How to Ground Meta-Circular OCL Descriptions – a Set-Theoretic Approach –

Thomas Baar

Universität Karlsruhe, Fakultät für Informatik
Institut für Logik, Komplexität und Deduktionssysteme
Am Fasanengarten 5, D-76128 Karlsruhe
Tel: ++49-721-608 7329
Email: baar@ira.uka.de

Abstract. Meta-modeling is a standard technique to describe the abstract syntax of the Unified Modeling Language (UML) including the Object Constraint Language (OCL). In order to overcome inherent deficiencies of meta-circular UML descriptions, the language MML based on an external formal semantics was proposed as a core of UML. Similarly, this paper proposes CINV as an alternative core of UML. CINV diagrams are defined by a set theoretic definition of syntax and semantics. CINV diagrams are sufficiently expressive to define a syntax which coincides with their own syntax. As a novel feature of our approach, the coincidence can be proven mathematically.

1 Motivation

Although the Unified Modeling Language (UML) is defined as an OMG standard [OMG00], it still lacks an overall agreed formal syntax and semantics description. Recent proposals for UML2.0 [WKC+01,MDC+01,KRS01] suggest different frameworks to describe UML and OCL. This paper combines some ideas of the cited proposals and presents a new approach for the formal definition of UML and OCL: As in [MDC+01,KRS01], we advocate a technique based on meta-modeling. With [WKC+01] we have in common, that our formalism is based on set theory, i.e. an intuitive mathematical notation most people are familiar with.

The proposal in [KRS01] uses a meta-model as a basis for the description of UML2.0. The meta-model captures the abstract syntax of UML in terms of UML class diagrams and well-formedness rules mainly expressed in OCL. The semantics of most language constructs are given by careful natural language statements. The language construct *constraint* is described only broadly; the formalism to express a constraint is left open; OCL is only one of the possibilities (cf. [KRS01], p. 16–71).

Although the definition is concise and elegant, the authors make some assumptions about the knowledge of the reader (cf. [KRS01], p. 5–15):

> *The specification uses a combination of languages - a subset of UML, an object constraint language, and precise natural language to describe*

the abstract syntax and semantics of the full UML. [...] Although this is a metacircular description (as a footnote: In order to understand the description of the UML semantics, you must understand some UML semantics.), understanding this document is practical since only a small subset of UML constructs are needed to describe its semantics.

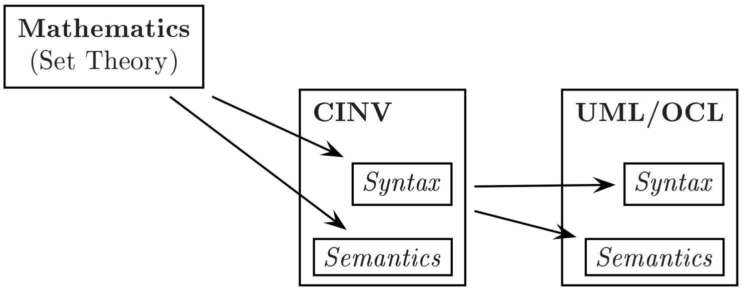
The proposal in [MDC$^+$01] formally defines the syntax and semantics of UML/OCL by an expression of the Meta Modeling Language (MML), i.e. MML serves as a meta-language to describe UML/OCL. The meta-language MML is a static language. MML offers almost the same syntax and constructs as UML and OCL (cf. [BCC$^+$00] for a detailed account).
The formal semantics of MML was published recently in [CEK01] in terms of an object-based calculus (an extension of Abadi/Cardelli [AC98]). Roughly speaking, the given (rather opaque) calculus can be seen as a model of an abstract machine, which is able to evaluate MML expressions. In order to 'understand' the static language MML one has to understand the dynamic evaluation process of the abstract machine.
This kind of semantics definition makes it difficult to reason about properties of MML. Thus, one should not call a MML-expression 'meta-model', i.e. an expression which describes syntax and/or semantics of MML, unless it is proved that the language definition based on the object-based calculus coincides with the meta-model. To our knowledge, such a proof has not been published yet for MML.

The proposal in [WKC$^+$01] avoids the problems inherent to meta-circular definitions. Here, the syntax and semantics of class diagrams are given by mathematical definitions (cf. [WKC$^+$01], Chapter 5.1). The definitions are elegant and reflect the intuition of class diagrams well. But there is a missing formal link to the 'official' UML syntax description which is given by a meta-model. As illustrated in Section 2, two independent syntax definitions can potentially interfere with the semantics definition and can lead to an inconsistent or incomplete semantics definition.

This paper proposes the language **CINV**, which is similar to UML **C**lass diagrams adorned by OCL-like **INV**ariants. Similar to [WKC$^+$01], the syntax and semantics of CINV is based on mathematical definitions. Thus, properties of the language CINV can be proven with mathematical rigor.
CINV aims to be simple and is aligned to established logical languages, as predicate logic. CINV does not support all concepts of UML, e.g. generalisation is excluded. Also some concepts of OCL are abandoned, e.g. undefined expressions, indetermistic terms, `iterate`. CINV is a pure static language. As another significant difference, CINV allows classes to be interpreted by infinite sets, not only by finite sets as in UML.

CINV is designed as a meta-language, i.e. its purpose is the description of object languages in syntax and semantics. Such an object language can be UML/OCL but much simpler languages as e.g. predicate logic are conceivable, too. However, this paper does not aim at the full definition of UML/OCL using

CINV as the description formalism. The main contribution of the paper is rather the architecture for defining specification languages in a meta-circular style.



Since CINV is based on set-theoretic definitions, the drawbacks quoted above from [KRS01] are avoided. One might insist, that the reduction to set theory is not a sufficient solution and postpones the problem only as it is not described how to get formally the semantics of set theory. We have to admit that this argument is true but since the failure of *Hilberts Programm* we know that this is the best we can expect.

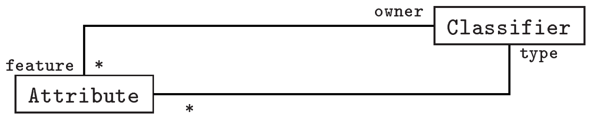
The rest of the paper is organised as follows: Section 2 clarifies the intended meaning of the term 'meta-model' in this paper. Section 3 lists the most important constructs of CINV and discusses their relationship to the constructs of UML/OCL. In Section 4, the syntax and semantics of CINV is given in form of mathematical definitions. Section 5 presents a CINV diagram that really deserves the term 'meta-model' in the sense explained in Section 2. We close with a conclusion and an overview of possible future work.

2 On the Justification of the Term 'Meta-Model'

In the UML world, the term meta-model is mainly used in the sense that UML is able to describe itself. This reflexive usage can be dangerous in the following sense. Suppose you have formally defined the syntax and semantics of a language $\mathcal{L}$ by definitions Def_{syn} and Def_{sem} written in an external formalism.

It is fully justified to describe other languages in terms of the well-defined language $\mathcal{L}$. However, in most cases, $\mathcal{L}$ is used to describe its own syntax by an expression $\mathcal{MM} \in \mathcal{L}$ called 'meta-model'. From a formal point of view, $\mathcal{MM}$ describes only the syntax of a (possibly different) language $\mathcal{L}_1$. So far, $\mathcal{L}_1$ has no semantics. However, if the coincidence of the syntax of $\mathcal{L}$ and $\mathcal{L}_1$ can be proven it is justified to call $\mathcal{MM}$ a *meta-model*.

Within the next example, the language $\mathcal{L}$ is a language of simple class diagrams consisting of classifiers, attributes, and associations. The syntax and semantics of class diagrams is already formally given by mathematical definitions in [WKC$^+$01], Section 5.1. In the following, we will refer to those definitions. Note, that similar definitions are given for CINV in Section 4.



The above $\mathcal{L}$-expression $\mathcal{MM}$ is intended to describe (a part of) the syntax of $\mathcal{L}$. In order to justify the term 'meta-model' we have to map the elements of $\mathcal{MM}$ formally to the elements of the mathematical syntax definition given in [WKC$^+$01].

Following the mathematical semantics definition, a class is interpreted by a set of instances. Thus, the two classes `Classifier` and `Attribute` can be mapped to the sets CLASS and ATT_C given in the mathematical syntax definition. Furthermore, an association connecting classes `A` and `B` and with multiplicity `1` at `B` is interpreted by a set of pairs, which encodes a mandatory and unique correlation of instances of `B` to instances of `A`. Thus, the two associations between `Classifier` and `Attribute` with multiplicity `1` at the association end `owner` and `type` can be mapped to the equivalent constructs in the mathematical syntax definition, i.e. attribute signatures $a : t_C \rightarrow t$ where t_C symbolise the owner-class and t the type-class of an attribute. Before calling $\mathcal{MM}$ a meta-model, such a mapping has to be established also for all other constructs of $\mathcal{L}_1$.

At the first glance, the mapping from $\mathcal{MM}$ to Def_{syn} is obvious and tedious. Nevertheless, it is necessary as the next example illustrates. Suppose, the semantics of $\mathcal{L}$ is slightly changed and an association with multiplicity `1` is interpreted not by an mandatory but optional correlation between the instances of the involved classes. Based on such a changed semantics, $\mathcal{MM}$ would admit to have attributes without an owner-class or a without a type-class. Consequently, the syntax of $\mathcal{L}_1$ would differ from that of $\mathcal{L}$. Even worse, $\mathcal{L}_1$ could not 'inherit' the semantics of $\mathcal{L}$, since the mathematical semantics definition given for $\mathcal{L}$ is based on the syntax of $\mathcal{L}$.

3 Differences between CINV and UML/OCL

The language CINV shares its concrete syntax and most of its language constructs with UML/OCL. The motivation for different constructs lies mainly in different application scenarios for UML/OCL and CINV. UML aims to support a modeller to develop concise UML models and offers a vast number of semantically rich constructs. Most of these constructs are expressible with the help of more basic constructs and therefore do not increase the expressive power of UML. CINV aims to be minimal and simple with respect to the definition of its semantics. Thus, many constructs of UML/OCL have been abandoned in CINV. On the other hand CINV is not just a strict subset of UML/OCL. In order to act as meta-language adequately some new concepts are introduced which perhaps will be adopted by UML/OCL some day.

The language CINV does **not support** the following constructs which are currently supported by the official version of UML/OCL.

Distinction between classes/data types The different treatment of objects (instances of classes) and data items (instances of data types) in UML is motivated by concepts of OO programming. OO programming languages usually assign to objects an identity based on its address on the heap, the identity of data items depends on their value. As we do not pursue an operational semantics (based on a computer model) the address of an object on the heap is not relevant and a value semantics is sufficient.

Generalisation Although generalisation is one of the key concepts of object oriented languages it can be (partly) simulated by 0..1-associations connecting the superclass with its subclasses and additional constraints (for a more detailed description, cf. [GR01]). It is obvious that such a simulation cannot reflect polymorphic objects, but for our purpose this restriction is acceptable. Without generalisation, the type system of CINV is considerably simpler than that of OCL.

N-ary association, association class Both concepts can be simulated using binary associations and OCL constraints (see [GR01] for details).

Attribute Attributes are omitted in CINV. An attribute can be simulated by an association with multiplicity `1`.

Operation Since CINV is a static language, there are no operations that are able to change a state. All operations usable in CINV are queries. Moreover, there is no need in CINV for queries with more than one parameter.

Collection There is no need for bags and sequences in CINV. The only collection type supported in CINV are sets.

Undefinedness of OCL expressions There are basically two reasons for an OCL expression being undefined: (1) Navigation over an optional attribute, and (2) unsuccessful cast to a subtype. Since both attributes and generalisation are not supported by CINV, expressions are never undefined.

Navigation over 1-association In OCL, navigation over an association with multiplicity `1` can be both a term of a collection type and a term of an object type. In CINV, every navigation over an association is a term of a set type.

Iterate There is no need to support `iterate` in CINV. However, CINV offers quantifiers as `forAll`, which in OCL are given by a definition based on `iterate`.

The above list shows constructs of UML/OCL which can be removed without loosing too much of expressive power. In the following list, **new concepts** of CINV are discussed.

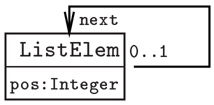
Interpreted Classes The official semantics of UML classes interprets a class by arbitrary sets of objects which only have to satisfy the restrictions imposed by the attached OCL constraints. Sometimes, however, one wishes classes to be interpreted by a fixed set. In the current UML/OCL standard the built-in data types are examples for that situation; e.g. in [OMG00], page 7-33 the built-in data type Integer is defined as: *The OCL type Integer represents the mathematical concept of integer.*

We generalise the idea of builtin types by admitting interpreted classes, which in fact is an established concept of formal logic (cf. e.g. [Har84] on the logical consequences of interpreted symbols).

Infinite Sets The current official semantics of UML restricts the interpretation of classes to finite sets of objects whereas data types as Integer are interpreted by infinite sets. The proposal of a set theoretic semantics in [WKC+01] follows this line.

The restriction to interpret classes by finite sets has its roots in the theory of object oriented programming languages. Here, the language semantics is usually given by the specification of an abstract machine which executes a program. The semantics of a class is given by all possible sets of objects, which could be instantiated in an arbitrary state of the abstract machine. An object can only exist if it was created which takes at least one execution step on the abstract machine. Thus, at any step during the execution only finitely many objects can have been created.

A semantics that allows data types to be infinite but classes to be interpreted only by finite sets has some non-intuitive consequences. Consider the next example:



The class `ListElem` models elements of a list. A list element has a position in the list and a successor. The mandatory successor for every list element has a great impact on the interpretations of the class `ListElem` and allows only three possibilities: 1) `ListElem` is interpreted by an empty set 2) `ListElem` is interpreted by a finite set and the structure induced by the next-function is cyclic 3) `ListElem` is interpreted by an infinite set. If a constraint which ensures the uniqueness of attribute `pos` as, e.g.

context ListElem **inv:**

```
    self.next.pos = self.pos + 1
```

is added to the design, case 2) becomes impossible.

Please note, that the current official semantics of UML/OCL admits only the trivial case 1) which seems to be too restrictive.

The semantics of CINV is defined differently and allows classes to be interpreted by infinite sets. The main reason for that decision is the application scenario of CINV. Suppose, the syntax and semantics of an object language is to be defined using CINV. If the object language is based on terms, there are usually infinitely many of them, e.g. `x, f(x), f(f(x)), ...`. A CINV class `Term` would not be able to model adequately all terms of the object language if the interpretation of CINV classes is restricted to finite sets.

4 Syntax and Semantics of CINV

In Section 3 we argued informally which concepts should be supported by CINV and which can be left out. In the following, the syntax and semantics of CINV is given more formally in terms of a mathematical notation. To understand the definitions, the reader is assumed to be familiar with ordinary first-order logic notation ($\forall$ (forall), $\exists$ (exists), $\exists!$ (exists one), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies), $\leftrightarrow$ (is equivalent to)) and basic mathematical notations (sets, functions, total functions, cardinality operator # for sets, cartesian product and p_i as functions to extract the i-th component of a tuple, $\textsc{Boolean}$ as set of boolean values $true$ and $false$, etc.). Due to space restrictions, we cannot formally define all constructs of CINV in this paper but concentrate on the most important aspects. The formal description capturing all constructs of CINV will be published elsewhere.

4.1 Syntax

Definition 1 defines the basic part of the abstract syntax of CINV. Following the tradition in logic, this part is called *Signature of CINV*.

Definition 1 (Signature of CINV)
A signature of CINV *is a tuple*

$$\mathcal{M} = \begin{array}{l}\langle \textsc{Classifier}, \textsc{Assoc}, \textsc{Query}, isBool, isNat,\\ associates, multiplicities, querysig\rangle\end{array}$$

satisfying the following properties :

1. $\textsc{Classifier}, \textsc{Assoc}, \textsc{Query}$ *are pairwise disjoint symbol-sets for* classes, *(*binary) associations, *and* (unary) queries. *We use the (sorted) variables* c, as, q *to denote elements of these sets.*
2. *The set* $\textsc{Classifier}$ *is assumed to comprise always a few elements, which reflects the concept of predefined types in OCL. Currently, we assume to have predefined classes representing the natural numbers and the Booleans. We call the predefined classes* interpreted classes *since their semantics is defined in Definition 3 by a fixed interpretation.*
   $isBool : \textsc{Classifier} \rightarrow \textsc{Boolean}$
   $isNat : \textsc{Classifier} \rightarrow \textsc{Boolean}$
   $\exists! c_1 c_2\ isBool(c_1) = true \wedge isNat(c_2) = true \wedge c_1 \neq c_2$
3. $associates$ *represents the types of association ends. It is a function which assigns to each element of* $\textsc{Assoc}$ *a pair over* $\textsc{Classifier}$*.*
   $associates : \textsc{Assoc} \rightarrow \textsc{Classifier} \times \textsc{Classifier}$
4. $multiplicities$ *represents the multiplicities of association ends. It is a function which assigns to each element of* $\textsc{Assoc}$ *a pair over* $\mathcal{P}(\mathbb{N})$*.*
   $multiplicities : \textsc{Assoc} \rightarrow \mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N})$
   Note, that – in difference to UML – the empty set is accepted as the multiplicity of an association end. Semantically, this means that the number of interpreting links of the association is not restricted, in particular infinitely many links are possible (cmp. Definition 3).
5. $querysig$ *represents the signature of queries, i.e. their owner class, parameter class and return class.* $querysig$ *is a function which assigns to each element of* $\textsc{Query}$ *a triple over* $\textsc{Classifier}$*.*
   $querysig : \textsc{Query} \rightarrow \textsc{Classifier} \times \textsc{Classifier} \times \textsc{Classifier}$

Based on the signature, all other syntactical constructs, e.g. the set of variables D_VAR or formulas D_FML, etc., are definable. That means, once the signature is chosen, the extensions of these sets are fixed. To indicate this, in our mathematical formalisation we use such names, which start with an *D_* or *d_*, what represents *derived*.

We do not present the formal definitions of all syntactical constructs. Only few examples are given. All other constructs can be defined in a similar way. Note, that our kind of language definition is very flexible. Once one wishes to integrate a new syntactical construct, e.g. my_fancy_terms, he or she can simply extend CINV by a set D_MYFANCYTERM.

Most of the constructs given below are assigned a type. This is formalised by a function $typeOf$.

- The set of set-types D_SETTYPE is a '1-1 copy' of CLASSIFIER.
  $\text{D\_SETTYPE} = \{\texttt{Set(}c\texttt{)} \mid c \in \text{CLASSIFIER}\}$
- The set of association ends D_ASSOEND is the union of D_AE1 and D_AE2 which in turn are 1-1 copies of ASSOC. We use the (sorted) variable ae to denote elements of D_ASSOEND.
  $\text{D\_ASSOEND} = \text{D\_AE1} \cup \text{D\_AE2}$
  $\text{D\_AE1} = \{as\texttt{.ae1} \mid as \in \text{ASSOC}\}$ $\text{D\_AE2} = \{as\texttt{.ae2} \mid as \in \text{ASSOC}\}$
  $typeOf(as\texttt{.ae1}) = p_1(associates(as))$
  $typeOf(as\texttt{.ae2}) = p_2(associates(as))$
- The set of variables D_VAR is defined as the cartesian product of CLASSIFIER and $\mathbb{N}$. Thus, there exists infinite many variables of each classifier. We use the (sorted) variable v to denote elements of D_VAR.
  The type of a variable $v \in$ D_VAR is the corresponding classifier.
  $\text{D\_VAR} = \text{CLASSIFIER} \times \mathbb{N}$
  $\forall v \forall c\; typeOf(v) = c \leftrightarrow (\exists i\; v = \langle c, i\rangle)$
- The set of `allInstances`-terms is defined as a 1-1 copy of CLASSIFIER. Their type is an element of D_SETTYPE.
  $\text{D\_ALLINSTANCESTERM} = \{c\texttt{.allInstances} \mid c \in \text{CLASSIFIER}\}$
  $typeOf(c\texttt{.allInstances}) = \texttt{Set(}c\texttt{)}$
- CINV distinguishes between formulas D_FML, object terms D_OBJTERM, and set terms D_SETTERM, where formulas can be seen as a special kind of object terms.
  The set D_SETTERM comprises all terms of type $\texttt{Set(}c\texttt{)}$. Thus, D_SETTERM is the union of `allInstances`-terms, association terms, select terms, etc. Analogously, the set of object terms D_OBJTERM is the union of variables, query terms, size terms, etc., and the set of formulas is the union of equation formulas, `and`-formulas, `not`-formulas, and `forAll`-formulas.
  $\text{D\_FML} = \text{D\_EQUALFML} \cup \text{D\_ANDFML} \cup \text{D\_NOTFML} \cup \text{D\_FORALLFML}$
  $\text{D\_OBJTERM} = \text{D\_VAR} \cup \text{D\_QUERYTERM} \cup \text{D\_SIZETERM} \cup \ldots$
  $\text{D\_SETTERM} = \text{D\_ALLINSTANCESTERM} \cup \text{D\_ASSOTERM} \cup \ldots$
  We demonstrate the formal definition of the remaining terms and formulas only at few examples, the rest can be defined analogously.

Since the following definitions for association terms, equation formulas, etc., are based on D_ObjTerm, D_SetTerm, D_Fml they are in fact mathematical inductive definitions.

– The set of association terms D_AssoTerm is a subset of the cartesian product over D_ObjTerm and D_AssoEnd, which obeys the normal type restrictions.
  $\text{D\_AssoTerm} = \{ote\texttt{.}ae \mid ote \in \text{D\_ObjTerm} \wedge$
  $\quad \exists as \exists ae' \; \{ae, ae'\} = \{as\texttt{.ae1}, as\texttt{.ae2}\} \wedge typeOf(ote) = typeOf(ae')\}$
  $typeOf(ote\texttt{.}ae) = typeOf(ae)$
– The set of equation formulas D_EqualFml is a subset of the cartesian product over D_ObjTerm∪D_SetTerm. In difference to OCL, an equation formula is syntactical correct only iff the involved terms have the same type. As all formulas, the type of equation formulas is the predefined classifier for Boolean.
  $\text{D\_EqualFml} = \{te_1\texttt{=}\; te_2 \mid te_1 \in \text{D\_ObjTerm} \cup \text{D\_SetTerm} \wedge$
  $\quad te_2 \in \text{D\_ObjTerm} \cup \text{D\_SetTerm} \wedge typeOf(te_1) = typeOf(te_2)\}$
  $\forall f \forall c \; f \in \text{D\_EqualFml} \rightarrow (typeOf(f) = c \leftrightarrow isBool(c))$
– The set of `forAll`-formulas D_ForAllFml is a subset of the cartesian product over D_SetTerm, D_Var, and D_Fml.
  $\text{D\_ForAllFml} = \{ste\texttt{->forAll}(v \mid f), \mid ste \in \text{D\_SetTerm} \wedge f \in \text{D\_Fml} \wedge$
  $\quad \exists c \; typeOf(ste) = \texttt{Set(}c\texttt{)} \wedge typeOf(v) = c\}$
  $\forall f \forall c \; f \in \text{D\_ForAllFml} \rightarrow (typeOf(f) = c \leftrightarrow isBool(c))$

Finally, the term *CINV diagram* can be defined in a formal way.

Definition 2 (CINV Diagram)
A CINV diagram *is a tuple* $\langle \mathcal{M}, \mathcal{I} \rangle$ *where*

– $\mathcal{M}$ *is a signature*
– $\mathcal{I}$ *is a set of formulas of signature* $\mathcal{M}$ ($\mathcal{I} \subseteq$ D_Fml)

The elements of $\mathcal{I}$ *are called* textual constraints *or – since CINV is a static language –* invariants.

4.2 Semantics

The semantics of CINV is based on a mathematical structure called *Interpretation*. An interpretation is similar to object diagrams, which interprets UML class diagrams.

Definition 3 (Interpretation of a signature)
For a given CINV signature

$$\mathcal{M} = \begin{array}{l}\langle \text{Classifier}, \text{Assoc}, \text{Query}, isBool, isNat, \\ associates, multiplicities, querysig\rangle\end{array}$$

the tuple
$\langle \text{Object}, objclass, \text{Link}, \text{QueryFunc}\rangle$
is called interpretation *iff the following properties are satisfied:*

1. Object *is a set whose elements are called* objects. *We use the (sorted) variable o to denote elements of* Object.
   It is assumed, that Object *contains at least all natural numbers and furthermore two elements which we denote by the booleans tr, fa.*
   $\mathbb{N} \subset \text{Object} \wedge \exists o_1 o_2\ o_1 \notin \mathbb{N} \wedge o_2 \notin \mathbb{N} \wedge o_1 \neq o_2 \wedge o_1 = tr \wedge o_2 = fa$
2. *The function $objclass$ denotes the type of each object and thus defines the interpretation of each classifier. The predefined classifiers representing natural numbers and booleans are interpreted suitably.*
   $objclass : \text{Object} \rightarrow \text{Classifier}$
   $\forall o\ (isNat(objclass(o)) \leftrightarrow o \in \mathbb{N}) \wedge (isBool(objclass(o)) \leftrightarrow o = tr \vee o = fa)$
3. Link *is a subset of the cartesian product over* Assoc, Object, Object.
   Objects which are connected by a link must have the same type as given by $associates$ for the corresponding association. The number of links which interprets an association correspondences with its multiplicity. Note, that for associations with empty multiplicity no restriction about the number of links must be satisfied.
   $\text{Link} \subseteq \text{Assoc} \times \text{Object} \times \text{Object}$
   $\forall as \forall o_1 o_2\ \langle as, o_1, o_2 \rangle \in \text{Link} \rightarrow objclass(o_1) = p_1(associates(as)) \wedge$
   $\quad objclass(o_2) = p_2(associates(as))$
   $\forall as \forall o_1\ objclass(o_1) = p_1(associates(as)) \wedge p_2(multiplicities(as)) \neq \emptyset \rightarrow$
   $\quad \#\{o_2 \mid \langle as, o_1, o_2 \rangle \in \text{Link}\} \in p_2(multiplicities(as))$
   $\forall as \forall o_2\ objclass(o_2) = p_2(associates(as)) \wedge p_1(multiplicities(as)) \neq \emptyset \rightarrow$
   $\quad \#\{o_1 \mid \langle as, o_1, o_2 \rangle \in \text{Link}\} \in p_1(multiplicities(as))$
4. QueryFunc *interprets queries by appropriate functions on* Object, *which obey the type conditions given by $querysig$ and which are total with respect to the assigned types.*
   $\text{QueryFunc} : \text{Query} \times \text{Object} \times \text{Object} \times \text{Object}$
   $\forall q \forall o_1 o_2 o_3\ \langle q, o_1, o_2, o_3 \rangle \in \text{QueryFunc} \rightarrow$
   $\quad \langle objclass(o_1), objclass(o_2), objclass(o_3) \rangle = querysig(q)$
   $\forall q \forall o_1 o_2\ objclass(o_1) = p_1(querysig(q)) \wedge objclass(o_2) = p_2(querysig(q)) \rightarrow$
   $\quad \exists! o_3\ \langle q, o_1, o_2, o_3 \rangle \in \text{QueryFunc}$

Note, that as for the elements of a signature the elements of its interpretation can be almost freely chosen.

Contrary to this, the semantics of the derived syntax constructs such as terms and formulas are given by a fixed definition based on fixed evaluation functions:

$$d\_eval_o : \text{D\_ObjTerm} \times \text{D\_VarAssign} \rightarrow \text{Object}$$
$$d\_eval_s : \text{D\_SetTerm} \times \text{D\_VarAssign} \rightarrow \text{D\_SetObj}$$
$$d\_eval_f : \text{D\_Fml} \times \text{D\_VarAssign} \rightarrow \{tr, fa\}$$

where D_VarAssign is the set of all assignments of variables to objects of the same type and D_SetObj is the powerset of Object.
$\text{D\_VarAssign} = \{va \mid va \subseteq \text{D\_Var} \times \text{Object} \wedge$
$\quad \forall v \exists! o\ \langle v, o \rangle \in va \wedge objclass(o) = typeOf(v)\}$
$\text{D\_SetObj} = \{so \mid so \subseteq \text{Object}\}$
Based on this, the semantics of terms and formulas is defined formally, e.g.:

– The evaluation of a variable is directly given by the variable assignment.
  $\forall v \forall va\ d\_eval_o(v, va) = o \leftrightarrow \langle v, o\rangle \in va$
– The evaluation of `allInstances`-terms is the set of all objects of the involved classifier.
  $\forall c \forall va\ d\_eval_s(c.\texttt{allInstances}, va) = \{o \mid objclass(o) = c\}$
– The semantics of `forAll`-formulas is reduced to that of the quantifier $\forall$.
  $\forall f \forall v \forall ste \forall va\ d\_eval_f(ste\texttt{->forAll}(v \mid f), va) = tr \leftrightarrow$
  $\forall o \forall va'\ o \in d\_eval_s(ste, va) \wedge newassign(v, o, va, va') \rightarrow d\_eval_o(f, va') = tr$
  where $newassign(v, o, va, va')$ encodes that va' is an 'updated' variable assignment. Formally, $newassign(v, o, va, va')$ is an abbreviation for
  $\exists va'' \forall o'\ va'' = va/\{\langle v, o'\rangle\} \wedge va' = va'' \cup \{\langle v, o\rangle\}$.

5 Meta-Model of CINV Syntax

Hitherto, we have described the syntax and semantics of CINV by mathematical definitions. Now, we want to proceed with the meta-modeling approach and define the syntax of CINV by a special CINV diagram $\mathcal{MM}$ which is called 'meta-model'. As discussed in Section 2, the term 'meta-model' should be justified by a formal proof showing that the semantics of $\mathcal{MM}$, i.e. the set of all interpretations which evaluate all textual constraints of $\mathcal{MM}$ to tr, really coincides with the syntax of CINV. That proof will be published elsewhere.

An overview of the meta-model $\mathcal{MM}$ is depicted in Appendix A. For the sake of clarity, some additional syntactical elements as lines, arrows, and stereotypes are exploited. No semantical meaning is attached to those elements; they just work as a kind of comment to improve readability.

The overview of $\mathcal{MM}$ does not contain all the detailed information of $\mathcal{MM}$ e.g. all textual constraints are suppressed. A detailed meta-model containing all constraints is specified in Subsection 5.3.

5.1 Concrete Syntax

As in UML/OCL, CINV diagrams consist of a mixture of diagrammatic and textual notation. So far, we have not discussed a concrete syntax for CINV diagrams. The definition of the concrete syntax is not within the scope of the paper. We will use the standard UML mapping from abstract to concrete syntax, e.g. classes are represented by rectangles, textual constraints are given within the usual textual notation, etc. The only exception is a class which is interpreted. Interpreted classes are represented by shadowed rectangles whereas normal classes use unshadowed rectangles.

Furthermore, some macros can be used when formulating terms and formulas. For instance, CINV does not contain the construct `isEmpty` which is used in OCL to test a set to be empty. Nevertheless, `isEmpty` is usable in CINV as well. It comes into the language as a macro for a bigger `size`-term, e.g. `employers->isEmpty` is expanded to `employers->size = 0`.
There are much more of such macros, mainly for formulas which are expanded

to formulas merely containing `not`, `and`, and `forAll`, e.g. the formula `ste->exists(v| f(v))` is expanded to `not(ste->forAll(v| not(f(v))))`.

In addition to such common abbreviations, there is one abbreviation not used in the OCL, yet. In CINV the expression `ste!prop` is possible, where `ste` is a set term of type `Set(X)` and `prop` is a property applicable to terms of type `X`. The expansion of `ste!prop` depends on the context `r(ste!prop, t1, ...,tn)` which is assumed to be the smallest enclosing expression of type `Boolean`. The expansion works on the enclosing expression

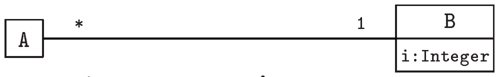
```
r(ste!prop, t1, ...,tn)
```

and transforms this to

```
ste->forAll(v| r(v.prop, t1,...,tn))
```

where `v` is a new variable.

Lets consider a little UML/OCL example which illuminates the motivation for this abbreviation:



The OCL constraint **context** `A` **inv:** `self.b.i > 0` cannot be expressed in CINV in the same way since the subterm `self.b` is an association term which always is of type `Set(X)` and `i` can only be applied on terms of type `B` but not on those of type `Set(B)`.

However, the above `ste!prop`-macro can be used in CINV to express shortly an equivalent constraint: **context** `A` **inv:** `self.b!i > 0`

The smallest enclosing term `r(...)` of `self.b!i` is `self.b!i > 0`. Thus, the expanded form is **context** `A` **inv:** `self.b->forAll(v| v.i > 0)`

Note, that multiplicity `1` for the association end on `B` allows the set `self.b` to have only a single element. Thus, the expanded CINV constraint has the same meaning as the OCL constraint given above.

The `ste!prop`-macro is mainly applied in association terms which use association ends with multiplicity `1`. In that case, it coincides with the dot-notation of OCL for association navigation. However, the `ste!prop`-macro is more flexible and also useful in other situations.

5.2 Template Instantiation

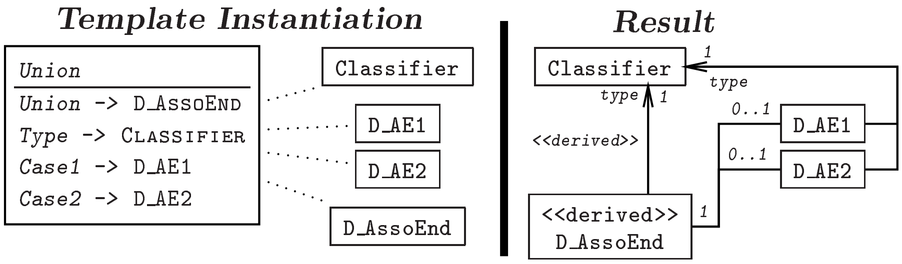
The detailed meta-model is complex in the sense that many constraints are attached to it. On the other hand, there are many regularities; e.g. the classes D_Fml, D_ObjTerm, D_SetTerm are defined based on the same technique.

We will define successively parts of the meta-model. Each part is defined directly, or by instantiation of one or more of the templates listed in Appendix B. A template is a CINV diagram with some textual constraints attached to it. A detailed description of that technique and details about the instantiation are given in [BHSS00] (where the templates are called patterns and textual constraints are called constraint schemata). A comparable but more sophisticated technique is strongly exploited in [MDC+01] to define a meta-model for UML.

Please note, that unlike in [MDC+01] our template mechanism is not part of the language of CINV diagrams itself but can be seen as a 'meta-notation'.

The instantiation of a template is given by a rectangle with the template name in the first compartment and a mapping from the template vocabulary to the vocabulary of the meta-model $\mathcal{MM}$ in the second compartment. Moreover, the resulting classes, connected by a dotted line with the template, are shown as well.

Example 1
The instantiation of the template Union (see App. B.2) on the left hand side is an abbreviation for the design of the right hand side together with the constraints shown below.



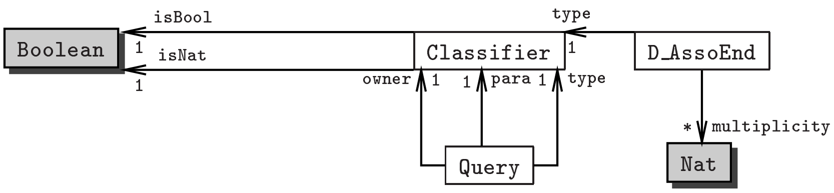
Additional Constraints in Result:

```
D_AssoEnd.allInstances->forAll(u1, u2| u1.d_AE1= u2.d_AE1 and
   u1.d_AE2= u2.d_AE2 implies u1 = u2)
context D_AssoEnd inv:
    self.d_AE1->union(self.d_AE2)->size = 1
context D_AssoEnd inv:
    (not(self.d_AE1->isEmpty) implies
    self.type = self.d_AE1!type) and
    (not(self.d_AE2->isEmpty) implies
    self.type = self.d_AE2!type)
```

5.3 Precise Definition of Meta-Model $\mathcal{MM}$

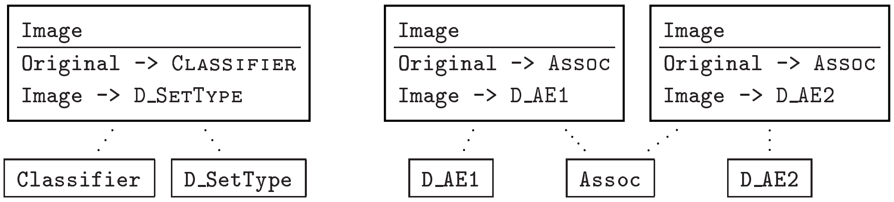
As in the case of the mathematical syntax definition we concentrate of selected constructs when defining $\mathcal{MM}$.

Basic meta-classes: Basic meta-classes reflect the signature of CINV. They can be seen as a counterpart of those constructs which are given in Definition 1.



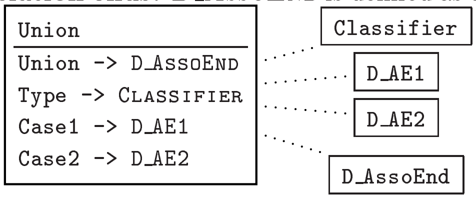
Additional Constraints:
None.

Set types, association ends: D_SetType, D_AE1, D_AE2 are simply 1-1 copies of Classifier, Assoc.



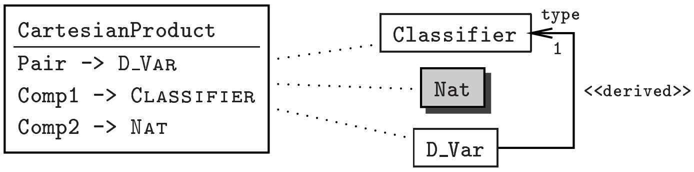
Additional Constraints:
None.

Association ends: D_AssoEnd is defined as the union of D_AE1 and D_AE2.



Additional Constraints:
None.

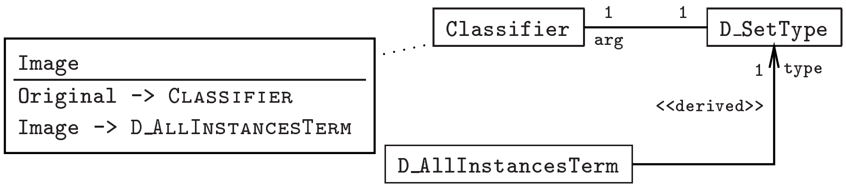
Variables: D_Var is the cartesian product of Classifier and Nat.



Additional Constraints:

1. The type of D_Var is derived.
   context D_Var **inv:**
   `self.type = self.arg1`

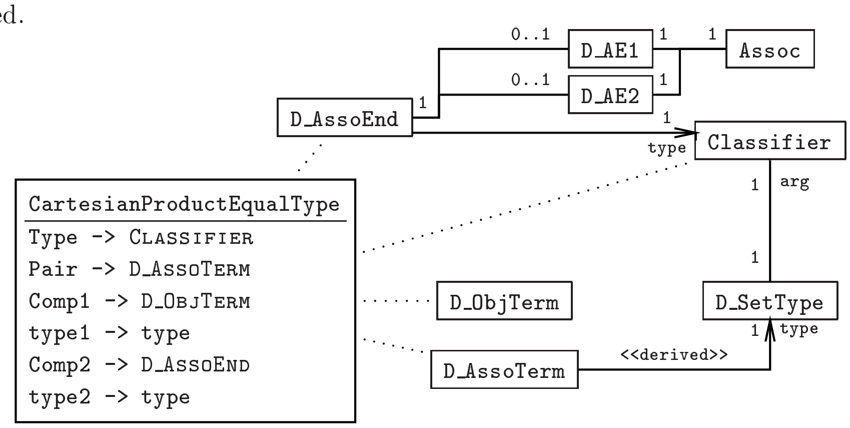
allInstances-terms: D_AllInstancesTerm is a 1-1 copy from Classifier. The type of elements of D_AllInstancesTerm is derived.



Additional Constraints:

1. The type of D_AllInstancesTerm is derived.
   context D_AllInstancesTerm **inv:**
   `self.type!arg = self.arg`

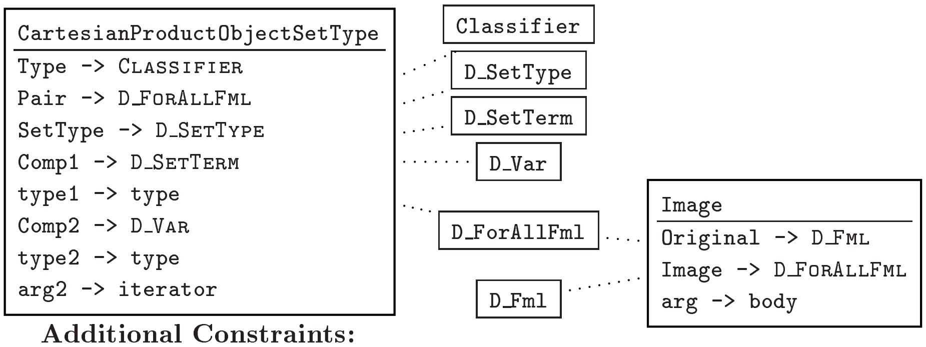
Association terms: D_AssoTerm is the cartesian product of object terms and association ends obeying additional typing rules. The type of D_AssoTerm is derived.



Additional Constraints:

1. The type of D_AssoTerm is derived from the type of the opposite association end.
   context D_AssoTerm **inv:**
   ```
   not(self.arg2!d_AE1->isEmpty) implies
   self.type!arg = arg2!d_AE1!asso!d_AE2!type
   ```
   context D_AssoTerm **inv:**
   ```
   not(self.arg2!d_AE2->isEmpty) implies
   self.type!arg = arg2!d_AE2!asso!d_AE1!type
   ```

`forAll`-formulas: D_ForAllFml is derived by combination of D_SetTerm, D_Var, and D_Fml. Within a `forAll`-formula the incorporated variable is called *iterator*, the incorporated formula is called *body*.



Additional Constraints:
None.

6 Conclusion

We have specified the static language CINV which can serve as a meta language when defining UML/OCL by a meta-model. Thus, CINV is similar to MML [BCC+00].

CINV is based on set theory and formally defined by mathematical definitions. As a consequence, mathematical methods can be applied to investigate properties of CINV. One important property of CINV is the coincidence of mathematically given syntax and the semantics of meta-model $\mathcal{MM}$ given in Section 5. The proof of that coincidence requires the interpretation of classes of CINV by infinite object sets.

The long-term goals of CINV are to get a better understanding of meta-modeling, and the separation of fundamental and derived concepts of UML. CINV tries to be minimal and has abandoned a lot of concepts which are accepted in UML (cf. Section 3). CINV has proven to be sufficiently expressive to describe its own syntax. In a next step, it will be applied to define the syntax and semantics of UML/OCL in a meta-modeling style. Besides, CINV will be used to describe other languages, especially Dynamic Logic [Har84].

Dynamic Logic deserves special interest since it can be used to express the semantics of real programming languages [Bec01]. Dynamic Logic is the logical foundation of the KeY-system [ABB+00], which is able to verify the correctness of a JAVA implementation with respect to a UML/OCL specification. Currently, the KeY-system relies on a semantics of UML/OCL given by a translation of UML models into formulas of Dynamic Logic [BBS01]. We expect, that an alternative CINV description of the same translation would lead to new insights.

Acknowledgements

My thanks are due to Reiner Hähnle and Bernhard Beckert for the careful reading of a draft of this paper.

References

[ABB$^+$00] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I. P. de Guzman, G. Brewka, and L. M. Pereira, editors, *Proceedings, Logics in Artificial Intelligence (JELIA), Malaga, Spain*, LNCS 1919. Springer, 2000.

[AC98] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1998.

[BBS01] Thomas Baar, Bernhard Beckert, and Peter H. Schmitt. An extension of Dynamic Logic for modelling OCL's @*pre* operator. In *Proceedings, Fourth Andrei Ershov International Conference, Perspectives of System Informatics, Novosibirsk, Russia*, volume 2244 of *LNCS*, pages 47–54. Springer, July 2001.

[BCC$^+$00] Steve Brodsky, Tony Clark, Steve Cook, Andy Evans, and Stuart Kent. Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach, September 2000. Available at: http://www.cs.york.ac.uk/puml/mmf/index.html.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[BHSS00] Thomas Baar, Reiner Hähnle, Theo Sattler, and Peter H. Schmitt. Entwurfsmustergesteuerte Erzeugung von OCL-Constraints. In K. Mehlhorn and G. Snelting, editors, *Informatik 2000, 30. Jahrestagung der Gesellschaft für Infomatik*, pages 389–404. Springer, September 2000.

[CEK01] Tony Clark, Andy Evans, and Stuart Kent. The metamodelling language calculus: Foundation semantics for UML. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering. 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6. 2001 Proceedings*, volume 2029 of *LNCS*, pages 17–31. Springer, 2001.

[GR01] Martin Gogolla and Mark Richters. Expressing UML Class Diagrams Properties with OCL. In Tony Clark and Jos Warmer, editors, *Advances in Object Modelling with the OCL*, pages 86–115. Springer, Berlin, LNCS 2263, 2001.

[Har84] David Harel. Dynamic Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*. Reidel, 1984.

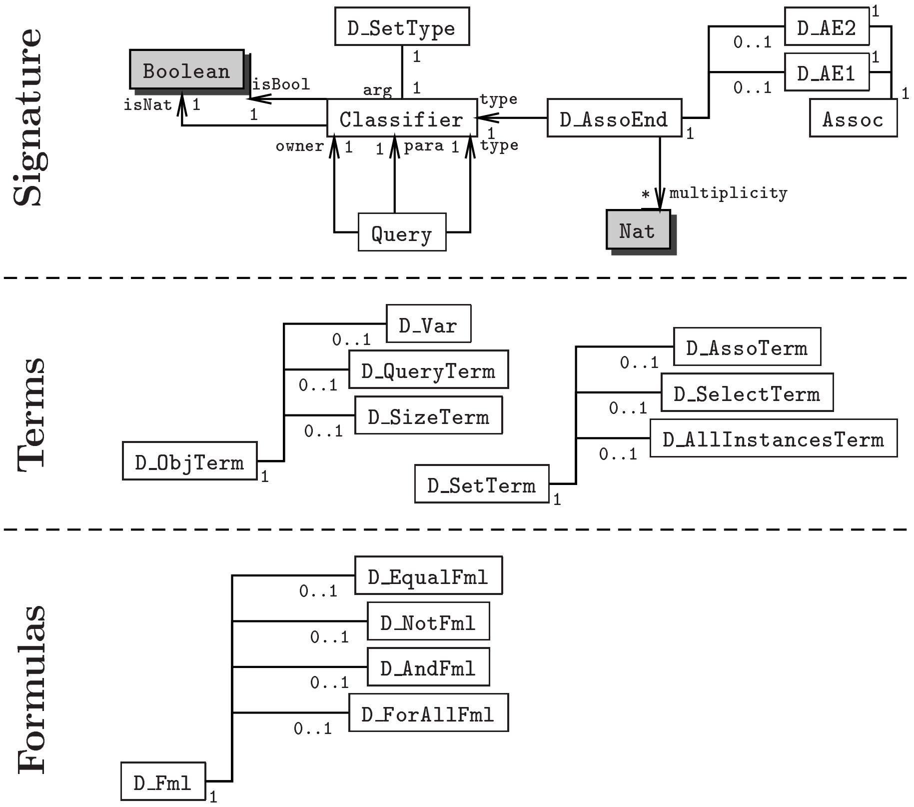
[KRS01] Cris Kobryn, Guus Ramackers, and Bran Selic. Infrastucture of the Unified Modeling Language 2.0 Specification version 0.6 (Response to UML2.0 RfP). Technical report, Submission to the OMG, 2001.

[MDC$^+$01] Stephen J. Mellor, Desmond D'Souza, Tony Clark, Andy Evans, and Stuart Kent. Infrastucture and Superstructure of the Unified Modeling Language 2.0 (Response to UML2.0 RfP). Technical report, Submission to the OMG, 2001.

[OMG00] OMG. OMG Unified Modeling Language Specification. Technical Report OMG-UML V1.3, Object Mangagement Group, March 2000.

[WKC$^+$01] Jos Warmer, Anneke Kleppe, Tony Clark, Anders Ivner, Jonas Högström, Martin Gogolla, Mark Richters, Heinrich Hussmann, Steffen Zschaler, Simon Johnston, David S. Frankel, and Conrad Bock. Object Constraint Language 2.0 (Response to UML2.0 RfP). Technical report, Submission to the OMG, 2001.

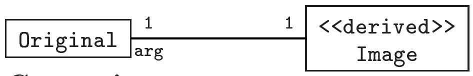
A Overview of the Meta-Model $\mathcal{MM}$



B A Template Catalogue

B.1 Image

The class `Image` is defined as a 1-1 copy of the class `Original`. Please note, that the 1-1 multiplicity on the association between `Image` and `Original` implies that the interpreting links form a bijection between the set of objects interpreting `Image` and those interpreting `Original`.

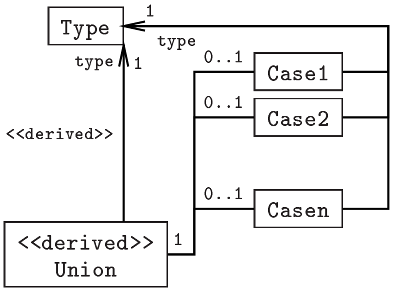


Additional Constraints:
None.

B.2 Union

The template Union defines a complete union of classes. We define a schematic template for n classes. The constraints are in fact schemata of OCL constraints.

When applying the template one first has to adapt the parameter n and to generate from OCL schemata the corresponding OCL constraints and in a second step to proceed as for normal templates.

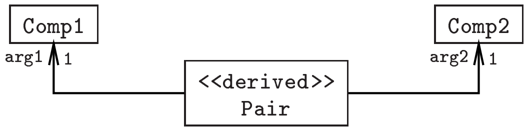


Additional Constraints:

1. Elements of `Union` are unique with respect to their components.
   `Union.allInstances->forAll(u1, u2| u1.case1 = u2.case1 and ...and u1.casen = u2.casen implies u1 = u2)`
2. Each element of `Union` has a link to exactly one element of a `Case` class.
   context Union **inv:**
   `self.case1->union(self.case2->union( ...->union(self.casen)..)->size = 1`
3. The association between `Union` and `Type` is derived.
   For $i = 1..n$ we have the conjunction of invariants:
   context Union **inv:**
   `not(self.casei->isEmpty) implies self.type = self.casei!type`

B.3 CartesianProduct

The class `Pair` represents the set of all pairs of `Comp1` and `Comp2`.

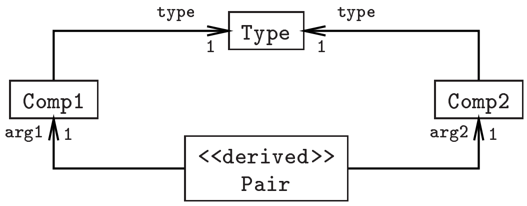


Additional Constraints:

1. Elements of `Pair` are unique with respect to their components.
   `Pair.allInstances->forAll(p1, p2 | p1.arg1 = p2.arg1 and p1.arg2 = p2.arg2 implies p1 = p2)`
2. `Pair` is maximal.
   context Comp1 **inv:**
   `Comp2.allInstances->forAll(c2 | Pair.allInstances->exists(p | p.arg1! = self and p.arg2! = c2))`

B.4 CartesianProductEqualType

The class `Pair` represents the set of all pairs one can create from objects of `Comp1`, `Comp2` with the same type.

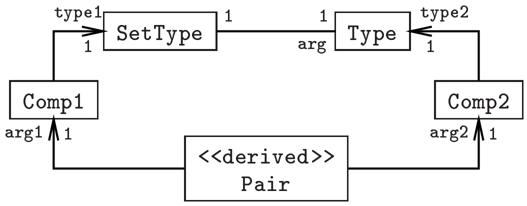


Additional Constraints:

1. Elements of `Pair` are unique with respect to their components.
2. The components of a pair have the same type.
   context Pair **inv:**
   ```
   self.arg1!type = self.arg2!type
   ```
3. `Pair` is maximal.
   context Comp1 **inv:**
   ```
   Comp2.allInstances->forAll(c2 | c2.type = self.type
   implies Pair.allInstances->exists(p |
   p.arg1! = self and p.arg2! = c2))
   ```

B.5 CartesianProductObjectSetType

The class `Pair` represents the set of all pairs one can create from objects of `Comp1`, `Comp2` with compatible types.



Additional Constraints:

1. Elements of `Pair` are unique with respect to their components.
2. The components of a pair have compatible types.
   context Pair **inv:**
   ```
   self.arg1!type1!arg = self.arg2!type2
   ```
3. `Pair` is maximal.
   context Comp1 **inv:**
   ```
   Comp2.allInstances->forAll(c2 |
   c2.type2 = self.type1!arg
   implies Pair.allInstances->exists(p|
   p.arg1! = self and p.arg2! = c2))
   ```