" ).ownedAttributes } \\
\rightarrow \text{select(name = "age")}
\rangle
\] \geq 16 \tag{17}

In (17), 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 17\textsuperscript{10}:

\[
\begin{array}{c}
\text{(v)} \\
\text{(vi): Integer}
\end{array}
\]

As a next step, we can also generalize the semantics of the property call expressions further to the application of a set of attributes to a class or a set of classes, see the following example showing again some ad-hoc syntax, with obvious semantics:

\[
\text{Person, Dog } \\
\text{self.(age, weight) } \geq 0 \tag{19}
\]

In an reflective extension to OCL we can exploit this new semantics in an expression like the following:

\[
\langle \langle \text{Class} \rangle_t \rangle_t . \text{allInstances} \rangle \uparrow . \text{allInstances} . \langle \langle \text{Class} \rangle_t \rangle_t . \text{allInstances}.\text{ownedAttributes} \\
\rightarrow \text{select(type = Integer)}
\] \geq 0 \tag{20}

The constraint 20 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. In the application of the OCL\textsubscript{R} we are free to omit all the special syntax for reification, reflection and also element picking, i.e., \langle \rangle, \langle \rangle and \langle \rangle unless we do not need it to for the disambiguation of expressions. Therefore we will not use the special syntax.

\textsuperscript{10}Technically, the property name \texttt{p} in a property call expression \texttt{op} is not a proper OCL expression, in the sense that it does not have a type. This does not harm. It is exactly, the reflection construct that opens a context for typed expressions. See how the type of (v) in 17 is immediately consumed by the type derivation with rule (28) from Sect. 2.2.5, i.e., how the typing of (vi) is not needed in the type derivation.
syntax beginning from Sect. 3 however must stay with it in the next section on precise semantics.

2.2.5. Precise Semantics of OCL\textsubscript{R} Reflection

In the definitions of this section we make extensive use of the notation introduced in 2.2.1 and heavily rely on the concepts defined earlier in this section, e.g., the reification and reflection operators $\Phi$ and $\Psi$. We define necessary the well-formedness rules strictly as augmentations to the existing notion of UML and OCL type correctness.

Typing Notation

Given a UML, OCL or OCL\textsubscript{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 [10; 65; 40] 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:

$$
\begin{align*}
\vdash e_1 : T_1 & \ldots \vdash e_n : T_n \quad C_1 \ldots C_m \\
\vdash e' : T' &
\end{align*}
$$

(21)

Given that we have already derived typings $e_i : T_i$ and further conditions $C_i$ hold true, a typing rule of kind (21) 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.

Semantic Bracketing

Furthermore, we use so-called semantic bracketing to define the value of expressions. Given an expression $M$ we use $\llbracket M \rrbracket$ 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 $\llbracket e_1 \ldots e_n \rrbracket$ as a value $\llbracket e_1 \rrbracket \ldots \llbracket e_n \rrbracket$ with $\llbracket e \rrbracket$ 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.

OCL\textsubscript{R} Language Constructs and Types

The language OCL\textsubscript{R} results from extending OCL by expressions for reflection and reification. As a minor point, a language construct for picking an element out of one-element sets is introduced. Fig. 4 shows the OCL\textsubscript{R} meta model. Elements of the OCL are shown in grey, whereas the new language constructs are shown in black. A major chunk of the OCL specification is provided in Fig. 10 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 operation refers another OCL expression as its reflected expression. A reification expression refers to a type expression as its reified type. The types of OCL\textsubscript{R} are the same as the types for OCL v.2.0 – see Appendix B Fig. 9.

Typing Rules and Values

We elaborate the semantics for three kind of expressions in this article, that cover the full range of OCL\textsubscript{R} semantics, i.e., type expressions, property call expressions and enumeration literal expressions. An expression of a reified data type, i.e., a reified data expression, yields an OclType-expression after reflection. Reflection recovers the M1-level value of reified data:
Examples for expressions that have \( \Phi(Class) \) as type and not a type of the form \( C(\Phi(Class)) \) for a collection \( C \) are the iterator variables in OCL loop expressions. A further example is the result of correctly applying the element picking operation, that can be applied to a collection of any type, including reified M2-level types. For all kinds of collection \( C \), i.e., \( Set \), \( Bag \) and \( Sequence \) we have that:\(^{11}\)

\[
\begin{align*}
\vdash e &: \Phi(Class) \\
\vdash \langle e \rangle^t &: OclType \\
\langle e \rangle^t &= \left[ \Psi^{-1}(\{e\}) \right] 
\end{align*}
\]  

\( \Psi^{-1} \) is defined for collections, bags, and sets. The lift on sets is the usual power set lift. We have defined lifts of embeddings for bags and sequences in Sect.2.2.1.

\(^{11}\)The definition of \( \langle m \rangle \) is only for convenience. It is already ensured by the well-formedness rule 24 that it is only applied to one-element sets, so that it also can be defined by \( \langle \{x\} \rangle = x \).
Table I. The collection type combinator \( \oplus \).

<table>
<thead>
<tr>
<th>( \Phi ) (T)</th>
<th>Set</th>
<th>Bag</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>Set(T)</td>
<td>Bag(T)</td>
<td>Sequence(T)</td>
</tr>
<tr>
<td>Bag</td>
<td>Bag(T)</td>
<td>Bag(T)</td>
<td>Bag(Sequence(T))</td>
</tr>
<tr>
<td>Sequence</td>
<td>Sequence(Set(T))</td>
<td>Sequence(Bag(T))</td>
<td>Sequence(Sequence(T))</td>
</tr>
</tbody>
</table>

We turn to property call expressions now. For all user-defined types \( T \) we define:

\[
\begin{align*}
\vdash & \ o : T_1 \quad p : \Phi(Property) \quad p.class = \Psi(T_1) \quad p.type = T_2 \\
& \vdash \ o.(p) : \Psi^{-1}(T_2) \\
\end{align*}
\]

\[
\llbracket o.(p) \rrbracket = \llbracket o.\Psi^{-1}(\llbracket p \rrbracket) \rrbracket 
\]

(28)

(29)

For all kinds of collection \( C \) we define:

\[
\begin{align*}
\vdash & \ o : T_1 : C \quad p : C(\Phi(Property)) \quad p.class = \Psi(T_1) \quad p.type = T_2 \\
& \vdash \ o.(p) : C(\Psi^{-1}(T_2)) \\
& p : Set(\Phi(Property)) \Rightarrow \llbracket o.(p) \rrbracket = \bigcup_{p' \in (\Psi^{-1})^+(\llbracket p \rrbracket)} \llbracket o.p' \rrbracket \\
& p : Bag(\Phi(Property)) \Rightarrow \llbracket o.(p) \rrbracket = \bigcup_{p' \in (\Psi^{-1})^+(\llbracket p \rrbracket)} \llbracket o.p' \rrbracket \\
& p : Sequence(\Phi(Property)) \Rightarrow \llbracket o.(p) \rrbracket_\lambda = \lambda i \in \mathbb{N}_0. \llbracket o.\llbracket p_i \rrbracket \rrbracket \\
\end{align*}
\]

(30)

(31)

(32)

(33)

Given the collection type combinator \( \oplus \) defined in Table I, for all kinds of collections \( C_1 \) and \( C_2 \) the following well-formedness rule must apply:

\[
\begin{align*}
\vdash & \ o : T_1 : C_1(Class) \quad p : C_2(\Phi(Property)) \quad p.class = \Psi(T_1) \quad p.type = T_2 \\
& \vdash \ o.(p) : (C_1 \oplus C_2)(\Psi^{-1}(T_2)) \\
\end{align*}
\]

(34)

We do not detail out the value specification for the case (34). We turn to enumeration literals now. We define that:

\[
\begin{align*}
\vdash & \ e : \Phi(EnumerationLiteral) \\
& \vdash \ (e) : EnumerationLiteral \\
\end{align*}
\]

(35)

The value specification for enumeration literal expressions is identical to the one in case of type expression, i.e., equation (23).

In the definition of \( \text{OCL}_R \) we have relied on the existing UML and OCL semantics defined in [56; 61] 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. It 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.

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 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.

3. ADEQUATELY MODELING FOR SETS OF SETS OF DOMAIN OBJECTS

The aim of this section is to discuss ways of adequately modeling sets of sets of domain objects. Fig. 5 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. 6 show further, more elaborate means to model the intended domain. Diagram (i) in Fig. 6 is, basically, the conceptual model copied from Fig. 5. It is included into Fig. 6 for an important presentation issue, i.e., in order to complete the full power type construction diamond.

---

13 The necessary OCL$_R$ constraints are given in the text as (36)–(42).
3.1. Plain Class Modeling for Sets of Sets of Domain Objects

The M0- and M1-level models in Fig. 5 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\(^{14}\) 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\(^{15}\) by the following OCL \(_R\) constraint:

\[
\text{Dog.allInstances.genus} \rightarrow \text{asSet} \rightarrow \text{size} \leq 1 \\
\text{(36)}
\]

In order to make the semantics of the constraint more explicit, we specify the same constraint now in a different, let's say, axiomatic style.

\[
\text{Dog.allInstances forall (dog_1, dog_2 |} \\
\text{dog_1.genus = dog_2.genus} \\
\text{or dog_1.genus \rightarrow asSet \rightarrow size = 0} \\
\text{or dog_2.genus \rightarrow asSet \rightarrow size = 0)} \\
\text{(37)}
\]

A reader might say that constraint (36) is superfluous, because the UML specification states that a class a 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 (36) is one means to make the semantics of class attributes precise. However, the purpose of constraint (36) in this paper is different, we want it to be at hand for comparison with the constraints for subset-global attributes like (38) to (41) in the sequel.

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:

\[
\text{Dog.allInstances \rightarrow select(breed = Collie).breednumber} \rightarrow \text{asSet} \rightarrow \text{size} \leq 1 \\
\text{Dog.allInstances \rightarrow select(breed = Collie).breedAvgAge} \rightarrow \text{asSet} \rightarrow \text{size} \leq 1 \\
\text{Dog.allInstances \rightarrow select(breed = Pitbull).breednumber} \rightarrow \text{asSet} \rightarrow \text{size} \leq 1 \\
\text{Dog.allInstances \rightarrow select(breed = Pitbull).breedAvgAge} \rightarrow \text{asSet} \rightarrow \text{size} \leq 1 \\
\text{(38) (39) (40) (41)}
\]

\(^{14}\)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.

\(^{15}\)We have chosen the current formalization of the concept of class attribute, because it is amenable in a straightforward manner on basis of OCL.

Later, when we consider subtype externalization in Sect. 3.3 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 constraint:

\[
\text{Dog.allInstances.breed} \rightarrow \text{asSet} \rightarrow \text{size} = \text{Dog.allInstances.breednumber} \rightarrow \text{asSet} \rightarrow \text{size}
\] (42)

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.

3.2. Making Constraints Robust against M1-level Model Updates

In Sect. 3.1 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. 6. However, the central theme of this section turns out to be the question how to make constraints robust against model updates at level M1.

The constraints (38) to (41) 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., Sausage Dog, is introduced by the modeler into the enumeration type, the subset-global attribute for breed numbers is not longer under the auspices of appropriate constraints, because the existing 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. The following constraint generalizes the breed number constraints (38) to (41) by abstracting from the concrete values in the enumeration type so that the constraint becomes robust against unwanted M1-level updates:

\[
\text{Enumeration.allInstances} \rightarrow \text{select(Breed).ownedLiteral} \rightarrow \text{forAll(breedId)}
\]

\[
\text{Dog.allInstances} \rightarrow \text{select(breedId = breedId).breednumber} \rightarrow \text{asSet} \rightarrow \text{size} \leq 1
\] (43)

Now: how to introduce a new breed? 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.

3.3. Subtype Externalization

Model (ii) in Fig. 6 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

\[\text{The necessary constraints are given in the text.}\]
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. 6. 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 \[53; 52\], so in general you would also need to specify the subclasses for which the intended properties are considered as set-global.

The constraints (38) to (41) 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:

\[
\begin{align*}
\text{Collie.allInstances.breednumber} &\rightarrow \text{asSet} \rightarrow \text{size} \leq 1 & (44) \\
\text{Collie.allInstances.breedAvgAge} &\rightarrow \text{asSet} \rightarrow \text{size} \leq 1 & (45) \\
\text{Pitbull.allInstances.breednumber} &\rightarrow \text{asSet} \rightarrow \text{size} \leq 1 & (46) \\
\text{Pitbull.allInstances.breedAvgAge} &\rightarrow \text{asSet} \rightarrow \text{size} \leq 1 & (47)
\end{align*}
\]

The sustainable constraint (43) can now be restated 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:

\[
\begin{align*}
\text{GeneralizationSet.allInstances} &\rightarrow \text{select(Breed).generalization specific} \\
&\rightarrow \forall (\text{allInstances.breednumber} \rightarrow \text{size} \leq 1)
\end{align*}
\]

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 every attribute of a type into a subtype – not only for enumeration typed attributes. For enumerations, we simply turn each literal into a type. For an infinite type we update the model by the introduction of a new subtype representing a value, whenever an object occurs that has this attribute value for the first time. This is merely a thought experiment; but it radically shows what the purely symbolic viewpoint of modeling is about: the evolving M1/M0-model as a whole is a data store.

### 3.4. 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 the class Breed a power type \[37; 36; 54\]. 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 (38) to (41) become obsolete with solution (iii). At least as long as the only domain states in which we are interested in always have exactly two breeds – corresponding to the number of enumeration type literals in solution (i). Then, we can turn this into an appropriate constraint:
The Reflective Extension of Object Constraint Languages

Breed.allInstances → size ≤ 2 (49)

Now, all we need to do to generalize this situation to an arbitrary number of breeds and to stay robust against inadequate model updates this way is to specify the uniqueness of the identifying breed numbers – compare it to constraint (42):

Breed.allInstances → asSet → size
= Breed.allInstances.breednumber → asSet → size (50)

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: up to the treatment of candidate keys as expressed by constraint (50) solution (iii) is no more than the type object pattern [41] of Johnson and Woolf. Unfortunately, we sometimes might want to model properties that are specific to certain breeds17, which leads us to Sect. 3.5!

3.5. Integrated Subtype and Power Type Externalization

Solution (iv) now shows an equivalent to the full UML power type construction [62; 45] 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 sync. First, we need a slight adoption of constraint (48), which expresses that all M0-objects of a given breed subclass are assigned to the same breed M0-object. The difference is only in the attribute breed vs. the attribute breednumber:

GeneralizationSet.allInstances → select(Breed).generalization.specific → forAll(Breed.allInstances.breed → asSet → size ≤ 1) (51)

Second, we also want to express that objects of different subclasses are assigned to different Breed objects – see how the literal Breed is overloaded by its usage as class name and generalization set name. We can achieve this with the following constraint:

Breed.allInstances → size
≥ GeneralizationSet.allInstances → select(name → includes(Breed)) → generalization.specific → asSet → size (52)

Constraint (52) enforces the existence of a breed description for each breed subclass. Interestingly, according to the constraint there might be M0-objects of class Breed that do not correspond to any of the breed subclasses. And this is OK! It just means that we have modeled not only breeds of dogs but breeds in general, also containing, e.g., breeds of cats. If we actually want to model breeds of dogs only, all we have to do is to change ≥ to = in constraint (52) or to set the "*-multiplicity on the element-of association into 1..*".

4. PRECISE SEMANTICS OF UML POWER TYPES

The current UML superstructure specification contains the following description of the semantics of power types [62]: “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.” First, the

17See also the remark on the disadvantage of implementation complexity in [41].

above statement is not a formal statement. But that is not the point. However, second, and this is very important, this statement is inconsistent against the background of the rest of the UML specification: an M1-level subclass is an instance of the M2-level class `Class` and not an instance of an M1-level classifier. 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 the intended domain. If you model with a power type construct the resulting model is not per se a meta model.

Where does the confusion stem from? One source of misunderstanding of the domain-relation ∈ 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 Appendix D. In D.2 collies is an element of breeds and 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 ∈ 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 reformulation of the above definition. For example, we could state that a power type is a class whose instances represent (intent [8; 9]) sets of domain objects, which are represented (intended) by the instances of subclasses of another classifier. Arbitrary subclasses? An arbitrary classifier? No. All the extra information expressed by the constraints (51) and (52) 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 statement with the Z specification in D.2. Here, in each system state, collies is an element of breeds and a subset of dogs.

Based on the reflective extension of OCL we can give a general semantics for UML power types. For example, we can generalize constraint (41) to all user-defined types:

```ocl
(Class | .allInstances.forAll(sup, sub, power |
   <GeneralizationSet | .allInstances->exists(gs |
      gs.powertype = power
      and gs.generalization.general->includes(sup)
      and gs.generalization.specific->includes(sub)
   )
   implies(
      super.ownedAttribute->select(type = power)->size = 1
      and
      super.ownedAttribute->select(type = power)->forall(p |
      <sub>↑ .allInstances.<p>↑->asSet->size = 1
   )
)
```

Constraint (53) needs to decide upon semantic issues. For example, with (53) the modeler is allowed to give an arbitrary name to a supertype-to-powertype relation. Consequently, (53) requires supertype-to-powertype relations to be unique. Con-
5. A Symbolic Viewpoint of Modeling Languages

5.1. The Classic Database Evolution Viewpoint

Figure 7 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 is an important concept in its own right.

The first viewpoint (i) in Fig. 7 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 constrains 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 different access rights than 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. 7. This data migration step can be very complex, because the existing data must be re-shaped. 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 [17]. Hopefully, the program has been designed for reuse [29] 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.

5.2. 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 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 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, without loss of generality, it fails to fulfill its intended purpose for M1-level model elements other than the ones it has been written for. This is particularly the case for those M1-level elements that are added to the model on behalf of a model update resulting in a new edition of this model. We call a constraint that generalizes an ephemeral constraint to all instances of the necessary M2-level model elements a M1-level model evolution persistent constraint, evolution persistent constraint, or sustainable constraint for short.

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. For example, the constraints needed to characterize class materialization in multilevel modeling frameworks turn out to be evolution persistent constraints.

5.3. The Purely Symbolic Viewpoint
We call viewpoint (ii) also a symbolic viewpoint because it stresses the fact that M1- and M0-level modeling elements can be considered as together intending [8; 9] 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. 7 as a refinement of viewpoint (ii).

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, for a database tool based on UML class diagrams this would mean that we capture and manipulate instances of classes 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 confused with each other: a) instantiations across levels M1 and M0, b) set membership relationships in the expert domain, c) representations of instantiations across levels M1 and M0 at level M1, d) instantiations across levels M2 and M1, e) M1-level model elements resulting from instantiations across levels M2 and M1, f) representations of set memberships in the expert domain at level M0, g) representations of set membership relationships in the expert domain at level M1.

You can perceive this M0-level free modeling (iii) in two ways. Either practical, 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. Note, that in symbolic computation there is also no dedicated grammatical tier for the ground terms. If we assume that all data is kept and maintained at M1-level this greatly eases and unifies the discussion.

Let us analyze UML instance specifications from this perspective. Instance specifications represent M0-level objects. We also have some instance specifications in the M1-model in Fig. 5. The UML considers instance specifications as examples only. There is explicitly no need to give an instance specification for each M0-level instance and an instance specification needs not to provide a slot and value specification for each attribute of the corresponding object. However, if we visualize an M0-object by an instance specification this instance specification should obey to the same rules that we impose as constraints for the M0-level objects. We can do this via appropriate M2-level constraints. We call a constraint an M2-level constraint in this paper only if it contains some M2-level model elements other than types introduced at M1-level\(^{18}\). We show this technique by turning constraint (38) into a M2-level constraint for instance specifications:

\[
\text{InstanceSpecification} \\
\rightarrow \text{select}(\text{classifier} \rightarrow \text{oclIsTypeOf(Dog)}).\text{select}(\text{slot} \rightarrow \text{forAll(}} \\
\text{\quad \text{definingFeature.name} = \text{"breed" \ implies}} \\
\text{\quad \quad \text{definingFeature.value} = \text{Collie}).\text{slot}} \\
\rightarrow \text{select}(\text{definingFeature.name} = \text{breednumber}).\text{value} \\
\rightarrow \text{asSet} \rightarrow \text{size} \leq 1
\]  

As shown by constraint (54) it is, in principle, possible to turn all necessary constraints on M0-level objects into constraints for M1-level instance specifications, although the resulting constraint writing becomes substantially more complex. However, it shows that we could get rid of the M0-level to foster a purely symbolic viewpoint.

\(^{18}\)Strictly speaking, each OCL-constraint is an M2-level constraint, because each OCL constraint is written at least in the context of an M1-type and each M1-type is considered as an extension of the collection of M2-types by the OCL. However, we want to call a constraint an M2-level constraint in this paper only if it contains some M2-level model elements other than M1-types.
Note that we can get rid off the instance specifications as well for this purpose. It is important to understand that the M1- and M0-level together form a language to describe states in the domain. If we compare UML modeling in that sense with symbolic computation, the M0-objects correspond to the ground terms, i.e., terms of ground type. So the existence of instance specifications merely introduces redundancy. Throwing away the M0-level has the disadvantage of complicating constraint writing for ground terms, because currently OCL is designed in terms of the notion of M0-objects. However, throwing away the M0-level has also an important advantage. The instantiation of objects is now always only present as a materialization of instance specifications which makes it much easier to understand this ground term materialization as a special case of M1-model element materialization across all M1-internal levels. We stick with combined M1/M0-modeling for the reason to ease constraint writing, however, with a purely symbolic viewpoint.

6. RELATED WORK

At the MoDELS’2005 conference we have presented a schema and data evolution framework [7] that we have designed and built in the IMIS project [18]. In [22] we have presented a solution for continuous, automatic, mobile data synchronization. Furthermore, the paper [7] also contains an exhaustive description of related work in the area of OO schema evolution.

Meta modeling tools like our AMMI tool [38; 20; 21] or the ATOM^3 tool [69] are the natural candidates for supporting multi-level modeling and clabject modeling. However, they should be improved by pervasive M2/M1/M0-level crossing constraint checking, means for information querying, and direct manipulation of objects as described. With respect to AMMI we are currently working on all of these aspects. The tool MELANIE [5] already offers mature support for all of these aspects and therefore is promising to become a true advocate for a purely symbolic modeling viewpoint. The constraint checking is currently restricted to the clabject rules. METADEPTH [15; 16] is an implementation of a multilevel modeling language on the basis of the ATOM^3 meta modeling tool [69]. It supports clabjects as crucial concept and also checks for adherence to the clabject rules. The article [21] also contains an exhaustive description of related work in the area of modeling and meta modeling tools.

Programming languages and their type systems, in particular, generative programming languages [13], form a mature field of study that is important for the current discussion. In our programming languages GENOUPE [24; 44] and FACTORY [25] it is possible to implement OCL_R constraints [10]. 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 [42]. 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 [30]. 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. 6 into code. This problem is even not overcome by nested inheritance as provide by Jx [50] and J & [51] or advanced nested composition constructs as provided by DEEPFJIG [12].

Nivel [2] is a reductionist multilevel modeling language that supports clabjects, associations, generalization sets, but no power types. In [2] a formal semantics for Nivel is provided by translation to the Weight Constraint Language [67], i.e., stable model semantics. This formalization re-states established descriptions of type systems, e.g., [10], and clabject rules, e.g., [6]. Nivel raises the question whether a multilevel modeling language can be a meta modeling language per se– certainly, it would become

---

19GENOUPE is a C#-extension, whereas FACTORY is a Java-extension.
a meta modeling language if it were used for modeling at M2-level in a meta modeling tool like AMMI [21] or a meta modeling architecture like the MOF (Meta Object Facility) [63]. In [43] it has been clarified that meta levels and linguistic instantiation must not be confused with the levels of a modeling hierarchy and ontological instantiation.

The symbolic viewpoints from Sect. 5.3 superficially resemble but must not be confused with the viewpoint of the important strand of research on models and evolution [66; 14]. 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. They are mere instructive devices that gain their value only from their tension with classical viewpoints on modeling. Current investigations on the relationship of OO conceptual modeling and ontological modeling are very promising [35; 48] and have impact [39] – see [49] 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 [11] is sufficient. In form-oriented analysis [27; 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 current line of argumentation in this article it is not necessary to understand conceptual modeling in terms of ontological modeling [28], i.e., as construction of an ontological commitment as characterized in [32].

We believe that the viewpoint of considering models as evolving data storing systems and vice versa is also particularly appropriate for the emerging paradigm of cloud-based software engineering [47], which eventually demands, in our opinion, for a more holistic approach to the design of (data) services and their utilization [3]. For example, in [4] we have coined the concept of viable software system which is about systems that are pro-actively designed, supported or even implemented 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 deeply grateful to Roland Wagner and Josef Küng for their support and intensive discussions concerning all aspects of theory and practice of information systems.

7. CONCLUSION
We have shown how to extend an object constraint language with reflection. 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 given precise, declarative semantics for OCL\_R. From the viewpoint of the compiler construction community or the formal languages community, reflective constraints are, without loss of generality, the context-sensitive properties of a model, whereas class diagrams or ER diagrams (entity-relationship diagrams) play the role of a context-free grammar. 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. We have elaborated sustainable constraints, i.e., constraints that persist model evolution, for the modeling of sets of sets. We have introduced a distinction between ephemeral constraint writing and evolution persistent constraint writing. 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. In particular, we have discussed the reductionist power of subtype externalization for eliminating all attributes, even infinitely typed attributes. 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 (Par-simonious Data) modeling language in form-oriented analysis [23], 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 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. 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. Sustainable constraint writing adds value. It is a software engineering pattern worth looking at. Sustainable constraint writing makes constraints robust against model updates. There are many use cases for sustainable constraints in different software engineering domains, i.e., both in system design and conceptual modeling. With respect to system design, sustainable 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, sustainable constraint writing is about the externalization of important domain knowledge that is otherwise captured in the ephemeral counterparts of the sustainable constraints.
A. UML META MODEL

Fig. 8 shows a cutout of the UML superstructure specification [62] consisting of all UML meta model elements used in the OCL constraints of Sect. 3.
Fig. 9 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 [56]. The singleton AnyType meta object OclAny serves as most general type for all OCL expressions, i.e., $e :: \text{OCLExpression}$ implies $e : \text{AnyType}$. The singleton TypeType meta object OclType serves as type for all OCL type expressions, i.e., $e :: \text{TypeExpr}$ implies $e : \text{OclType}$. 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 [62] have white color.

**On the Choice of OCL version v2.0**

We have chosen to take the 2006 version OCL v2.0 [56] instead of the current version v2.4 [60] and the ISO standard version v2.3.1 [58; 59] 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 $e :: \text{TypeExp}$ – see Appendix 10. 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:
oclIsTypeOf(type: OclType) : Boolean  (55)
oclIsTypeOf(type: Classifier) : Boolean  (56)

The operation applies to all objects, i.e., objects o : OclAny. It tests whether the object's type equals the type given as parameter. For example, the following constraint evaluates to true:

```ocl
context Person inv: self.oclIsTypeOf(Person)  (57)
```

The OCL v2.0 definition (55) of oclIsTypeOf is correct, whereas the v2.0 definition (56) cannot be ill-typed with respect to its described semantics. Even worse, in accordance with its described semantics, the operation oclIsTypeOf(type : Classifier) cannot be typed at all with the types available 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 type : 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 9 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 [57].
Fig. 10. Meta model of OCL expressions.

Fig. 10 gives a substantial cutout of the OCL abstract syntax as specified in [58]. Basically, it shows, as a single overview, the structure of the OCL syntax kernel as given in Figure 8.2. in [58] plus more elements that are crucial for understanding the syntax and semantics of OCL, in particular, Fig. 10 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 [62] have white color.
D. Z SPECIFICATION OF A POWER TYPES SCENARIO

In this appendix we restate the domain model from Sect. 3 in the specification language set. The aim is twofold. The specification offers a particularly dense presentation of the crucial domain knowledge discussed throughout the article and therefore is amenable to foster the understanding. Second, and even more important, it can foster in separation of concerns - semantics and pragmatics of information system approaches should always be avoided. The specification language Z allows to describe system states on the basis of set theory and first-order 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 [64] with ASMs (abstract state machines) as most recent member [31]. We use only the data facet of Z in this appendix. I recommend [68] as a reference, and also [33; 34; 46].

D.1. Types Specification

We introduce the basic sets of dogs, addresses, intelligence, and aggressiveness. There are kept completely opaque in the following.

\[
\{\text{DOG}\} \quad (58) \\
\{\text{ADDRESS, INTELLIGENCE, AGGRESSIVENESS}\} \quad (59)
\]

We introduce further derived types. The type breed is the set of sets of dogs. The type of collies and pitbull are set equal to the basic type of dogs. Only later, when we specify the schema, we will establish that the sets of collies as well as the set of pitbulls are sub sets of dogs. This is typical set style. The structure of possible system states manifests in variables and axioms declared in the schema. Mathematical notation is first class-citizen in Z. For the modeler this means, that he must often specify concepts that he usually has available as syntactic sugar. Nevertheless, Z specification are usually rather dense than bloated. The advantage is that we can hardly deviate from the declarative, mathematical semantics.

\[
\begin{align*}
\text{BREED} & \equiv \mathbb{P}\text{DOG} \\
\text{COLLIE} & \equiv \text{DOG} \\
\text{PITBULL} & \equiv \text{DOG} \\
\text{GENUS} & \equiv \{\text{Canis, . . .}\}
\end{align*}
\]

D.2. Schema Specification

In D.2 we provide a straightforward specification of the system state. In D.2 we add the attributes - compare this to the domain model provided in Fig. 5. In our Z specification of we model attributes as functions that yield a value for a reference. This means, we explicitly model an object-mechanism that is implicitly given in each object-oriented modeling language.

Each variable in D.2 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.
REFERENCES


