



*International
Virtual
Observatory
Alliance*

PDL: The Parameter Description Language Version 1.0

IVOA Working Draft 10 May 2013

This version:

1.0: 10 May 2013 (SVN Revision 195)

Latest version:

PDL: The Parameter Description Language, IVOA Working draft v. 0.1

Previous versions:

Working Group:

Grid And Web Services

Editor(s):

Authors:

Carlo Maria Zwölf, Paul Harrison and Franck Le Petit

Abstract

This document discusses a Parameter Definition Language (PDL).

1 Status of this document

This document has been produced by the Grid and Web Services Working Group. It is still a draft.

Acknowledgements

TBD

Contents

1	Status of this document	1
2	Introduction	4
2.1	The service description: existing solutions and specific needs	4
2.2	Interoperability issues	5
2.3	A new Parameter Description Language: a unique solutions to description and interoperability needs	6
3	The Service object	7
4	The SingleParameter Object	9
5	The ParameterReference object	10
6	The ParameterType object	11
7	The ParameterGroup object	11
8	The Expression Objects	12
8.1	The AtomicParameter expression	12
8.2	The AtomicConstant expression	13
8.3	The parenthesisContent expression	15
8.4	The Operation object	16
8.5	The FunctionType object	17
8.6	The Function object	17
8.7	The FunctionExpression object	18
9	Expressing complex relations and constraints on parameters	19
9.1	The ConstraintOnGroup Object	19
9.2	The ConditionalStatement object	19
9.2.1	The AlwaysConditionalStatement	19
9.2.2	The IfThenConditionalStatement	20
9.2.3	The WhenConditionalStatement object	20
9.3	The ConditionalClause object	20
9.4	The AbstractCriterion object	22
9.4.1	The Criterion object	23
9.4.2	The ParenthesisCriterion object	23
9.5	The LogicalConnector object	24
9.6	The AbstractCondition object	24
9.6.1	The IsNull condition	24
9.6.2	The "numerical-type" conditions	25
9.6.3	The BelongToSet condition	25
9.6.4	The ValueLargerThan object	26

9.6.5	The ValueSmallerThan object	26
9.6.6	The ValueInRange object	26
9.6.7	The ValueDifferentOf object	28
9.6.8	The DefaultValue object	28
9.7	Evaluating and interpreting criteria objects	29
10	PDL and formal logic	31
11	Description Examples	31

2 Introduction

In the context of the *International Virtual Observatory Alliance* researchers would like to provide astronomical services to the community.

These services could be

- access to an existing catalogue of images and/or data,
- the entry point to a database listing the results of complex and heavy numerical simulations,
- a computation code exposed online, etc...

In the following we will ignore any specific feature and will use the term *generic service* to refer to any kind of process that receives input parameters and produces output ones.

Let us notice that users from the community will not be able to use a new service unless they have the knowledge of what the service does (and how). Moreover this new service will be even more useful if it can be immediately interactive and is well integrated with other services.

Service description and **Interoperability** are indeed two key points for building efficient and useful services.

2.1 The service description: existing solutions and specific needs

For a client starting to interact with an unknown service, its description is fundamental: in a sense it is this description that puts the service from the *unknown* to the *known* state.

Since the client could be a computer system, a generic description should be machine-readable.

There are several description languages. The most known for their high expression level and their wide use are *WSDL* (<http://www.w3.org/TR/wsdl20/>) and *WADL* (<http://www.w3.org/Submission/wadl/>).

With those tools, people providing a given service could easily express what parameters the service expects and what data structures it returns. It thus serves a roughly similar purpose as a method-signature in a programming language.

In the case of *generic services* for science, description needs are very specific: since we have to deal with complex physics and models, one should be able to describe for each parameter its physical meaning, its unit and precision and the range (or set) of admissible values (according to the model).

In many cases, especially for theoretical simulations, parameters could be linked by complex conditions or have to satisfy, under given conditions, a set of constraints (that

could involve mathematical properties and formulas). Two examples of this high level description we would be able to provide are the following:

$$\text{Service1} \left\{ \begin{array}{l} \text{Input} \left\{ \begin{array}{l} \vec{p}_1 \text{ is a } m/s \text{ vector speed and } \|\vec{p}_1\| < c \\ p_2 \text{ is time (in second) and } p_2 \geq 0 \\ p_3 \text{ is a } kg \text{ mass and } p_3 > 0 \end{array} \right. \\ \text{Output} \left\{ \begin{array}{l} p_4 \text{ is a Joule Kinetic Energy and } p_4 \geq 0 \\ p_5 \text{ is a distance (in meter)} \end{array} \right. \end{array} \right. \quad (1)$$

$$\text{Service2} \left\{ \begin{array}{l} \text{Input} \left\{ \begin{array}{l} \mathbb{R} \ni p_1 > 0; p_2 \in \mathbb{N}; p_3 \in \mathbb{R} \\ \bullet \text{ if } p_1 \in]0, \pi/2] \text{ then } p_2 \in \{2; 4; 6\}, \\ p_3 \in [-1, +1] \text{ and } (|\sin(p_1)^{p_2} - p_3|)^{1/2} < 3/2 \\ \bullet \text{ if } p_1 \in]\pi/2, \pi] \text{ then } 0 < p_2 < 10, \\ p_3 > \log(p_2) \text{ and } (p_1 \cdot p_2) \text{ must belong to } \mathbb{N} \end{array} \right. \\ \text{Output} \left\{ \begin{array}{l} \vec{p}_4, \vec{p}_5 \in \mathbb{R}^3 \\ \text{Always } \frac{\|\vec{p}_5\|}{\|\vec{p}_4\|} \leq 0.01 \end{array} \right. \end{array} \right. \quad (2)$$

To our knowledge, no existing description language meets these fine needs coming with scientific services. This leads us naturally to work on a new solution and consider about developing a new description language.

Remark: The PDL descriptions for the two examples above are online: Example 1 and Example 2.

2.2 Interoperability issues

Nowadays, with the massive spread and diffusion of *cloud* services, interoperability has become an important element for the success and usability of services. This remains true in the context of astronomy. For the astronomical community, the ability of systems to work together without restrictions (and without further *ad hoc* implementations) is of high value: this is the ultimate goal that guides the *IVOA*.

Computer scientists have developed different tools for setting up service interoperability and orchestration. The most well known are

- *BAbel* (<https://computation.llnl.gov/casc/components/>),
- *Taverna* (<http://www.taverna.org.uk>),
- *OSGI* and *D-OSGI* (<http://www.osgi.org/>),
- *OPalm* (http://www.cerfacs.fr/globc/PALM_WEB/),
- *GumTree* (<http://docs.codehaus.org/display/GUMTREE/>).

In general, with those tools one could coordinate only the services written with given languages. Moreover the interoperability is achieved only in a basic "computer" way: if the input of the B service is a double and the output of A service is a double too, thus the two services could interact.

Our needs are more complex than this: let us consider a service B' whose inputs are a density and a temperature and a service A' whose outputs are density and temperature too.

The interoperability is not so straightforward: the interaction of the two services has a sense only if the two densities (likewise the two temperatures)

- have the same "computer" type (ex. double),
- are expressed in the same system of units,
- correspond to the same physical concepts (for example, in the service A' density could be an electronic density whereas in the service B' the density could be a mass density)

But things could be more complicated, even if all the previous items are satisfied: the model behind the service B' could implement an Equation of State which is valid only if the product (density \times temperature) is smaller than a given value. Thus the interoperability with A' could be achieved only if the outputs of this last satisfy the condition on product.

Again, as in case of descriptions no existing solutions could meet our needs and we are oriented towards building our own solution.

2.3 A new Parameter Description Language: a unique solutions to description and interoperability needs

To overcome the lack of a solution to our description and interoperability needs, it is proposed to introduce a new language. Our aim is to finely describe the set of parameters (inputs and outputs of a given generic services) in a way that

- could be understood easily by human beings,
- could be interpreted and handled by a computer,
- complex relations and constraints involving parameters could be formulated unambiguously. Indeed we would like to express
 - all the possible mathematical laws/formulas,
 - all the possible conditional sentences (provided they have a logical sense)

involving parameters.

The new language is based on a generic data model (DM). Each object of the DM corresponds to a syntactic element. Sentences are made by building object-structures. Each sentence can be interpreted by a computer by parsing the sentence-related object structure.

For describing the physical scientific concept or model behind a given parameter, the idea is to use *SKOS* concepts (<http://www.w3.org/TR/skos-reference/>) or, in more complicated cases, ontologies.

Since the inputs and outputs of every service (including their constraints and complex conditions) could be described with this fine grained granularity, interoperability becomes possible in the *smart* and *intelligent* sense we really need: services should be able to work out if they can sensibly use their output as input for another one, by simply looking at its description.

With no loss of generality and to ensure that the model could work with the largest possible number of programming languages, we decided to fix it under the form of an XML schema (this choice is also convenient because there are many libraries and tools for handling and parsing XML documents).

Remark: We recall that PDL is a syntactic framework for describing parameters (with related constraints) of generic services. Since a PDL description is rigorous and unambiguous, starting from it, it is possible to verify if the instance of a given parameter (i.e. the value of the parameter that a user send to the service) is consistent with the description.

In what follows in this document, we will often use the terms *evaluate* and *interpret* with reference to an expression and/or conditions composed with PDL. By this we mean that one must replace in the PDL expressions/conditions the referenced parameters by the set of values provided to the service by user. The replacement mechanisms will be explained in detail, case by case.

3 The Service object

The root element of the PDL description of a generic service is the object *Service* (see figure 1). This **must contain**

- A unique *ServiceName*. This field is a String containing the name of the service.
- A unique *ServiceId*. This field is a String containing the technical Id of the service. It is introduced for a future eventual integration of PDL into the IVOA registry. Each service in the registry will be marked with its own unique id.
- A unique *Description*. This field is a String and contains a human readable description of the service. This description is not intended to be understood/parsed by a machine.
- A unique *Parameters* field which is a list of *SingleParameter* object type (cf. paragraph 4). This list contains the definition of all parameters (both inputs and outputs) of the service. The two following fields specify if a given parameter is a input or an output one.
- A unique *Inputs* field of type *ParameterGroup* (cf. paragraph 7). This object

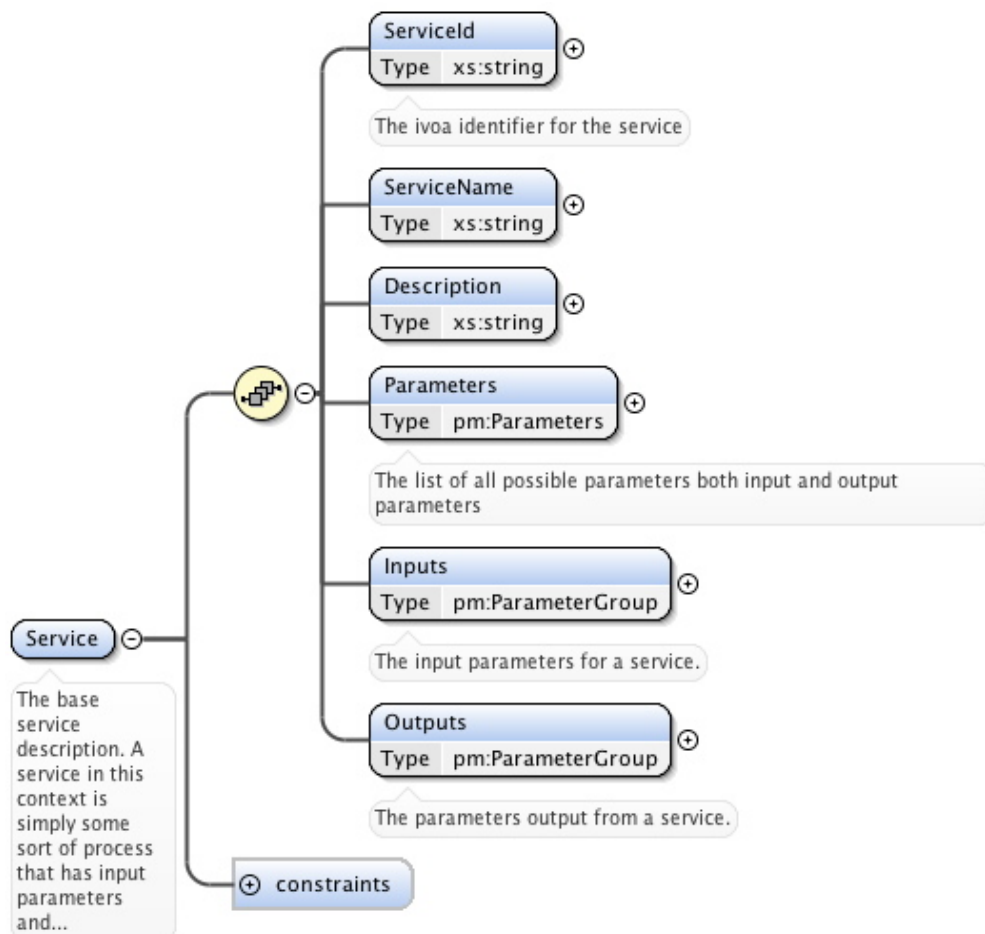


Figure 1: Graphical representation of the Service object

contain the detailed description (with constraints and conditions) of all the input parameters.

- A unique *Outputs* field of type *ParameterGroup*. This object contain the detailed description (with constraints and conditions) of all the output parameters.

4 The SingleParameter Object

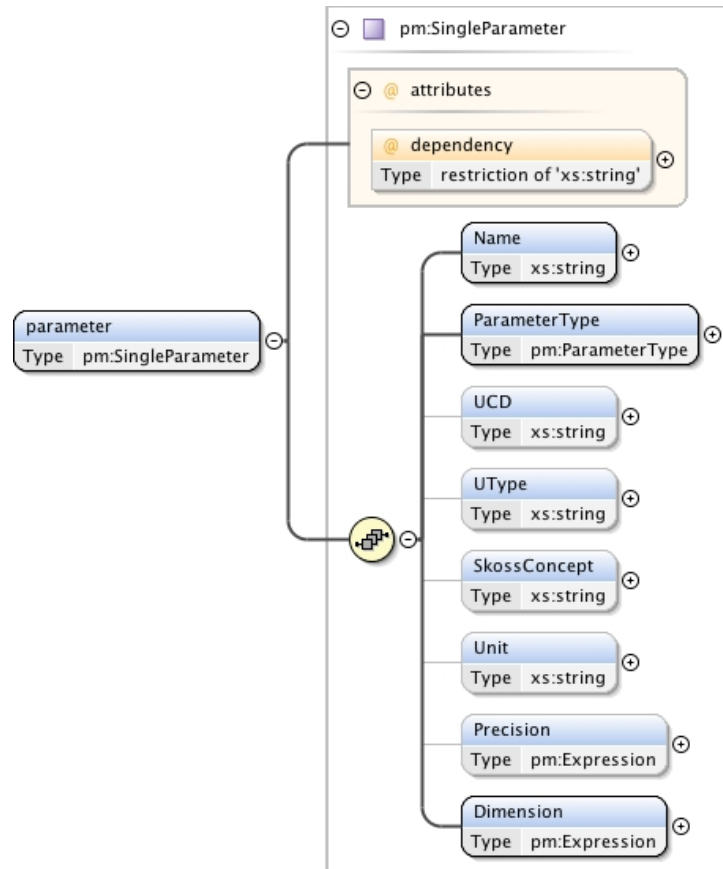


Figure 2: Graphical representation of the Parameter object

The *SingleParameter* object (see figure 2) is the core element for describing jobs. Every object of this type must be characterized by:

- A name (which is unique and is the Id of the parameter);
- A unique parameter type, which explains the nature of the current parameter. The allowed types are : boolean, string, rational, complex, integer, real, date;
- A unique dimension. A 1-dimension corresponds to a scalar parameter whereas a dimension equal to N corresponds to a N-size vector. The dimension is expressed

using an *expression* (cf. paragraph 8). The result of the expression that appears in this *SingleParameter*-field object **must be integer**.¹

The unique attribute *dependency* can take one of the two values **required** or **optional**. If required the parameter **must be** provided to the service. If optional, the service could work even without the current parameter and the values will be considered for processing only if provided.

Optional fields for the *SingleParameter* object are:

- a unique UCD : which is a reference to an existing UCD for characterizing the parameter (**the reference is typically a text string**);
- a unique Utype : which is a reference to an existing Utype for characterizing the parameter (**the reference is typically a text string**);
- a unique Skos Concept (**the reference is typically a text string**).
- a unique Unit (**which is a string reference to a valid VOUnits element**).
- a unique precision. This field must be specified only for parameter types where the concept of precision has a meaning. It has indeed no sense for integer, rational or string. It has sense, for instance, on a real type. For understanding the meaning of this field, let the function f be a model of a given service. If i denotes the input parameter, $f(i)$ denotes the output. The precision δ is the smaller value such that $f(i + \delta) \neq f(i)$.

The precision is expressed using an *expression* (cf. paragraph 8). The result of the expression that appears in this *precision*-field **must be** of the same type as (or could be naturally cast to) the type appearing in the field *parameter type*.

NB: The name of every *SingleParameter* is unique.

5 The ParameterReference object

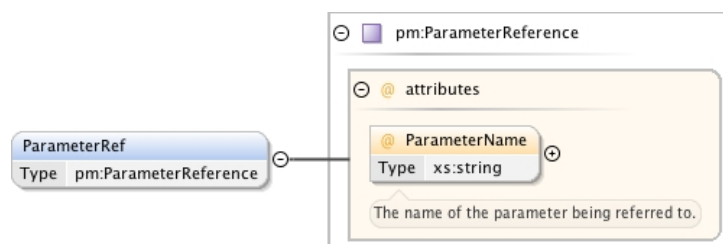


Figure 3: Graphical representation of the Parameter Reference object

This object, as its name indicates, is used to reference an existing parameter defined in the *Service* context (cf. paragraph 3). It contains only a unique attribute

¹This is obvious, since this value corresponds to a vector size.

ParameterName of type String which must corresponds to the *Name* field of an existing *SingleParameter* (cf. paragraph 4).

6 The ParameterType object

This object is used to explain the type of a parameter (cf. paragraph 4) or an expression (cf. paragraph 8.2). The allowed types are : boolean, string, rational, complex, integer, real, date;

7 The ParameterGroup object

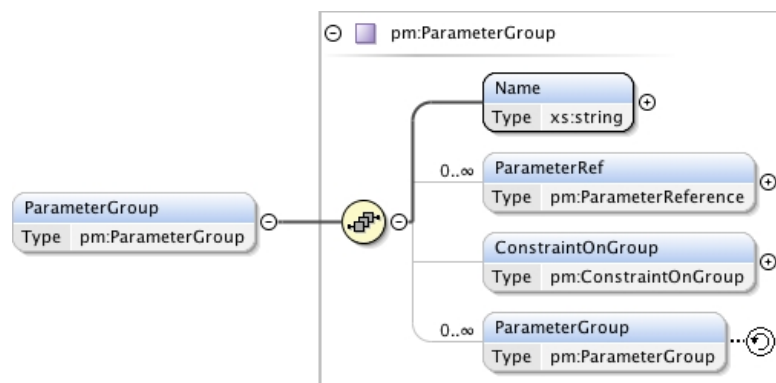


Figure 4: Graphical representation of the ParameterGroup object

The *ParameterGroup* object (see figure 4) is used for grouping parameters according to a criterion of relevancy arbitrarily chosen by users (for instance parameters may be grouped according to the physics : position-group, speed-group; thermodynamic-group). However, the *ParameterGroup* is not only a kind of parameter set, but also can be used for defining complex relations and/or constraints involving the contained parameters (cf. paragraph 9.1).

This object **must contain** a unique Name. This name is a String and is the identification label of the *ParameterGroup*, and no two groups can have the same Name.

Optional fields are

- the references to the parameters (cf. paragraph 5) one want to include into the group;
- a unique object *ConstraintOnGroup* of type *ConstraintOnGroup* (cf. paragraph 9.1). This object is used for expressing the complex relations and constraints involving parameters.
- the objects of type *ParametersGroup* contained within the current root group. Indeed the *ParametersGroup* is a recursive object which can contain other sub-groups.

NB: The name of every *ParameterGroup* is unique.

NB: A given *SingleParameter* object could only belong to one *ParameterGroup*².

NB: For any practical use, the number on the parameter referenced into a given group summed to the number of sub-groups of the same group must be greater than one. Otherwise the group would be a hollow shell.

8 The Expression Objects

The *Expression* is the most versatile component of the PDL. It occurs almost everywhere: in defining fields for *SingleParameters* (cf. paragraph 4) or in defining conditions and criteria).

Expression itself is an abstract object. In this section we are going to review all the concrete object extending and specializing expressions.

N.B. In what follows, we will call a **numerical expression** every *expression* involving only numerical types. This means that the evaluation of such expressions should lead to a number (or a vector number if the dimension of the expression is greater than one).

8.1 The AtomicParameter expression

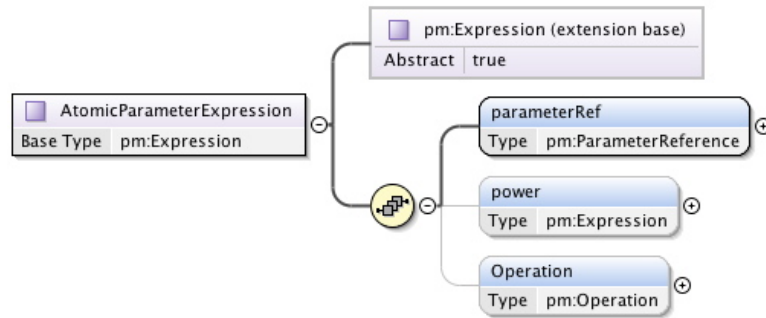


Figure 5: Graphical representation of the AtomicParameter expression object

The *AtomicParameterExpression* (extending *Expression*, see figure 5) is the simplest expression that could be built involving a defined parameter. This object **must contain** unique reference to a given parameter.

Optional fields, valid only for numerical types, are :

²As we will see in paragraph 9.1, constraints on parameters are defined at the level of the group. If a *SingleParameter* belongs only to one group, it will be easier to verify that there is no contradictions on conditions

- A unique **numerical** power expression;
- A unique operation (cf. paragraph 8.4).

Let p and exp be respectively the parameter and the power expression we want to encapsulate. The composite object could be presented as follows:

$$\begin{array}{c}
 \text{Operation type} \\
 \overbrace{\left(\begin{array}{c} + \\ - \\ * \\ \cdot \\ \div \end{array} \right)}^{\text{expression contained in operation}} \\
 p^{exp} \quad \underbrace{\left(\text{AnotherExpression} \right)}_{\text{Operation object}} \\
 \underbrace{\hspace{10em}}_{\text{Atomic Parameter Expression}}
 \end{array} \tag{3}$$

To evaluate a given *AtomicParameterExpression*, one proceeds as follows: Let d_p , d_{exp} and d_{oo} be respectively the dimension of the parameter p referenced, the dimension of the power expression and the dimension of the expression contained within the operation object.

The exponent part of the expression is legal if and only if:

- $d_p = d_{exp}$. In this case p^{exp} is a d_p -size vector expression and $\forall i = 1, \dots, d_p$ the i component of this vector is equal to $p_i^{exp_i}$, where p_i is the value of the i component of vector parameter p and exp_i is the value obtained by interpreting the i component of vector expression exp .
- Or $d_{exp} = 1$. In this case, $\forall i = 1, \dots, d_p$ the i component of the vector result is equal to p_i^{exp} , where p_i is the same as defined above.

Whatever the method used, let us note ep the result of this first step. It is clear that the dimension of ep is always equal to d_p . In order to complete the evaluation of the expression, one should proceed as shown in paragraph 8.4, by setting there $b = ep$.

8.2 The AtomicConstant expression

The *AtomicConstantExpression* (extending *Expression*, see figure 6) is the simplest expression that could be built involving constants. Since this object could be used for defining a constant vector expression, it **must contain**

- A unique list of String which expresses the value of each component of the expression. Let d_c be the size of the String list. If $d_c = 1$ the expression is scalar and it is a vector expression if $d_c > 1$.
- A unique attribute *ConstantType* of type *ParameterType* (cf. paragraph 6) which defines the nature of the constant expression. The allowed types are the same as in the field *parameterType* of the object *SingleParameter*.

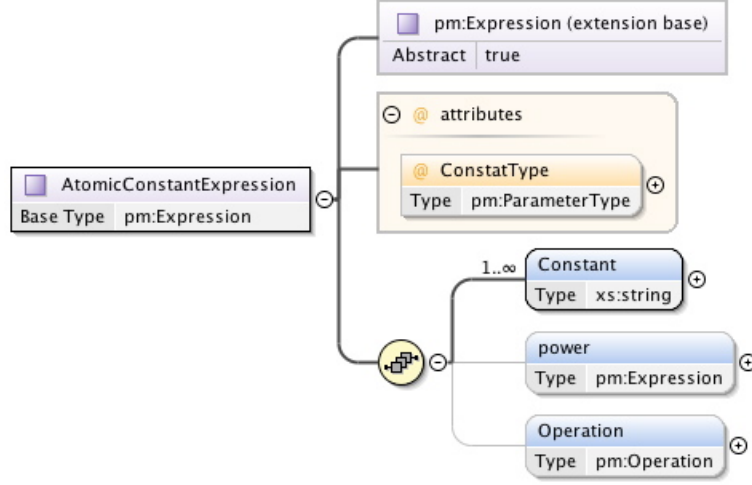


Figure 6: Graphical representation of the AtomicParameter expression object

The object **is legal if and only if** every element of the String list could be cast into the type expressed by the attribute *constantType*.

Optional fields, valid only for numerical types, are :

- A unique **numerical** power expression;
- A unique operation (cf. paragraph 8.4).

Let s_i ($i = 1, \dots, d_c$) and exp be respectively the i component of the String list and the power expression we want to encapsulate. The composite object could be presented as follows:

$$\underbrace{\underbrace{\underbrace{(s_1, s_2, \dots, s_{d_c})^{exp}}_{\text{List of String to cast into the provided type}} \quad \underbrace{\begin{pmatrix} + \\ - \\ * \\ \cdot \\ \div \end{pmatrix}}_{\substack{\text{Operation type} \\ \text{expression contained in operation}}} \quad \underbrace{(\text{AnotherExpression})}_{\text{Operation object}}}_{\text{Atomic Constant Expression}} \quad (4)$$

To evaluate a given *atomicConstantExpression*, one proceeds as follows: let d_{exp} and d_{oo} be respectively the dimension of the parameter p referenced, the dimension of the power expression and the dimension of the expression contained within the operation object.

The exponent part of the expression is legal if and only if:

- $d_c = d_{exp}$. In this case $(s_1, \dots, s_{d_c})^{exp}$ is a d_c size vector expression and $\forall i = 1, \dots, d_c$ the i -th component of this vector is equal to $s_i^{exp_i}$, where exp_i is the value obtained by interpreting the i component of vector exp .

- Or $d_{exp} = 1$. In this case, $\forall i = 1, \dots, d_c$ the i component of the vector result is equal to s_i^{exp} .

Whatever the method used, let us note ep (whose dimension is always equal to d_c) is the result of this first step. In order to complete the evaluation of the expression, one should proceed as exposed in paragraph 8.4, by substituting there $b = ep$.

8.3 The parenthesisContent expression

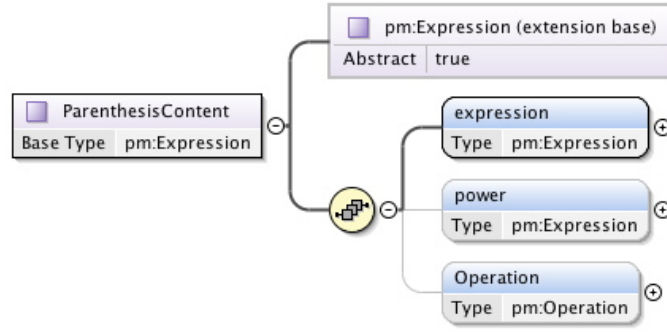


Figure 7: Graphical representation of the ParenthesisContent expression object

The *parenthesisContentExpression* (extending *Expression*, see 7) object is used to explicitly denote precedence by grouping the expressions that should be evaluated first. This object **must contain** a unique **numerical** object *Expression* (referred to hereafter as exp_1).

Optional fields are

- A unique **numerical** power expression (referred to hereafter as exp_2);
- A unique operation (cf. paragraph 8.4).

This composite object could be presented as follows:

$$\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{(exp_1)}_{\text{Priority term}}}_{exp_2}}_{\text{Operation type}}}_{\text{expression contained in operation}}}_{\text{Operation object}}_{\text{Parenthesis Expression}} \quad (5)$$

In order to evaluate this object expression, one proceeds as follows: first one evaluates the expression exp_1 that has the main priority. Then one proceeds exactly as in paragraph 8.1 (after the equation (3)) by substituting $p = exp_1$ and $exp = exp_2$.

8.4 The Operation object

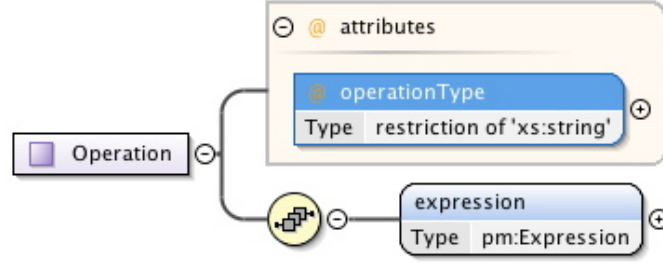


Figure 8: Graphical representation of Operation object

The *Operation* object (see figure 8) is used for expressing operations involving two **numerical** expressions. This object **must contain**:

- a unique attribute *operationType*. This attribute could take the following values: plus for the sum, minus for the difference, multiply for the standard product, *scalarProduct* for the scalar product and *divide* for the standard division. Hereafter these operators will be respectively denoted $+$, $-$, $*$, \cdot , \div .
- a unique expression.

$$\underbrace{\left(\begin{array}{c} + \\ - \\ * \\ \cdot \\ \div \end{array} \right)}_{\text{Operation type}} \underbrace{\left(\text{expression contained in operation} \right)}_{\text{(ContainedExpression)}} \quad (6)$$

Operation object

The *Operation* object is always contained within a **numerical** *Expression* (cf. paragraph 8) and could not exist alone. Let a be the result of the evaluation of the expression object containing the operation³ and let b be the result of the evaluation of the **numerical** expression contained within the operation. As usual, we note d_a and d_b the dimensions of a and b .

The operation evaluation is legal if and only if:

³this came from the evaluation of *parameterRef* field in case of an *AtomicParameterExpression* cf. paragraph 8, from the evaluation of *constant* field in the case of a *AtomicConstantExpression* (to be extended...)

- $d_a = d_b$ and operation type (i.e. the operator) $op \in \{+, -, *, \div\}$. In this case $a \text{ op } b$ is a vector expression of size d_a and $\forall i = 1, \dots, d_a$ the i component of this vector is equal to $(a_i \text{ op } b_i)$ (i.e. a term by term operation).
- Or $d_a = d_b$ and operation type op is \cdot . In this case $a \cdot b$ is the result of the scalar product $\sum_{i=1}^{d_a} a_i * b_i$. It is obvious that the dimension of this result is equal to 1.
- Or $d_b = 1$ and operation type (i.e. the operator) $op \in \{+, -, *, \div\}$. In this case $a \text{ op } b$ is a vector expression of size d_a and $\forall i = 1, \dots, d_a$ the i component of this vector is equal to $(a_i \text{ op } b)$.
- Or $d_a = 1$ and operation type (i.e. the operator) $op \in \{+, -, *, \div\}$. This case is symmetric to the previous one.

The type of the result is automatically induced by standard cast operation performed during the evaluations (Indeed for example a double vector added to an integer vector is a double vector).

8.5 The FunctionType object

This object is used for specifying the mathematical nature of the function contained within a *Function* object (cf. paragraph 8.6). The unique String field this object contains could take one of these values: size, abs, sin, cos, tan, asin, acos, atan, exp, log, sum, product. In paragraph 8.6 it is explained how these different function types are used and handled.

8.6 The Function object

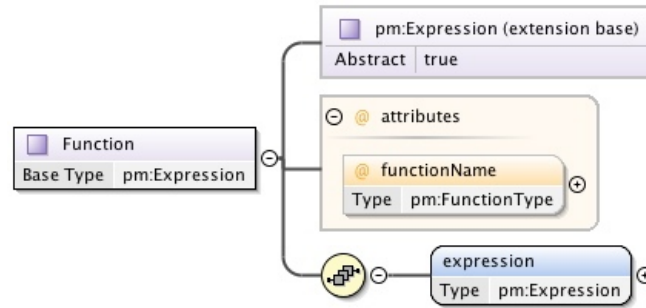


Figure 9: Graphical representation of Function object

The *function* object (extending *expression*, see figure 9) is used for expressing a mathematical function on expressions. This object **must contain**

- A unique attribute *functionName* of type *functionType* (cf. paragraph 8.5) which specifies the nature of the function.
- A unique expression (which is the function argument).

Let exp be the result of the evaluation of the function argument expression and d_{exp} its dimension. The *function* object evaluation is **legal if and only if**:

- $f \in \{\text{abs}, \text{sin}, \text{cos}, \text{tan}, \text{asin}, \text{acos}, \text{atan}, \text{exp}, \text{log}\}$ and the function argument is a **numerical** expression. In this case the result is a d_{exp} -size vector and each component $r_i = f(exp_i)$, $\forall i = 1, \dots, d_{exp}$.
- Or $f = \text{sum}$ (likewise $f = \text{product}$) and the argument is a **numerical** expression. In this case the result is a scalar value equal to $\sum_{i=1}^{d_{exp}} exp_i$ (likewise $\prod_{i=1}^{d_{exp}} exp_i$), where exp_i is the value obtained by interpreting the i component of vector expression exp .
- Or $f = \text{size}$. In this case the result is the scalar integer value d_{exp} .

From what we saw above, the result of the interpretation of a function object is **always a number**.

8.7 The FunctionExpression object

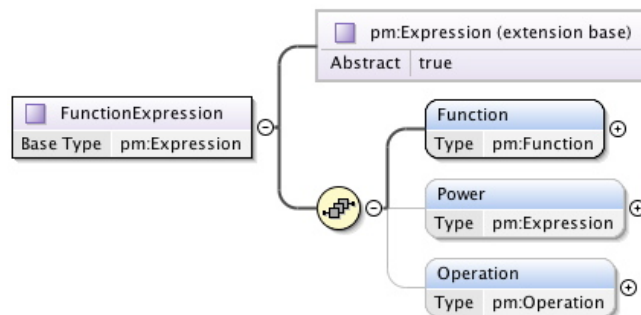


Figure 10: Graphical representation of FunctionExpression object

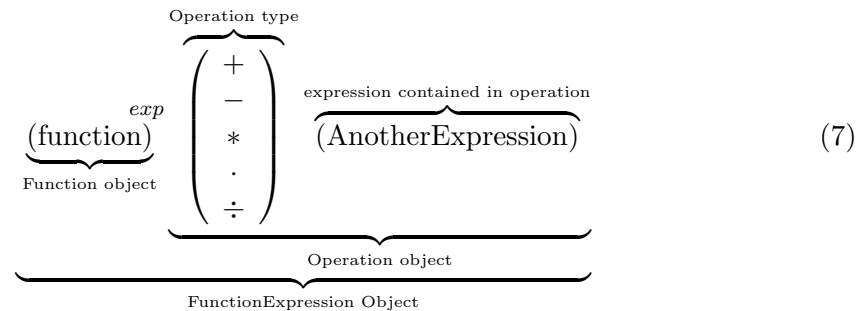
The *FunctionExpression* object (extending *Expression*, see figure 10) is used for building mathematical expressions involving functions.

This object **must contains** a unique *Function* object (cf. paragraph 8.6).

Optional fields, valid only for numerical types, are :

- A unique **numerical** power expression;
- A unique operation (cf. paragraph 8.4).

This composite object could be presented as follows:



In order to evaluate this object expression, one proceed as follows: first one evaluate the funtion expression as explained in paragraph 8.6. Then one proceed exactly as in paragraph 8.1 (after the equation (3)) by taking p =function.

9 Expressing complex relations and constraints on parameters

In this part of the document we will explain how PDL objects could be used for building complex constraints and conditions involving input and/or output parameters.

9.1 The ConstraintOnGroup Object



Figure 11: Graphical representation of ConstraintOnGroup object

The *ConstraintOnGroup* object (see figure 11) is always contained within a *ParameterGroup* object and could not exist alone. This object **must contain** the *ConditionalStatement* objects. The latter are used, as is shown in paragraph 9.2, for expressing the complex relations and constraints involving parameters.

9.2 The ConditionalStatement object

The *ConditionalStatement* object, as its name indicates, is used for defining conditional statements. This object is abstract. In this section we are going to review the two concrete objects extending and specializing *ConditionalStatement*.

9.2.1 The AlwaysConditionalStatement

As its name indicates, this object (see figure 12) is used for expressing statement that must always be valid. It **must contain** a unique *Always* object (which extends *ConditionalClause*, cf. paragraph 9.3).

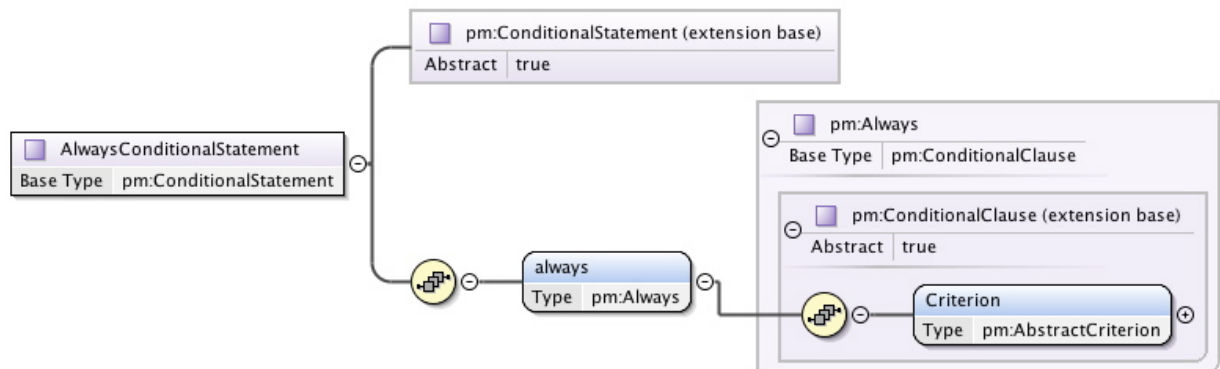


Figure 12: Graphical representation of AlwaysConditionalStatement object

9.2.2 The IfThenConditionalStatement

As its name indicates, this object (see figure 13) is used for expressing statements that are valid only if a previous condition is verified. It **must contain**:

- a unique *If* object (which extends *ConditionalClause*, cf. paragraph 9.3).
- a unique *Then* object (which extends *ConditionalClause*, cf. paragraph 9.3).

If the condition contained within the *If* object is valid, the condition contained within the *Then* object **must be** valid too.

9.2.3 The WhenConditionalStatement object

The when conditional statement is valid when the enclosed *When* conditional clause evalutes to true.

9.3 The ConditionalClause object

The *ConditionalClause* object (see figure 14) is abstract. It **must contain** a unique Criterion object of type *AbstractCriterion* (cf. paragraph 9.4).

The four concrete objects extending the abstract *ConditionalClause* are (see figure 15):

- *Always*;
- *If*;
- *Then*;
- *When*.

The Criterion contained within a *Always* object must always be valid (cf paragraph).

The Criterion contained within a *When* object will be valid only when the enclosed Expression is True.

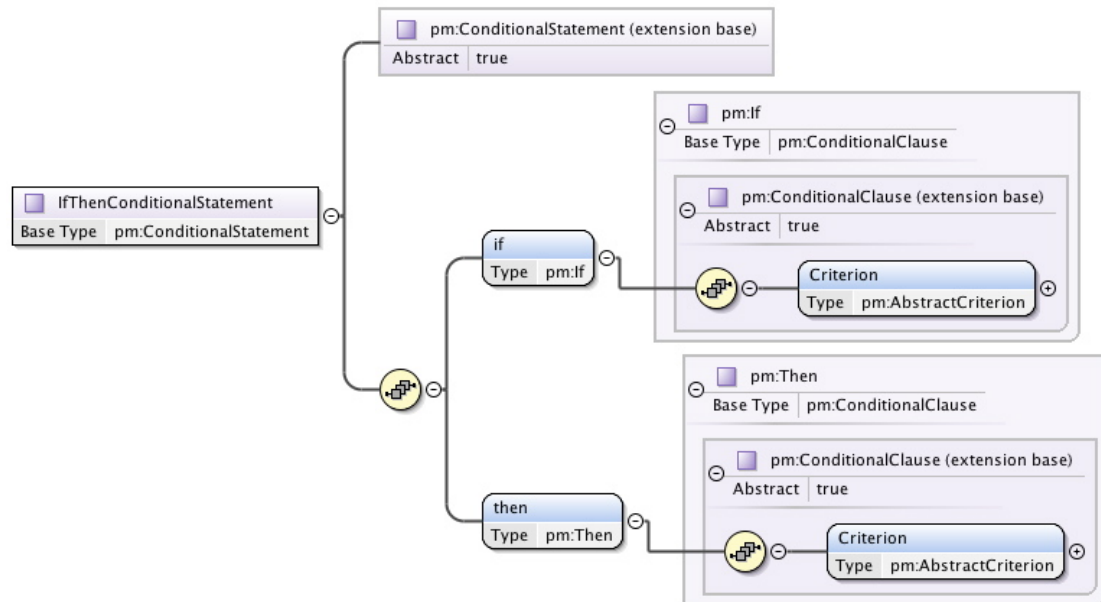


Figure 13: Graphical representation of IfThenConditionalStatement object

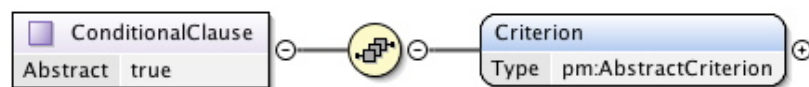


Figure 14: Graphical representation of ConditionalClause object

TODO new picture including When

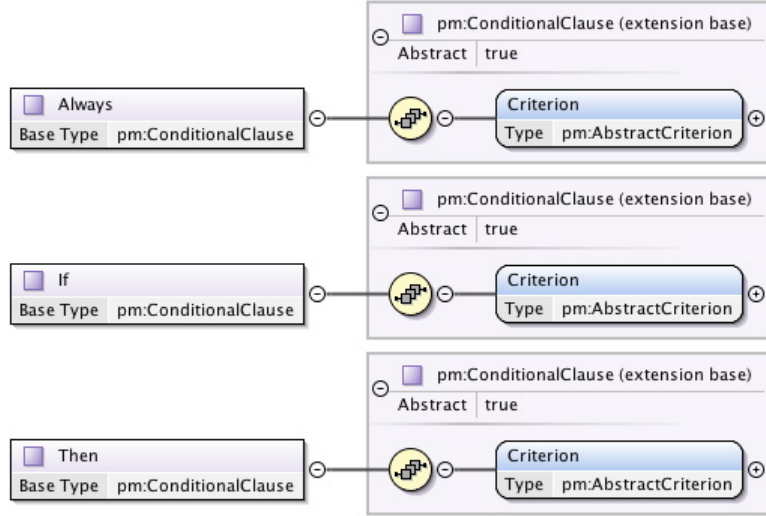


Figure 15: Graphical representation of Always, If and Then clauses

The *If* and *Then* objects work as a tuple by composing the *IfThenConditionalStatement* (cf. paragraph 9.2.2).

9.4 The AbstractCriterion object

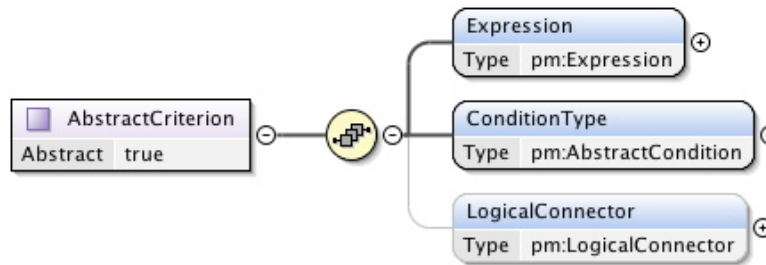


Figure 16: Graphical representation of AbstractCriterion object

The objects extending *AbstractCriterion* (see figure 16) are fundamentals for building *ConditionalStatements* (cf. paragraph 9.2) since they are contained within the *Always*, *If* and *Then* objects (cf. paragraph 9.3). An *AbstractCriterion* object **must contain**:

- a unique *Expression* object (cf. paragraph 8);
- a unique *ConditionType* which is an object of type *AbstractCondition* (cf. paragraph 9.6). This object specifies which condition must be satisfied by the previous *Expression*.

An optional field is the unique *LogicalConnector* object (cf. paragraph 9.5) used for building logical expressions.

The two concrete objects extending *AbstractCriterion* are *Criterion* and *ParenthesisCriterion*. The difference between these two objects is in the priority they induce for interpreting and linking the criteria (cf. paragraph 9.7).

9.4.1 The Criterion object

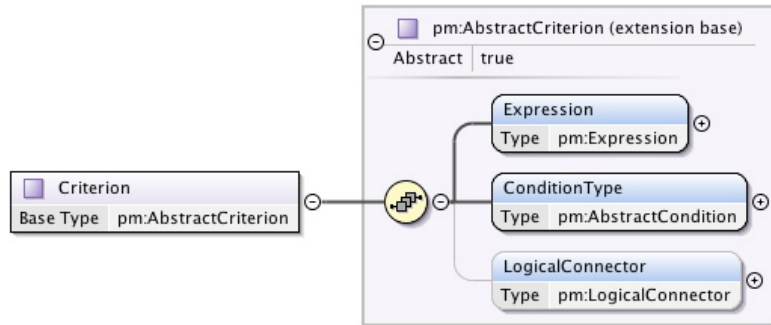


Figure 17: Graphical representation of Criterion object

This object (see figure 17) extends the *AbstractCriterion* without specializing it. It is indeed just a concrete version of the abstract type.

9.4.2 The ParenthesisCriterion object

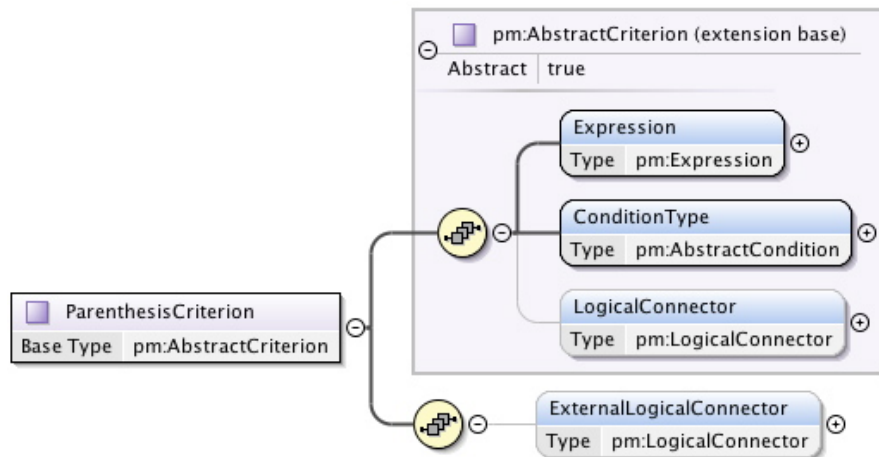


Figure 18: Graphical representation of ParenthesisCriterion object

This object (see figure 18) extends and specialize the *AbstractCriterion*. It is used for defining arbitrary priority in interpreting boolean expression based on criteria. The optional field of *ParenthesisCriterion* is a unique *ExternalLogicalConnector* object of type *LogicalConnector*. It is used for linking other criteria, out of the priority perimeter defined by the parenthesis (cf. paragraph 9.7).

9.5 The LogicalConnector object

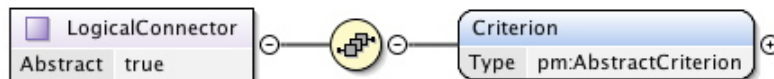


Figure 19: Graphical representation of LogicalConnector object

The *LogicalConnector* object (see figure 19) is used for building complex logical expressions. It is an abstract object and it **must** contain a unique Criterion of type *AbstractCriterion* (cf. paragraph 9.4).

The two concrete objects extending *LogicalConnector* are:

- the *And* object used for introducing the logical AND operator between two criteria;⁴
- the *Or* object used for introducing the logical OR operator between two criteria.

9.6 The AbstractCondition object

AbstractCondition is abstract type. The objects extending it always belong to an *AbstractCriterion* (cf. 9.4). In this context, they are used combined with an *Expression* object, for expressing the condition that the expression must satisfy.

Let us consider a given criterion object \mathcal{CR} (extending *AbstractCriterion*) and let us note \mathcal{E} and \mathcal{C} the expression and the condition contained within \mathcal{CR} . In what follows we are going to explain the different objects specializing *AbstractCondition* and their behavior.

9.6.1 The IsNull condition

This object is used for specifying that the expression \mathcal{E} has no assigned value (this is exactly the same concept as the NULL value in Java or the None value in Python). Indeed, if and only if \mathcal{E} has no assigned value, the evaluation of the tuple $(\mathcal{E}, \mathcal{C})$ leads to a TRUE boolean value. Thus, in the case \mathcal{CR} has no *LogicalConnector*, the criterion is true.

⁴The first criterion is the one containing the *LogicalConnector* and the second is the criterion contained within the connector itself.

9.6.2 The "numerical-type" conditions

These objects are used for specifying that the result of the evaluation of the expression \mathcal{E} is of a given numerical type. The tuple $(\mathcal{E}, \mathcal{C})$ is legal if and only if \mathcal{E} is a **numerical** expression.

The "numerical-type" objects extending *AbstractCondition* are:

- *IsInteger*, in this case the evaluation of the tuple $(\mathcal{E}, \mathcal{C})$ leads to a TRUE boolean value if and only if the evaluation of the numerical expression \mathcal{E} is an integer.
- *IsRational*, in this case the evaluation of the tuple $(\mathcal{E}, \mathcal{C})$ leads to a TRUE boolean value if and only if the evaluation of the numerical expression \mathcal{E} is a rational number.
- *IsReal*, in this case the evaluation of the tuple $(\mathcal{E}, \mathcal{C})$ leads to a TRUE boolean value if and only if the evaluation of the numerical expression \mathcal{E} is a real number.

9.6.3 The BelongToSet condition

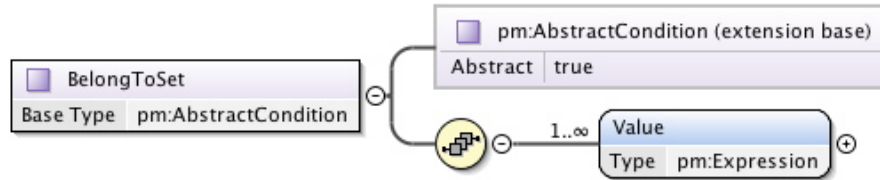


Figure 20: Graphical representation of BelongToSet object

This object (see figure 20) is used for specifying that the expression \mathcal{E} could take only a finite set of values. It **must contain** the *Values* (which are objects of type *Expression*) defining the set of legal values. The number of *Values* must be greater than one.

This object is legal only if all the *Expressions* of the set are of the same type (e.g. they are all numerical, or all boolean or all String expressions).

The tuple $(\mathcal{E}, \mathcal{C})$ leads to a TRUE boolean value if and only if:

- the expression \mathcal{E} and the expressions composing the set are of the same type
- and an element \mathcal{E}_s exists in the set such that $\mathcal{E}_s = \mathcal{E}$.

This last equality is to be understood in the following sense: let $=_t$ be the equality operator induced by the type (for numerical type the equality is in the real number sense, for String type the equality is case sensitive and for boolean the equality is in the classic boolean sense).

Two expressions are equal if and only if

- the expressions have the same size $d_{\mathcal{E}}$,
- and $\mathcal{E}_s^i =_t \mathcal{E}^i$, $\forall i = 1, \dots, d_{\mathcal{E}}$, where \mathcal{E}_s^i and \mathcal{E}^i are respectively the result of the evaluation of the i component of expressions \mathcal{E}_s and \mathcal{E} .

9.6.4 The ValueLargerThan object

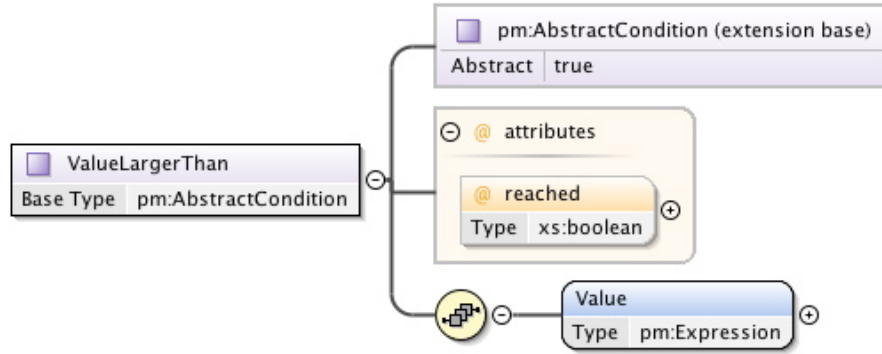


Figure 21: Graphical representation of ValueLargerThan object

This object (see figure 21) is used for expressing that the result of the evaluation of the expression \mathcal{E} must be greater than a given value.

It **must contain**

- a unique **numerical** expression \mathcal{E}_c .
- a unique *Reached* attribute which is a boolean type.

The tuple $(\mathcal{E}, \mathcal{C})$ is legal only if \mathcal{E} is a numerical expression.

This tuple leads to a TRUE boolean value if and only if the result of the evaluation of the expression \mathcal{E} is greater than the result of the evaluation of the expression \mathcal{E}_c and the attribute *Reached* is false. Otherwise if the *Reached* attribute is true the expression \mathcal{E} may be greater than or equal to the result.

9.6.5 The ValueSmallerThan object

This object (see figure 22) is used for expressing that the result of the evaluation of the expression \mathcal{E} must be smaller than a given value.

It **must contain**

- a unique **numerical** expression \mathcal{E}_c .
- a unique *Reached* attribute which is a boolean type.

The tuple $(\mathcal{E}, \mathcal{C})$ is legal only if \mathcal{E} is a numerical expression.

This tuple leads to a TRUE boolean value if and only if the result of the evaluation of the expression \mathcal{E} is smaller (otherwise smaller or equal when the attribute *Reached* is true) than the result of the evaluation of the expression \mathcal{E}_c .

9.6.6 The ValueInRange object

This object (see figure 23) is used for expressing that the result of the evaluation of the expression \mathcal{E} must belong to a given interval. The definition of the interval is made

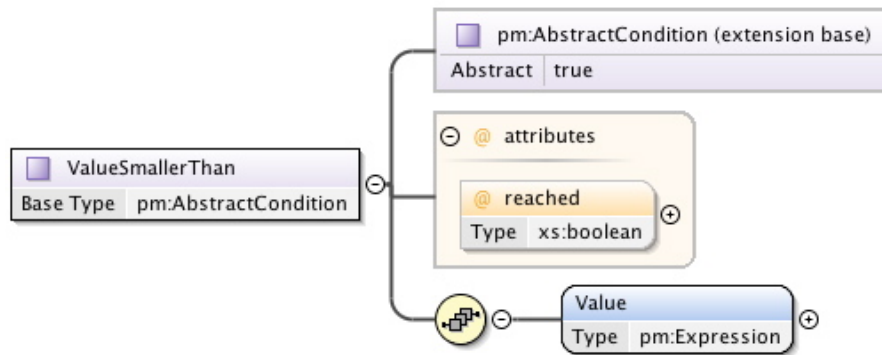


Figure 22: Graphical representation of ValueSmallerThan object

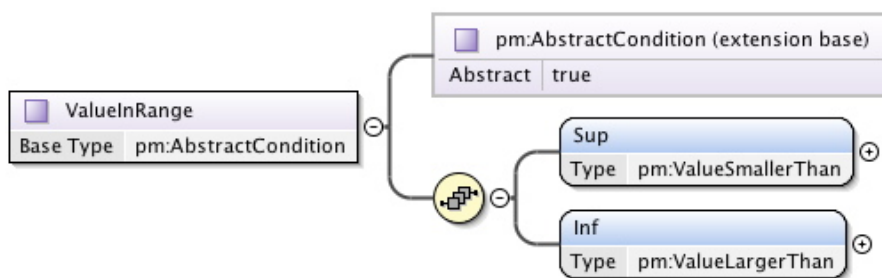


Figure 23: Graphical representation of ValueInRange object

using the *ValueLargerThan* *ValueSmallerThan* objects. Indeed, the *ValueInRange* object **must contain**:

- a unique *ValueLargerThan* object,
- a unique *ValueSmallerThan* object.

The tuple $(\mathcal{E}, \mathcal{C})$ is legal only if \mathcal{E} is a numerical expression.

This tuple leads to a TRUE boolean value if and only if the evaluation of both tuples $(\mathcal{E}, \text{ValueSmallerThan})$ and $(\mathcal{E}, \text{ValueLargerThan})$ lead to TRUE boolean values.

9.6.7 The ValueDifferentOf object

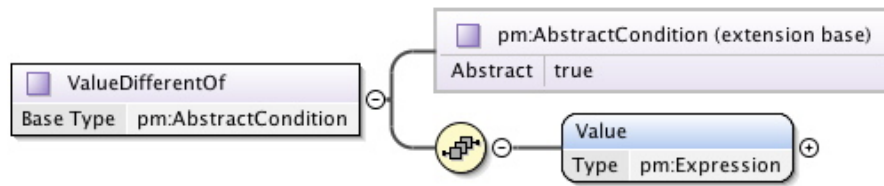


Figure 24: Graphical representation of ValueDifferentOf object

This object (see figure 24) is used for specifying that the expression \mathcal{E} must be different from a given value. It **must contain** a unique *Expression* \mathcal{E}_c .

In order to be compared, the two expressions \mathcal{E} and \mathcal{E}_c must have the same type. The evaluation of the tuple $(\mathcal{E}, \mathcal{C})$ leads to a TRUE boolean value only if $\mathcal{E} \neq \mathcal{E}_c$. This inequality has to be understood in the sense explained in paragraph 9.6.3 (in the second point of the list).

9.6.8 The DefaultValue object

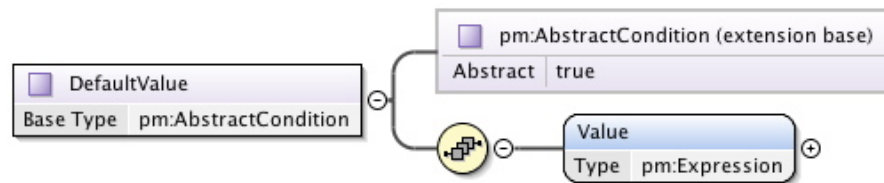


Figure 25: Graphical representation of DefaultValue object

This object (see figure 25) is used for specifying the default value of a parameter.

It **must contain** a unique expression \mathcal{E}_c .

Since the default value of an expression involving functions, multiple parameters, etc. has no particular sense, in the case of the present object the tuple $(\mathcal{E}, \mathcal{C})$ is legal only if

- \mathcal{E} is an *AtomicParameterExpression* (cf. paragraph. 8.1)
- and the dimension and the type of the expression \mathcal{E}_c are equal to the dimension and type expressed in the *SingleParameter* object referenced into the *AtomicParameterExpression*.

Moreover, for having a legal *DefaultValue* object, the criterion \mathcal{CR} containing it must be contained within the *Always* or *Then* objects (cf. paragraph 9.3).

9.7 Evaluating and interpreting criteria objects

The evaluation of the criterion type objects (cf. paragraph 9.4) always leads to a boolean value (the only exception is what we saw in paragraph 9.6.8, where the criterion contains a *DefaultValue* condition).

We use hereafter the same notation introduced in 9.6: let us consider a given criterion (extending *AbstractCriterion*) \mathcal{CR} and let us note \mathcal{E} and \mathcal{C} the expression and the condition contained within \mathcal{CR} .

When \mathcal{CR} contains no *LogicalConnector* objects, the evaluation of the criterion is straightforward : the result is equal to the boolean-evaluation of the tuple $(\mathcal{E}, \mathcal{C})$. This tuple is evaluated according to the concrete class involved, as explained in paragraphs 9.6.1 to 9.6.8

It is a bit more complex when criteria contain *LogicalConnectors*. Let us see how to proceed.

To begin with, let us consider only *Criterion* concrete objects:

As we saw in the previous paragraphs, criteria object are (with the help of *LogicalConnectors* object) recursive and hierarchical objects.

This hierarchical structure composing a complex criterion could be graphically represented as follows.

$$(\mathcal{E}_1, \mathcal{C}_1) \xrightarrow[\text{AND/OR}]{LC_1} (\mathcal{E}_2, \mathcal{C}_2) \xrightarrow[\text{AND/OR}]{LC_2} \dots (\mathcal{E}_i, \mathcal{C}_i) \xrightarrow[\text{AND/OR}]{LC_i} \dots (\mathcal{E}_{N-1}, \mathcal{C}_{N-1}) \xrightarrow[\text{AND/OR}]{LC_{N-1}} (\mathcal{E}_N, \mathcal{C}_N) \quad (8)$$

where the index 1, i and N are respectively for the root, the i and the leaf criterion composing the structure. The term LC_i denotes the *LogicalConnector* contained within the criterion \mathcal{CR}_i .

As we saw in paragraphs 9.6.1 to 9.6.8 every tuple $(\mathcal{E}_i, \mathcal{C}_i)$, $i = 1, \dots, N$ could be evaluated (according to the concrete object involved) and leads to a boolean value \mathcal{B}_i . Thus the expression (8) become

$$\mathcal{B}_1 \xrightarrow[\text{AND/OR}]{LC_1} \mathcal{B}_2 \xrightarrow[\text{AND/OR}]{LC_2} \dots \mathcal{B}_i \xrightarrow[\text{AND/OR}]{LC_i} \dots \mathcal{B}_{N-1} \xrightarrow[\text{AND/OR}]{LC_{N-1}} \mathcal{B}_N \quad (9)$$

This last is a classic sequential boolean expression. It is evaluated from left to right and the operator AND takes precedence over the OR operator.

Let us now consider *ParenthesisCriterion* criteria. A representation of such a criterion \mathcal{CR} could be the following:

$$\left\langle (\mathcal{E}, \mathcal{C}) \xrightarrow{LC} \mathcal{CR}_c \right\rangle_{\mathcal{CR}} \xrightarrow{ELC}, \quad (10)$$

where $\mathcal{E}, \mathcal{C}, LC, \mathcal{CR}_c$ are respectively the *Expression*, the condition, the *LogicalConnector* and the criterion contained within LC . The term ELC is the *ExternalLogicalConnector* of \mathcal{CR} .

The criterion structure contained within $\langle \cdot \rangle_{\mathcal{CR}}$ has the highest priority and has to be evaluate, before the *ExternalLogicalConnector* evaluation.

In the case where \mathcal{CR}_c is composed only of *Criterion* objects (so with no *ParenthesisCriterion*), the evaluation of the content of $\langle \cdot \rangle_{\mathcal{CR}}$ is performed as shown before in (8) and (9).

In the case where \mathcal{CR}_c contains at least one *ParenthesisCriterion*, one has to go deeper in the criterion structure to find the deepest criterion \mathcal{CR}_d such that $\langle \cdot \rangle_{\mathcal{CR}_d}$ contains only criteria of type *Criterion*. Thus one can simply evaluate the content of $\langle \cdot \rangle_{\mathcal{CR}_d}$ as already shown.

For illustrating how to proceed, let us consider the following complex-criterion structure:

$$\begin{aligned} & \left\langle (\mathcal{E}_1, \mathcal{C}_1) \xrightarrow{LC_1} (\mathcal{E}_2, \mathcal{C}_2) \right\rangle_{\mathcal{CR}_1} \xrightarrow{ELC_1} \dots \\ & \quad \left\langle (\mathcal{E}_{i-1}, \mathcal{C}_{i-1}) \xrightarrow{LC_{i-1}} \left\langle (\mathcal{E}_i, \mathcal{C}_i) \xrightarrow{LC_i} (\mathcal{E}_{i+1}, \mathcal{C}_{i+1}) \right\rangle_{\mathcal{CR}_i} \right\rangle_{\mathcal{CR}_{i-1}} \xrightarrow{ELC_{i-1}} \\ & \quad \dots \left\langle (\mathcal{E}_{N-1}, \mathcal{C}_{N-1}) \xrightarrow{LC_{N-1}} (\mathcal{E}_N, \mathcal{C}_N) \right\rangle_{\mathcal{CR}_{N-1}} \end{aligned} \quad (11)$$

From what we saw above, the expression (11) becomes

$$\begin{aligned} & \left\langle \mathcal{B}_1 \xrightarrow{LC_1} \mathcal{B}_2 \right\rangle_{\mathcal{CR}_1} \xrightarrow{ELC_1} \dots \\ & \quad \left\langle \mathcal{B}_{i-1} \xrightarrow{LC_{i-1}} \left\langle \mathcal{B}_i \xrightarrow{LC_i} \mathcal{B}_{i+1} \right\rangle_{\mathcal{CR}_i} \right\rangle_{\mathcal{CR}_{i-1}} \xrightarrow{ELC_{i-1}} \\ & \quad \dots \left\langle \mathcal{B}_{N-1} \xrightarrow{LC_{N-1}} \mathcal{B}_N \right\rangle_{\mathcal{CR}_{N-1}} \end{aligned} \quad (12)$$

and finally

$$\begin{aligned} & \left(\mathcal{B}_1 \xrightarrow[AND/OR]{LC_1} \mathcal{B}_2 \right) \xrightarrow[AND/OR]{ELC_1} \dots \\ & \quad \left(\mathcal{B}_{i-1} \xrightarrow[AND/OR]{LC_{i-1}} \left(\mathcal{B}_i \xrightarrow[AND/OR]{LC_i} \mathcal{B}_{i+1} \right) \right) \xrightarrow[AND/OR]{ELC_{i-1}} \\ & \quad \dots \left(\mathcal{B}_{N-1} \xrightarrow[AND/OR]{LC_{N-1}} \mathcal{B}_N \right). \end{aligned} \quad (13)$$

This last is a classical sequential boolean expression. It is evaluated from the left to the right. The sub-expression between the parenthesis must be evaluated with the highest priority and the operator AND takes precedence over the OR operator.

10 PDL and formal logic

We recall that PDL is a grammar and syntax framework for describing parameters and their constraints. Since the description is rigorous and unambiguous, PDL could verify if the instance of a given parameter is consistent with the provided description and related constraints. For example, consider the description

$$\begin{cases} p_1 \text{ is a Kelvin temperature} \\ \text{Always } p_1 > 0 \end{cases} . \quad (14)$$

According to the description, the PDL framework could automatically verify the validity of the parameter provided by the user. If he/she provides $p_1 = -3$, then this value will be rejected.

In any case PDL is not a formal-logic calculation tool. One could build the following description with no problem:

$$\begin{cases} p_1 \in \mathbb{R} \\ \text{Always } ((p_1 > 0) \text{ AND } (p_1 < 0)) \end{cases} . \quad (15)$$

PDL lacks the capabilities to perceive the logical contradiction and will work according to its rules. In this case any parameter p_1 provided by user will be rejected.

In other words **people providing descriptions of services must pay great attention to their contents.**

Remark: In further developments PDL will include a formal-logic module. This will permit finding contradictions inside the descriptions. Moreover this kind of module is required for implementing the automatic computation of *a priori* interoperability graphs

11 Description Examples

Examples of the descriptions defined by equation (1) and (2) are available respectively at the following links:

- Example 1 ;
- Example 2.