Metamodels without Metacircularities

Thomas Baar

Universität Karlsruhe, Fakultät für Informatik
Institut für Logik, Komplexität und Deduktionssysteme
Am Fasanengarten 5, D-76128 Karlsruhe

baar@ira.uka.de

ABSTRACT. *Although modeling languages as the UML (Unified Modeling Language) are primarily used in the context of software development, other application scenarios exist. A prominent example is the definition of languages; e.g. UML is partly defined in terms of UML. This technique of metamodeling became popular and has many advantages, but metacircularity is one of the most serious disadvantages. This paper proposes an approach that allows the formulation of metamodels but prevents metacircularity at the same time. The core of our approach is the UML-like formalism CINV that is formally defined in a set-theoretical way and serves as a metalanguage to define UML in syntax and semantics. In this paper we concentrate on the application of CINV to describe the abstract syntax of a language. In particular, we will compare our CINV descriptions with similar metamodels formulated in UML. That comparison reveals some principal shortcomings of UML when used as a metalanguage.*

RÉSUMÉ. *Bien que les langages de modélisation comme l'UML (Unified Modeling Language) soient utilisés en premier lieu pour le développement de software, il y a des autres possibilités d'application. Un exemple très connu est la définition des langages, par exemple l'UML est défini partiellement par l'UML. Cette méthode de méta-modélisation est devenue populaire parce qu'elle offre beaucoup d'avantages ; mais sa méta-circularité reste un de ses plus gros inconvénients. Cette étude propose un procédé qui permet de formuler des méta-modèles tout en excluant la méta-circularité. Le fond de cette étude constitue le formalisme CINV qui est semblable à l'UML qui se définit formellement sur la théorie des ensembles et qui sert de méta-langage pour définir la syntaxe et la sémantique de l'UML. Dans cette étude, nous nous concentrons sur l'application de CINV pour la description de la syntaxe abstraite d'un langage. En particulier, nous comparons nos descriptions de CINV avec des méta-modèles semblables formulés in l'UML. Cette comparaison montre plusieurs désavantages de l'UML quand il est utilisé en tant que méta-langage.*

KEYWORDS: *Metamodel, syntax description, language semantics, OO specification techniques, UML, OCL, MML, CINV*

MOTS-CLÉS : *méta-modèle, description de la syntaxe, sémantique de langage, OO spécification techniques, UML, OCL, MML, CINV*

1re soumission à *L'objet*, le 20th December 2002.

1. Motivation

Modeling languages as the UML (Unified Modeling Language) are primarily designed to facilitate the formulation of abstract views on software systems. However, there are other applications, too. A prominent example can be found in the UML's language definition [OMG 00], where a UML model describes the abstract syntax of UML. It became popular to call that model *metamodel of the UML* since it describes properties of the language UML by the help of UML itself. A known drawback of the UML metamodel is *metacircularity*, i.e. in order to understand the metamodel one has to understand UML's language definition first. The circularity is due to the fact that the metamodel is part of UML's language definition. The problem of metacircularity was already identified but is almost ignored, as a quote from [OMG 00, page 2-8] reveals:

> *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.*

In this paper, we propose an approach that allows the formulation of metamodels but prevents metacircularity at the same time. The core of our approach is a UML-like formalism called **CINV** (**C**LASSDIAGRAMS + **INV**ARIANTS) that is formally defined in a mathematical, set-theoretical way. The language CINV is designed to serve as a metalanguage and allows the formal definition of other languages. For instance, in [BAA 02] we give a language definition of UML including its *Object Constraint Language (OCL)* in terms of CINV which captures both syntax and semantics. In this paper we concentrate on the application of CINV to describe the abstract syntax of a language. In particular, we will compare our CINV descriptions with similar metamodels formulated in UML. That comparison reveals some principal shortcomings of UML when used as a metalanguage.

The language CINV is much inspired by the *Meta Modeling Language (MML)* which was first proposed in [BRO 00]. The language MML was developed to describe precisely other languages in syntax and semantics. MML offers almost the same constructs as UML including OCL and the syntactical appearance of MML-expressions is very similar to that of UML-expressions, too. Due to its application domain, MML has abandoned UML's dynamic constructs, e.g. operations and pre-/post-conditions. Please keep in mind, that a static language is sufficient to describe the syntax and semantics of a language, even when the described language offers dynamic constructs. The language CINV is a static language as well.

Although the languages MML and CINV have a lot in common, their semantical foundations are completely different. The formal semantics of MML was published in [CLA 01] in terms of an object-based calculus. The chosen semantics of MML allows a direct and concise description of complicated language constructs with an

’execution oriented semantics’. An example is the construct `iterate` of OCL which can be defined in terms of MML very smoothly just by reduction to a corresponding construct of MML (see [MEL 01, page 125 f] for details). On the other hand, the calculus-based semantics of MML makes it very difficult to analyse the meaning of a MML-expression in a simple, mathematical way. Therefore, we have chosen for the language CINV a different semantical foundation based on set theory (comp. Section 4). The chosen approach is similar to the model-theoretic semantics of traditional logical languages as for instance first-order predicate logic and was also applied by Richters [RIC 01] in order to define the semantics of UML in a formal manner. Compared to the semantical foundation given by Richters for UML, our semantical description of CINV is rather small since CINV is designed as a simple and small language.

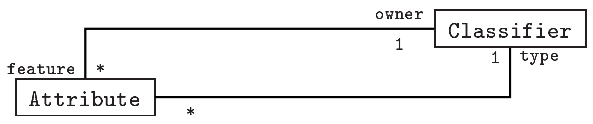
CINV does not support all concepts of UML, e.g. generalisation is excluded. Also some concepts of OCL are abandoned, e.g. undefined expressions, indetermistic terms, `iterate`. As another significant difference, CINV allows classes to be interpreted by infinite sets, not only by finite sets as in UML.

The rest of the paper is organised as follows: Section 2 clarifies the intended meaning of the term ’metamodel’ in this paper. Section 3 lists the most important constructs of CINV and discusses their relationship to the constructs of UML. In Section 4, the syntax and semantics of CINV is given in form of mathematical definitions, Section 5 discusses some useful syntactic sugar. Section 6 presents a CINV diagram that really deserves the term ’metamodel’ in the sense explained in Section 2. Finally, we draw a conclusion and give an overview of possible future work.

2. On the Justification of the Term ’Metamodel’

The language definition for UML uses the term ’metamodel’ very frequently and inconsiderately. It is claimed that the metamodel defines the abstract syntax of UML. At the same time, the semantics of UML is given only informally.

The informal style used to explain the semantics of UML blurs the difficulties imposed by a syntax definition in form of a metamodel. However, metacircularity becomes evident once one tries to formalise the semantics of UML constructs. For the purpose of illustration we pick a part of the UML metamodel and try to formalise the meaning of classes and attributes.



The depicted class diagram is a simplified version of the original metamodel given in [OMG 00]. By the usage of the simplified version we ignore for the rest of the Section some significant issues of UML, e.g. generalization relationship among classes, multiplicities of attributes, etc. The simplifications are due to the sake of brevity and clarity. The relevant insights about the nature of metamodels and metacircularity can be gained from the simplified version as well.

Our simplified metamodel declares the two language constructs classes and attributes represented by the (meta)classes `Classifier` and `Attribute`. Their semantics could be described in an informal style as follows:

> The semantics of a class is given by a finite set of objects. The semantics of an attribute is given by a set of slots, where each slot is attached to exactly one object of the *owner class* of the attribute and vice versa. Furthermore, each slot has a value which is an object of the same class as the *type class* of the attribute.

Note, that this informal semantics definition presupposes implicitly some facts about the relationship between classes and attributes. In particular, there are two classes attached to an attribute, one class is called *owner class* and the other class is called *type class*. However, from a formal point of view we do not know anything about the relationship between classes and attributes until we have understood the metamodel which captures the abstract syntax. Here, the relationship between classes and attributes is stipulated by the meaning of the two associations between `Classifier` and `Attribute`. If one tries to define the semantics of associations and multiplicities, he or she has to make assumptions similar to the assumptions made in the semantics definition for attributes. Thus, metacircularity is unavoidable and makes a pure metamodeling approach questionable.

Fortunately, the syntax and semantics of UML can be defined differently and can easily be based on mathematical definitions what avoids all the problems mentioned above. For our simplified version of UML, the syntax can be given by the tuple:

$$[\textsc{Class}, \textsc{Attribute}, \mathit{attOwner}, \mathit{attType}]$$

where $\textsc{Class}$, $\textsc{Attribute}$ are finite sets and $\mathit{attOwner}$, $\mathit{attType}$ are total functions from $\textsc{Attribute}$ into $\textsc{Class}$ ($\mathit{attOwner}, \mathit{attType} : \textsc{Attribute} \rightarrow \textsc{Class}$). The tuple just reflects mathematically the intuition that a UML-diagram consists of a set of classes and a set of attributes and that exactly two classes are attached to each attribute, one of them could be called *owner class*, the other one *type class*.
The semantics of classes and attributes can be mathematically given by the tuple

$$[\textsc{Object}, \mathit{objClass}, \mathit{attSlot}]$$

where $\textsc{Object}$ is a finite set, $\mathit{objClass}$ is a total function from $\textsc{Object}$ into $\textsc{Class}$ ($\mathit{objClass} : \textsc{Object} \rightarrow \textsc{Class}$), i.e. each element of $\textsc{Object}$ is uniquely assigned to an element of $\textsc{Class}$. The component $\mathit{attSlot}$ represents the semantics of attributes. Mathematically, $\mathit{attSlot}$ is a total function from $\textsc{Attribute}$ into the space of unary partial functions over $\textsc{Object}$. Furthermore, $\mathit{attSlot}$ may assign to an attribute att only such partial functions, which satisfy some restrictions depending on argument att. These restrictions are best written down using the mathematical notation:

$$attSlot : \textsc{Attribute} \rightarrow (\textsc{Object} \nrightarrow \textsc{Object})$$

$$Dom(attSlot(att)) = \{o \mid objClass(o) = attOwner(att)\}$$

$$Rng(attSlot(att)) \subseteq \{o \mid objClass(o) = attType(att)\}$$

where att, o denote elements from OBJECT, ATTRIBUTE and Dom, Rng denote the domain and the range of a function, respectively.

To summerise, a formal semantics definition of UML is prevented by metacircularity as long as the syntax of UML is purely given by a metamodel. Once the syntax is given by an additional mathematical definition, the formal semantics can be defined based on that syntax in a set-theoretic way. In that case, the metamodel is rendered superfluous since the whole language definition is based on mathematical definitions.

In the next Sections, we illustrate the sketched approach by the mathematical definition of the UML-like language CINV. Besides, a CINV-diagram describing the syntax of CINV is given. As argued above, the label 'metamodel' needs to be justified. The justification can be given by a mathematical proof of coincidence of what the labeled expression describes and of what the mathematical definition has prescribed to be the syntax of CINV.

3. Differences between CINV and UML

The language CINV shares its concrete syntax and most of its language constructs with UML including OCL. The motivation for different constructs lies mainly in different application scenarios for UML 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 have been abandoned in CINV. On the other hand CINV is not just a strict subset of UML. In order to act as meta-language adequately some new concepts are introduced which perhaps will be adopted by UML some day.

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

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.

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. [GOG 01]). It is obvious that such a simulation cannot reflect polymorphic objects, but for our purpose this restriction is acceptable. Compared with OCL, the omission of the generalisation concept yields to a considerably simpler type system in CINV.

N-ary association, association class Both concepts can be simulated using binary associations and invariant constraints (see [GOG 01] 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`.

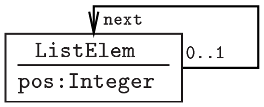
The above list shows constructs of UML which can be removed without loosing too much of expressive power. In the following list, **new concepts** of CINV are discussed.

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 standard the built-in data types are examples for that situation; e.g. in [OMG 00], 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. [HAR 84] 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 semantics of UML given in [WAR 01] follows this line.

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 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. The object language is usually based on terms, i.e. there are infinitely many of them, e.g. `x, f(x), f(f(x)), ...` A class `Term` in our language CINV is able to model adequately all terms of the object language, since the class can be interpreted by infinitely many objects. This is a serious advantage compared to UML where the interpretation of classes is restricted to finite sets of objects. Consequently, UML is not appropriate to model term-based languages as UML including OCL which comprises infinitely many expressions.

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, powerset operator $\mathcal{P}$, cartesian product and p_i as functions to extract the i-th component of a tuple, $\mathbb{N}$ as the set of natural numbers, BOOLEAN as the set of boolean values $true$ and $false$, etc.). Due to space restrictions, it is not desirable to define the complete language CINV in this paper. Thus, we present only for few language constructs the full definition and concentrate more on the motivation and the explanation of the chosen style of language definition. Furthermore, there are some powerful constructs in CINV which are not needed until CINV is not used to describe the semantics of another language. These constructs are not taken into consideration here since we concentrate on syntax definition as the application domain in this paper. A complete definition of CINV can be found in [BAA 02].

4.1. *Syntax*

For the language CINV, we distinguish two forms of syntax. Below, the first form is formally defined. CINV is introduced as a traditional term-based language, i.e. after choosing a signature the sets of terms and formulas are given by the usual definitions based on structural induction. The second form discussed in Section 5 makes it easier to handle CINV expressions. While the first form is purely character based, the second form is graphical oriented and similar to the appearance of UML expressions. As in UML, especially in OCL, there are numerous ways to use short forms of certain CINV-expressions.

Definition 1 defines the basic part 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 representing* classes*,* (binary) associations*, and* (unary) queries.

2) The set CLASSIFIER *is assumed to comprise in each signature two predefined elements to represent natural numbers and the Booleans. We call the predefined elements* interpreted classes *since their semantics is defined in Definition 3 by a fixed interpretation.*

$isBool, isNat : \textsc{Classifier} \rightarrow \textsc{Boolean}$

$\exists! c_1 c_2\ isBool(c_1) = true \wedge isNat(c_2) = true \wedge c_1 \neq c_2$

3) As in UML, there are additional information attached to associations and queries; in particular the type and multiplicity of association ends and the signature of queries. These information are formally captured by the following functions:

$$\begin{array}{ll} \mathit{associates} & : \textsc{Assoc} \rightarrow \textsc{Classifier} \times \textsc{Classifier} \\ \mathit{multiplicities} & : \textsc{Assoc} \rightarrow \mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N}) \\ \mathit{querysig} & : \textsc{Query} \rightarrow \textsc{Classifier} \times \textsc{Classifier} \times \textsc{Classifier} \end{array}$$

Based on the signature, the sets of all other syntactical items, e.g. the set of variables D_VAR or formulas D_FML, etc., are fixed. That means, once the signature is chosen, the extensions of these sets are fixed. To improve the readability we denote such sets only by names beginning with an $D\_$ or $d\_$, what stands for *derived*.

In order to define the sets of syntactical items we exploit basically two known techniques: set comprehension and structural induction. Set comprehension is sometimes sufficient as for the definition of plain set-type symbols, the set of association ends, the set of variables, and the set of `allInstances`-terms. Note, that all syntactical items are sets of symbols and concatenation of symbols is done by writing one after the other. Terminals are displayed in `true type fonts`. We shall moreover assume, that most of the used terminals are reserved symbols.

$$\begin{array}{l} \text{D\_}\textsc{SetType} = \{\texttt{Set(}c\texttt{)} \mid c \in \textsc{Classifier}\} \\ \text{D\_}\textsc{AssoEnd} = \text{D\_AE1} \cup \text{D\_AE2} \\ \text{D\_AE1} = \{as\texttt{.ae1} \mid as \in \textsc{Assoc}\} \;\; \text{D\_AE2} = \{as\texttt{.ae2} \mid as \in \textsc{Assoc}\} \\ \text{D\_}\textsc{Var} = \{\texttt{<}c\texttt{,}i\texttt{>} \mid c \in \textsc{Classifier} \wedge i \in \mathbb{N}\} \\ \text{D\_}\textsc{AllInstancesTerm} = \{c\texttt{.allInstances} \mid c \in \textsc{Classifier}\} \end{array}$$

The function typeOf assignes to each CINV-term a type symbol, i.e. an element of CLASSIFIER or D_SETTYPE. Depending on the assigned type, terms in CINV are separated into object terms and set terms. The sets of object terms, set terms, and formulas are defined as unions of more specialised sets. Each of these sets represents a single language construct of CINV, e.g. D_EQUALFML the equation formulas, D_QUERYTERM the application of a query on an object term, D_ASSOTERM the navigation over association. All these sets and the function typeOf are defined simultaneously. We only give few examples.

$$\begin{array}{l} \text{D\_}\textsc{Fml} = \text{D\_}\textsc{EqualFml} \cup \text{D\_}\textsc{AndFml} \cup \text{D\_}\textsc{ForAllFml} \cup \ldots \\ \text{D\_}\textsc{ObjTerm} = \text{D\_}\textsc{Var} \cup \text{D\_}\textsc{QueryTerm} \cup \text{D\_}\textsc{SizeTerm} \cup \ldots \\ \text{D\_}\textsc{SetTerm} = \text{D\_}\textsc{AllInstancesTerm} \cup \text{D\_}\textsc{AssoTerm} \cup \ldots \\ \text{D\_}\textsc{QueryTerm} = \{ote\texttt{.}q\texttt{(}ote1\texttt{)} \mid \{ote, ote1\} \subseteq \text{D\_}\textsc{ObjTerm} \wedge \\ \qquad \mathit{typeOf}(ote) = p_1(\mathit{querysig}(q)) \wedge \mathit{typeOf}(ote1) = p_2(\mathit{querysig}(q))\} \\ \mathit{typeOf}(ote\texttt{.}q\texttt{(}ote1\texttt{)}) = p_3(\mathit{querysig}(q)) \\ \text{D\_}\textsc{AssoTerm} = \{ote\texttt{.}ae \mid ote \in \text{D\_}\textsc{ObjTerm} \wedge \\ \qquad \exists as \exists ae' \; \{ae, ae'\} = \{as\texttt{.ae1}, as\texttt{.ae2}\} \wedge \mathit{typeOf}(ote) = \mathit{aeType}(ae')\} \\ \mathit{typeOf}(ote\texttt{.}ae) = \texttt{Set(}\mathit{aeType}(ae)\texttt{)} \end{array}$$

For the definition of D_ASSOTERM we used the auxiliary function $aeType$ defined on D_ASSOEND as:

$aeType(as.\texttt{ae1}) = p_1(associates(as)) \qquad aeType(as.\texttt{ae2}) = p_2(associates(as))$
Note, that the terms from D_ASSOTERM always have a type of form `Set(`X`)`. As already emphasised, this is different to the type of navigation terms in OCL.

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 definition for 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

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

the tuple

$\langle \textsc{Object}, objClass, \textsc{Link}, \textsc{QueryFunc} \rangle$

is called interpretation *iff the following properties are satisfied:*

1) OBJECT *is a infinite set whose elements are called* objects*. It is assumed, that* OBJECT *contains at least all natural numbers and furthermore two elements which we denote by the boolean constants* tr*,* fa*. The variable* o *(possibly decorated with subscripts) denotes always an element of* OBJECT*.*

$\mathbb{N} \subset \textsc{Object} \wedge \exists o_1 o_2\ o_1 \notin \mathbb{N} \wedge o_2 \notin \mathbb{N} \wedge o_1 \neq o_2 \wedge o_1 = tr \wedge o_2 = fa$

2) The function $objClass$ *denotes the type of each object and thus defines the interpretation of each classifier. Differently to UML, the interpretation of classifiers is not restricted to finite object sets. The predefined classifiers representing natural numbers and booleans are interpreted suitably.*

$objClass : \textsc{Object} \rightarrow \textsc{Classifier}$
$\forall o\ (isNat(objClass(o)) \leftrightarrow o \in \mathbb{N}) \wedge (isBool(objClass(o)) \leftrightarrow o = tr \vee o = fa)$

3) LINK *serves as the interpretation of associations and is formally defined as 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 corresponds with its multiplicity.*

$\text{LINK} \subseteq \text{ASSOC} \times \text{OBJECT} \times \text{OBJECT}$
$\forall as \forall o_1 o_2\ \langle as, o_1, o_2\rangle \in \text{LINK} \rightarrow objClass(o_1) = p_1(associates(as)) \wedge$
$\quad objClass(o_2) = p_2(associates(as))$
$\forall as \forall o_1\ objClass(o_1) = p_1(associates(as)) \wedge p_2(multiplicities(as)) \neq \emptyset \rightarrow$
$\quad \#\{o_2 \mid \langle as, o_1, o_2\rangle \in \text{LINK}\} \in p_2(multiplicities(as))$
$\forall as \forall o_2\ objClass(o_2) = p_2(associates(as)) \wedge p_1(multiplicities(as)) \neq \emptyset \rightarrow$
$\quad \#\{o_1 \mid \langle as, o_1, o_2\rangle \in \text{LINK}\} \in p_1(multiplicities(as))$

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.*

$\text{QUERYFUNC} \subseteq \text{QUERY} \times \text{OBJECT} \times \text{OBJECT} \times \text{OBJECT}$
$\forall q \forall o_1 o_2 o_3\ \langle q, o_1, o_2, o_3\rangle \in \text{QUERYFUNC} \rightarrow$
$\quad \langle objClass(o_1), objClass(o_2), objClass(o_3)\rangle = querysig(q)$
$\forall q \forall o_1 o_2\ objClass(o_1) = p_1(querysig(q)) \wedge objClass(o_2) = p_2(querysig(q)) \rightarrow$
$\quad \exists! o_3\ \langle q, o_1, o_2, o_3\rangle \in \text{QUERYFUNC}$

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:

$$\begin{aligned} d\_eval_o &: \text{D\_OBJTERM} \times \text{D\_VARASSIGN} \rightarrow \text{OBJECT} \\ d\_eval_s &: \text{D\_SETTERM} \times \text{D\_VARASSIGN} \rightarrow \text{D\_SETOBJ} \\ d\_eval_f &: \text{D\_FML} \times \text{D\_VARASSIGN} \rightarrow \{tr, fa\} \end{aligned}$$

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.

$\text{D\_VARASSIGN} = \{va \mid va \subseteq \text{D\_VAR} \times \text{OBJECT} \wedge$
$\qquad \forall v \exists! o\ \langle v, o\rangle \in va \wedge objClass(o) = typeOf(v)\}$
$\text{D\_SETOBJ} = \{so \mid so \subseteq \text{OBJECT}\}$

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. Improving Readability of CINV-Diagrams

The syntax definition given in the last Section defines CINV as a purely character-based language. This kind of syntax definition serves well as a basis to define the formal semantics. However, CINV-diagrams given in this form are tedious to read and hardly to understand.

Thus, a second form of syntax is needed for CINV to gain acceptance. The second form is a mixture of graphical and textual elements, similar to class diagrams in UML adorned by OCL-constraints. The elements of a signature $\mathcal{M}$ are mapped in the standard way to graphical elements, i.e. classes are represented by rectangles, associations by lines between rectangles according to the function $associates$, multiplicities are adorned to the lines according to the function $multiplicities$, etc. As in UML, one can also add role names to association ends and use them instead the symbols from D_ASSOEND. Other rules known from OCL to simplify the appearance of an expression can be applied as well; e.g. the form **context** `C` **inv:** `invbody(self)` can be used as another notation for textual constraints of form
`C.allInstances->forAll(self| invbody(self))`.

5.1. *Macros*

There is a plenty of macros defined in CINV. 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

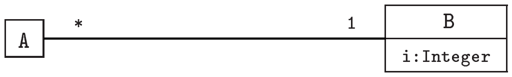
```
r(ste!prop, t1, ..., tn)
```

and transforms this to

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

where `v` is a new variable.

Lets consider a little UML 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.

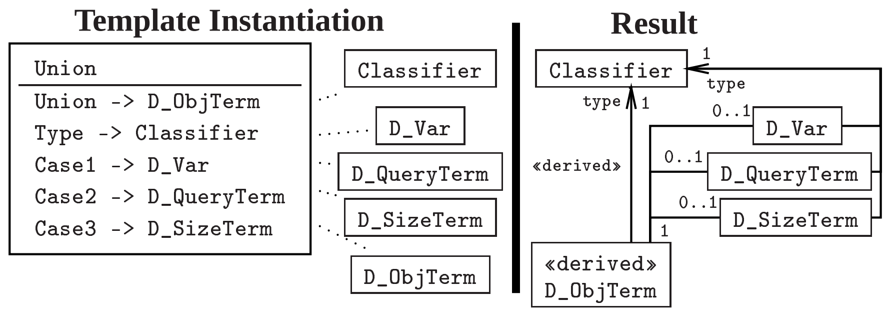
The `ste!prop`-macro is mainly applied in association terms which use association ends with multiplicity `1`. In that case, it coincides with the dot-notation of OCL for association navigation. However, the `ste!prop`-macro is more flexible and also useful in other situations.

5.2. *Template Instantiation*

Another powerful technique to simplify the appearance of CINV-diagrams is template instantiation. A template is a CINV diagram with some textual constraints attached to it. The templates used in this paper are listed in Appendix A, in [BAA 02] one can find a more complete collection. A detailed description of that technique and details about the instantiation were first given in [BAA 00] (where the templates are called patterns and textual constraints are called constraint schemata). A comparable but more sophisticated technique is strongly exploited in [MEL 01] to define a metamodel for UML. Please note, that unlike in [MEL 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 target vocabulary 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. A.2) on the left hand side is an abbreviation for the CINV-diagram of the right hand side together with the constraints shown below.



Additional Constraints in Result:

context `D_ObjTerm` **inv:**
```
self.d_Var->size + self.d_QueryTerm->size
+ self.d_SizeTerm->size=1
```

context `D_ObjTerm` **inv:**
```
(not(self.d_Var->isEmpty) implies
self.type = self.d_Var!type) and
(not(self.d_QueryTerm->isEmpty) implies
self.type = self.d_QueryTerm!type) and
(not(self.d_SizeTerm->isEmpty) implies
self.type = self.d_SizeTerm!type)
```

6. Metamodel of CINV Syntax

Hitherto, we have described the syntax and semantics of CINV by mathematical definitions. Now, we investigate the ability of CINV to describe its own syntax and develop a metamodel $\mathcal{MM}$. As discussed in Section 2, the term 'metamodel' should be justified by a formal proof showing that the semantics of $\mathcal{MM}$, i.e. the interpretations in which all textual constraints of $\mathcal{MM}$ are evaluated to tr, really coincides with the set of CINV-diagrams defined by the mathematical definition. The proof that $\mathcal{MM}$ really deserves the label 'metamodel' is technical and can be found in [BAA 02].

An overview of the metamodel $\mathcal{MM}$ is depicted in Figure 1. The overview of $\mathcal{MM}$ suppress all textual constraints and some of the associations between classes. The two horizontal dashed lines indicate the three layers of $\mathcal{MM}$. The layer 'Signature' basically mimics the mathematical tuple $\mathcal{M}$ given in Definition 1. The classes `Classifier`, `Assoc`, `Query` in the metamodel represent the corresponding components of $\mathcal{M}$. The component $querysig$ of $\mathcal{M}$ is coded in the metamodel by the three

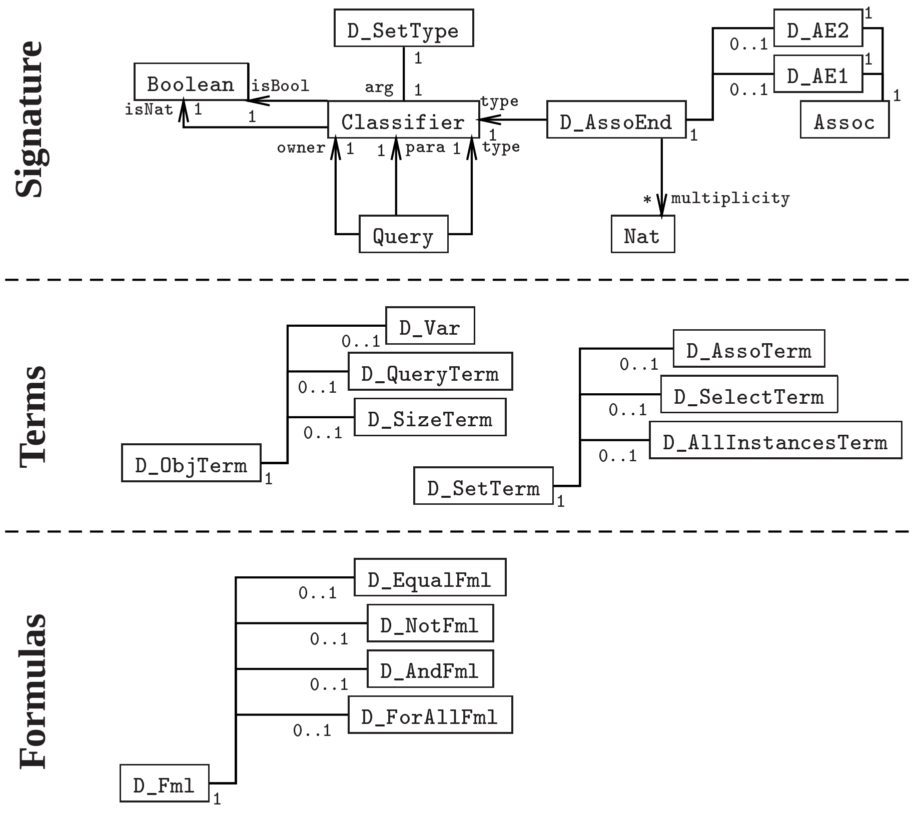


Figure 1. *Overview of the Metamodel* $\mathcal{MM}$

associations between Query and Classifier. Besides, there are classes D_SetType, D_AssoEnd, D_AE1, D_AE2 in order to represent set-type symbols and association ends. The simple set-comprehension definition of D_SETTYPE for example, is simulated in the metamodel by the association between Classifier and D_SetType with multiplicities 1-1. The components $associates$, $multiplicities$ of $\mathcal{M}$ are reflected in the metamodel implicitly by the associations between the classes D_AssoEnd and Classifier, Nat.

The classes in the layers 'Terms', 'Formulas' are mainly defined by template instantiation, since this is the easiest way to mimic the mathematical syntax definition. Again, we can only give some few examples.

allInstances-**terms** The objects of class D_AllInstancesTerm are isomorphic copies of objects of class Classifier. In fact, the template Image is also used to define the classes D_SetType, D_AE1, D_AE2.

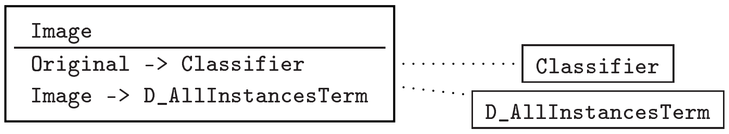


Figure 2. *Definition of `allInstances`-terms*

Additional Constraints:

None.

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

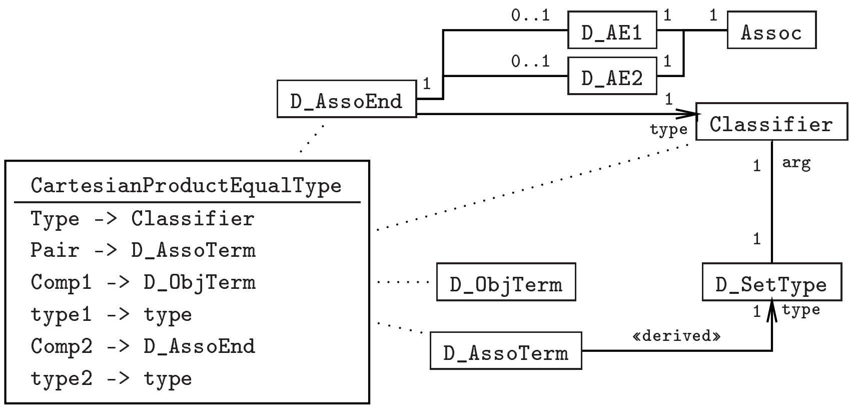


Figure 3. *Definition of association terms*

Additional Constraints:

1) The type of objects 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
```

7. Conclusion

We have specified the static language CINV. Its intended application domain is the definition of other languages in syntax and semantics. Thus, CINV serves as a meta-language and can be compared partly with UML and with MML [BRO 00].

CINV is based on set theory and formally defined by mathematical definitions. As a consequence, mathematical methods can be applied to investigate the meaning of CINV-diagrams. One important property of CINV is the coincidence of mathematically given syntax and the semantics of metamodel $\mathcal{MM}$ given in Section 6. The proof of that coincidence requires the interpretation of classes of CINV by infinite object sets.

Differently from UML, the metamodel $\mathcal{MM}$ serves not as the primary syntax definition of CINV, but offers the same clear description of the syntax. This approach avoids CINV to be defined by metacircular definitions.

The long-term goals of CINV are to get a better understanding of metamodeling, 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 the syntax and semantics of other languages, in particular UML including OCL. The syntax definition for UML is very similar to that given here for CINV. The semantics of UML was described in [BAA 02] on top of a semantics description for Dynamic Logic [HAR 84] in terms of CINV.

Dynamic Logic deserves special interest since it can be used to express the semantics of real programming languages [BEC 01]. Dynamic Logic is the logical foundation of the KeY-system [AHR 00], which is able to verify the correctness of a JAVA implementation with respect to a UML specification. Currently, the KeY-system relies on a semantics of UML given by a translation of UML models into formulas of Dynamic Logic [BAA 01]. 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.

8. References

[AHR 00] AHRENDT W., BAAR T., BECKERT B., GIESE M., HABERMALZ E., HÄHNLE R., MENZEL W., SCHMITT P. H., "The KeY Approach: Integrating Object Oriented Design and Formal Verification", OJEDA-ACIEGO M., DE GUZMAN I. P., BREWKA G., PEREIRA L. M., Eds., *Proceedings, Logics in Artificial Intelligence (JELIA), Malaga, Spain*, LNCS 1919, Springer, 2000.

[BAA 00] BAAR T., HÄHNLE R., SATTLER T., SCHMITT P. H., "Entwurfsmustergesteuerte Erzeugung von OCL-Constraints", MEHLHORN K., SNELTING G., Eds., *Informatik 2000, 30. Jahrestagung der Gesellschaft für Infomatik*, Springer, Sep. 2000, p. 389–404.

[BAA 01] BAAR T., BECKERT B., SCHMITT P. H., "An Extension of Dynamic Logic for Modelling OCL's $@pre$ Operator", *Proceedings, Fourth Andrei Ershov International Conference, Perspectives of System Informatics, Novosibirsk, Russia*, vol. 2244 of *LNCS*, Springer, Jul. 2001, p. 47–54.

[BAA 02] BAAR T., "Über die Semantikbeschreibung OCL-artiger Sprachen", PhD thesis, Fakultät für Informatik, Universität Karlsruhe, 2002, to appear.

[BEC 01] BECKERT B., "A Dynamic Logic for the Formal Verification of Java Card Programs", ATTALI I., JENSEN T., Eds., *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, Springer, 2001, p. 6–24.

[BRO 00] BRODSKY S., CLARK T., COOK S., EVANS A., KENT S., "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.

[CLA 01] CLARK T., EVANS A., KENT S., "The Metamodelling Language Calculus: Foundation Semantics for UML", HUSSMANN H., Ed., *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*, vol. 2029 of *LNCS*, Springer, 2001, p. 17–31.

[GOG 01] GOGOLLA M., RICHTERS M., "Expressing UML Class Diagrams Properties with OCL", CLARK T., WARMER J., Eds., *Advances in Object Modelling with the OCL*, p. 86–115, Springer, Berlin, LNCS 2263, 2001.

[HAR 84] HAREL D., "Dynamic Logic", GABBAY D., GUENTHNER F., Eds., *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*, Reidel, 1984.

[MEL 01] MELLOR S. J., D'SOUZA D., CLARK T., EVANS A., KENT S., "Infrastucture and Superstructure of the Unified Modeling Language 2.0 (Response to UML2.0 RfP)", report , 2001, Submission to the OMG.

[OMG 00] OMG, "OMG Unified Modeling Language Specification", report num. OMG-UML V1.3, March 2000, Object Mangagement Group.

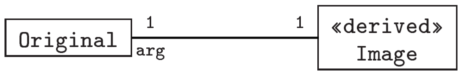
[RIC 01] RICHTERS M., "A precise approach to validating UML models and OCL constraints", PhD thesis, Bremer Institut für Sichere Systeme, Universität Bremen, Logos-Verlag, Berlin, 2001.

[WAR 01] WARMER J., KLEPPE A., CLARK T., IVNER A., HÖGSTRÖM J., GOGOLLA M., RICHTERS M., HUSSMANN H., ZSCHALER S., JOHNSTON S., FRANKEL D. S., BOCK C., "Object Constraint Language 2.0 (Response to UML2.0 RfP)", report , 2001, Submission to the OMG.

A. A Template Catalogue

A.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`.



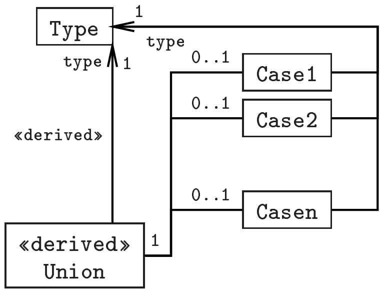
Additional Constraints:

None.

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

The objects of `Union` are isomorphic copies of those of class `Case1`, . . . , `Casen`. Furthermore, objects of `Union` 'inherit' their type from the unique corresponding object of `Case1`, . . . , `Casen`.



Additional Constraints:

1) Each element of `Union` has a link to exactly one element of a `Case` class.

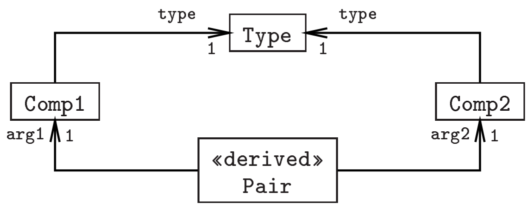
context `Union` **inv:**
```
    self.case1->size + ...  + self.casen->size = 1
```
2) The association between `Union` and `Type` is derived.

For $i = 1..n$ we have the conjunction of invariants:

context `Union` **inv:**
```
    not(self.casei->isEmpty) implies
    self.type = self.casei!type
```

A.3. *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.

```
Pair.allInstances->forAll(p1, p2 |
   p1.arg1 = p2.arg1 and p1.arg2 = p2.arg2 implies p1 = p2)
```

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))
```