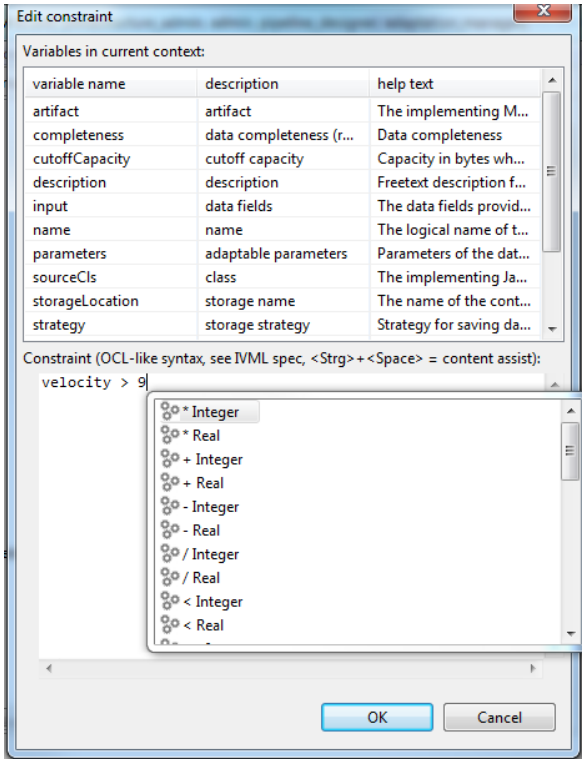
# QualiMaster Infrastructure Configuration Tool – Constraint Guide

This document provides an introduction into the constraint language used in the QualiMaster Infrastructure Configuration Tool (QM-IConf) for specifying Service Level Agreements (SLAs) and runtime constraints. The constraint language is part of the Integrated Variability Modeling Language (IVML) used to describe the QualiMaster configuration model. In this document, we provide a summary of the relevant concepts for runtime constraints in QM-IConf. For more details, please refer to the IVML language guide.

In the configuration model, each configuration setting is described as a typed (decision) variable. Constraints can be used to link and restrict the values of these variables. In QM-IConf, the constraint editor of the data sources, data sinks, algorithms and pipeline elements supports defining constraints through a syntax-driven, content-assisted editor also indicating the variables that can be used in the actual context as illustrated by the figure below:




Below, we summarize the syntax and the semantics of the IVML constraint language used for defining constraints in QM-IConf. As constraints in IVML heavily rely on the Object Constraint Language (OCL), most of the content in this document is taken from OCL and adjusted to the notational conventions and the semantics of IVML. Please note that the constraints in QM-IConf typically refer to **runtime variables**, i.e., variables that are filled by monitoring data at runtime. Typically, constraints shall not modify these variables. In addition, constraints may refer to **pre-runtime variables**, i.e., variables defined by the QM-IConf editors used for instantiating the infrastructure and the pipelines. These variables are **frozen** at runtime and cannot be changed.

Constraints are used to define validity rules for a variability model, e.g. by specifying dependencies among decision variables. The syntax of constraints in the IVML basically follows the structure of expressions in propositional logic and, thus, is composed of:

- Simple sentences, which represent constants, decision variables and types which can be named by (qualified) identifiers.
- Compound sentences created by applying the operations to simple sentences and, in turn, to compound sentences. A correct compound sentence requires that the arguments passed to

operations match the arity of the operation and the types of the parameters or operations, respectively.

The operations available in IVML as well as the type compliance rules will be discussed below.

The constraints in IVML will mostly rely on the relevant part of the syntax as well as on a large subset of the operations defined in OCL. In IVML we use the constraint expression syntax of OCL, but omit the OCL contexts used to relate constraints to UML modelling elements. Similar to OCL, all elements defined in an IVML model will be accessible to constraints. Two examples for constraints are given below, one propositional and one first-order logic example using a quantifier:

- `(10 <= a and a <= 20) implies b == a;`
  If `a` is in the range `(10; 20)` this implies that `b` must have the same value as `a`.
- `1 <= mySet.size() and mySet.size() <= 100;`
  Cardinality restriction of mySet containing arbitrary decision variables.
- `mySet->forAll(x|x > 100);`
  All elements in `mySet` must be larger than `100`

## Reserved Keywords

Keywords in IVML constraint expressions are reserved words. That means that the keywords cannot occur anywhere in an expression as the name of a decision variable or a compound. The list of keywords for the constraint language is shown below:

- **and**
- **def**
- **else**
- **endif**
- **if**
- **iff**
- **implies**
- **in**
- **let**
- **not**
- **or**
- **self**
- **then**
- **xor**

## Prefix operators

IVML defines two prefix operators, the unary

- Boolean negation '**not**'.
- Numerical negation '−' which changes the sign of a Real or an Integer.

## Infix operators

Similar to OCL, in IVML the use of infix operators is allowed. The operators '+,' '−,' '* .' '/,' '<,' '>,' '<>' '<=' '>=' are used as infix operators. If a type defines one of those operators with the correct signature, they will be used as infix operators. The expression:

```
a + b
```

is conceptually equal to the expression:

```
a.+(b)
```

that is, invoking the "+" operation on a (the *operand*) through the dot- access notation with b as the parameter to the operation. The infix operators defined for a type must have exactly one parameter. For the infix operators '<,' '>,' '<=,' '>=,' '<>,' '**and**,' '**or**,' '**xor**', '**implies**', '**iff**' the return type must be Boolean.

Please note that, while using infix operators, in IVML integer is a subclass of real. Thus, for each parameter of type real, you can use integer as the actual parameter. However, the return type will always be real.

Further, please note that expressions on the left side of implications (implies) and two-sided implications (iff) must not be assignments ('=').

## Equality and assignment operators (default logic)

In contrast to OCL, IVML provides two operators which are related to the equality of elements with different semantics, namely the default assignment '=' and the equality constraint operator '==''..
Basically, a decision variable in IVML is considered as **undefined**, i.e., the variable does not have an effect on the instantiation. Constraints may explicitly refer to the undefined state via the operation "isDefined". Variables can have a **default value**, a kind of basic configuration. However, the value of a variable can be modified only once in a given model (assigned or changed). This restriction is required due to the fact that IVML does not provide support to define the sequence of evaluations. As a default value is treated as a shortcut of an assignment, further assignments or changes of the default value must not be done in the same model, i.e., if a default value is assigned to variable *x* in project *P*, *x* may be changed in projects importing P unless frozen, but not directly in *P*.

As the '=' operator defines a default value which may be overridden, it is not possible to use that operator to express that a decision variable must have a certain value (under some conditions). This can be achieved using the equality operator '==''. Basically, the equality operator checks whether the left hand and the right hand operand have **equal values**. In two distinct cases, the equality operator **enforces the value** specified by the right hand operand. The cases are the

- Unconditional value constraint, e.g., `a == 5`.
- Conditional value constraint given as the right side of an implication, e.g., `c < 5 implies a == 5`.

In these two cases, the equality operator expresses that the left hand operand (an expression denoting a decision variable) must have the same value as the right hand operand. If the left hand operand contains a default value, then the default value will be overridden. However, if two expressions aim at enforcing different values for the same decision variable, the model becomes unsatisfiable.

## Precedence rules

The precedence order for the operations, starting with highest precedence, in IVML is:

- dot and arrow operations: '**.**' (for element and operation access) and '->' (to access collection operations such as **forAll** or **exists**).
- unary '**not**' and unary minus '-'
- '*' and '/'
- '+' and binary '-'
- '**if-then-else-endif**'
- '<', '>', '<=', '>='
- '==' (equality), '<>', '!=' (alias for '<>')
- '**and**', '**or**' and '**xor**'
- Default assignment '='

'**`implies`**', '**`iff`**'Parentheses '`(`' and '`)`' can be used to change precedence.

## Datatypes

All datatypes defined in IVML including the user-defined ones such as compounds, restricted types or annotations are available to the constraint language and may be used in constraint expressions. Below, we give some specific notes on the use of datatypes, in particular in relation to OCL.

- In addition to the string operations defined for OCL, we added two operations based on regular expressions, namely matches and substitutes.
- Enumerations literals are used just like qualified names, i.e. using a dot. For a certain enumeration type only the enumeration literals may be used with assignment ('='), equality ('==') or inequality ('`!=`', '<>') operators. In case that ordinals are explicitly specified for enumeration literals, also relational operators ('<', '>', '<=', '>=') may be used.
- Decision variable declarations defined within a compound can be accessed using the dot operator '**`.`**'. `self` refers to the value of a compound and can be used in (implicitly all-quantized) constraints within compounds.
- In addition to the string operations defined for OCL, we added two operations based on regular expressions, namely matches and substitutes.

For more details on the data types and their operations, please refer to the IVML language guide.

## Type operations

IVML provides the following type-specific operations: **isTypeOf**(), **isKindOf()** and **typeOf()**. The first two operations are similar to the related operations in OCL. The latter one returns the actual type (MetaType) of a decision variable, compound field or container element. MetaType allows equality and unequality comparisons. In addition, the collections provide the operations **typeSelect** and **typeReject** which select elements from a collection according to their actual type based on the **isTypeOf** operation. Currently, IVML neither supports re-typing nor casting, but implicit casting though dynamic dispatch of user-defined operations.

## Side effects

IVML is designed as a modelling and configuration language for Software Product Lines. As a configuration language, an assignment of values to decision variables is mandatory. Thus, in contrast to OCL, some constraint expressions in IVML may lead to side effects in terms of value assignments ('='). Please note that all operations except for assignments are free of side effects (similar to OCL).

## Undefined values

Basically, variables are undefined in order to enable partial configuration. Unless a default value ('=') or a value (via '=' or '==') is assigned. Due to undefined variables, some expressions will, when evaluated, have an undefined value. During evaluation, undefined (sub-) expressions are ignored.

## Collection operations

IVML defines many operations on the collection types. These operations are specifically meant to enable a flexible and powerful way of constraining the contents of collections or projecting new collections from existing ones. However, we support only a relevant subset of the various notations in OCL. The different constructs are described in the following paragraphs. All collection operations (and only those) are accessed using the arrow-operator '`->`'.

In the first versions of OCL, all collection operations returned flattened collections, i.e. the entries of nested collections instead of the collections were taken over into the results. However, this was considered as an issue in OCL and does not fit to the explicit hierarchical nesting in IVML. Thus, collection operations in IVML do not apply flattening.

Sometimes an expression using operations results in a collection, while we are interested only in a special subset of the collection. The **select** operation specifies a subset of a collection:

```
collection->select(t|boolean-expression-with-t)

collection->select(ElementType t|

  boolean-expression-with-t)
```

Both expressions result in a collection that contains all the elements from `collection` for which the `boolean-expression-with-t` evaluates to true. Thereby, `t` is an iterator which will successively receive all values stored in `collection`. In the second form the type of the elements is explicitly specified. Note that the type of the iterator must comply with the element type of the collection. To find the result of this expression, for each element in `collection` the expression `boolean-expression-with-t` is evaluated. If this evaluates to true, the element is included in the result collection, otherwise not.

**Example**:

```
/* Get all elements of the set "contents" with a "highBitrate"
of less than 128 */

contents->select(t|t.highBitrate < 128);
```

The **reject** operation is identical to the select operation, but with reject we get the subset of all the elements of the collection for which the expression evaluates to False. The reject syntax is identical to the select syntax.

The select and reject operations always result in a sub-collection of the original collection. When we want to specify a collection which is derived from some other collection, but which contains different objects from the original collection (i.e., it is not a sub-collection), we can use a **collect** operation. The **collect** operation uses the same syntax as the select and reject and is written as one of:

```
collection->collect(t|expression-with-t)

collection->collect(ElementType t|expression-with-t)
```

Many times a constraint is needed on all elements of a collection. The **forAll** operation in IVML allows specifying a Boolean expression, which must hold for all objects in a collection:

```
collection->forAll(t|boolean-expression-with-t)

collection->forAll(ElementType t|

  boolean-expression-with-t)
```

**Example**:

```
/* None of the elements of the set "contents" must have a
"highBitrate" of greater than 512 */

contents->forAll(t|t.highBitrate <= 512);
```

The **forAll** operation has an extended variant in which more than one iterator is used. Both iterators will iterate over the complete collection. Effectively this is a **forAll** on the Cartesian product of the collection with itself.

```
collection->forAll(t1, t2|
```

```
    boolean-expression-with-t1-and-t2)

collection->forAll(ElementType t1, t2|

    boolean-expression-with-t1-and-t2)
```

Many times one needs to know whether there is at least one element in a collection for which a constraint holds. The **exists** operation in IVML allows you to specify a Boolean expression that must hold for at least one object in a collection:

```
collection->exists(t|boolean-expression-with-t)

collection->exists(ElementType t|

    boolean-expression-with-t)
```

Depending on the type of the collection further related operation may be defined such as **isUnique**. Details on the operations of the individual types are given in the IVML language specification.

One special case of collection operation is to aggregate one value over all values in a collection by applying a certain expression or function. However, this comes close to the iterate operation in OCL. As we specifically target value aggregations define the **apply** operation while reusing the already known syntax:

```
collection->apply(t, ResultType r = initial|

    r = expression-with-t)

collection->apply(ElementType t, ResultType r = initial|

    r = expression-with-t)
```

This operation initializes the result "iterator" `r` with the `initial` expression and applies the `expression-with-t` to each element in the collection. The result of `expression-with-t` is used to update successively the result "iterator". Finally, the operation returns the value of `r` after processing the last element in `collection`. Please note that the result "iterator" is always defined using a specific type which, in turn, defines the result type of the apply operation.

**Example**:

```
/* Return the sum of all (default) bitrates of the elements of
the set "contents" */

contents->apply(t, Integer r| r = r + t.bitrate);
```

s

The research leading to these results has received funding from the European Union Seventh Framework Programme [FP7/2007-2013] under grant agreement nr. 619525.