International
Virtual
Observatory
Alliance

PDL: The Parameter Description Language Version 1.0

IVOA Proposed Recommendation December 19, 2013

This version:
1.0: December 19, 2013 (SVN Revision 201)

Latest version:
N/A

Previous versions:
PDL: The Parameter Description Language, IVOA Working Draft, Version 1.0

Working Group:
Grid and Web Services

Editor(s):
Carlo Maria Zwölf

Authors:
Carlo Maria Zwölf
Paul Harrison
Julián Garrido
Jose Enrique Ruiz
Franck Le Petit

Abstract

This document discusses the definition of the *Parameter Description Language* (PDL). In this language parameters are described in a rigorous data model. With no loss of generality, we will represent this data model using XML.
It intends to be a expressive language for self-descriptive web services exposing the semantic nature of input and output parameters, as well as all necessary complex constraints. PDL is a step forward towards true web services interoperability.

1 Status of this document

This document has been produced by the Grid and Web Service Working Group. It follows the previous working draft.

Acknowledgements

We wish to thank the members of the IVOA Grid and Web Services working group for the discussions around PDL it has hosted during the Interop meetings (starting from Naples, May 2011). A special thanks are due to André Shaaff for his advice, useful discussions and feedback on every version of this work.

Contents

2 Preface

Before going into technical details and explanations about PDL, we would like to suggest what are the categories of users potentially interested in this description language and underline in what field PDL has a strategic impact.

PDL is particularly addressed to scientists or engineers

- wishing to expose their research codes (without limits or compromise on the complexity of the exposed code) online as public services,
- wishing to interconnect their codes into workflows.

We will continue this preface by a quick 'theoretical' overview of PDL. For a practice-oriented introduction, reader should refer to the annex paragraph 13.1.

The online code aspect - Usually, people who are about to publish their codes as online services are wondering if user will be able to correctly use the new services: the code may have many parameters (usually with obscure names, due to historical reasons). These parameters can be intercorrelated and/or have forbidden ranges, according to the configuration or validity domain of the implemented model. In other words the use of the code may be reserved to experts.
How can the service provider ensure that the users will be aware of all the subtleties (units, physical meaning, value ranges of parameters)? The natural answer is to provide a good code documentation to the community. However, our experience as service providers showed that rarely users read carefully the documentation. And even if they read it entirely, they are not safe from making gross and/or distraction errors. Two commons consequences of this situation are the abandonment of the service by users, after few inconclusive tests and the abandonment of the service by the provider him(her)self, buried by e-mails from users containing question whose answer are ... in the provided documentation.
PDL is a powerful tool for improving both the user and the provider experiences: it may be seen as a way for hardcoding all the subtleties and constraints of the code into the software components used by the provider for exposing the code and by users for interacting with it.
The fine expertise on the code to expose is fixed into the PDL description. The PDL software framework is able to automatically generate client interfaces (for assisting the user in interacting with the service) and checking algorithms (for verifying that the data submitted by users are compliant with the description). Moreover the softwares composing the PDL framework are generic elements which are automatically configured by the description into ad-hoc components (cf. paragraph 12 for further details and explanation about these concepts). The work for people wishing to expose code is indeed essentially restricted to redaction of the description. For these reasons PDL is particularly indicated for people wishing to expose their code but don't have much time or the technical skills for building web services.

The workflow aspect - Scientists or engineers wishing to integrate their code into workflow engines have to write ad hoc artifacts: (in the case of the *Taverna* engine) these could be *Java Beans*, *Shell* and *Python* artefacts. Normally one has to write a specific artefact for every code to integrate. This could be extremely time consuming and, from the technical point of view, this is not an 'out of the box' procedure for people starting using workflows engine.
PDL is indicated to facilitate the integration of code into workflow engines: the integration phase is reduced to the redaction of the description.
Moreover PDL introduce a new feature into the workflow domain: since every description embeds fine grained details and metadata on parameters (also with their related constraints), the physical sense (meaning and integrity) of a workflow could be automatically verified.

The PDL application domain is not limited only to online code or workflows. We are now going to detail all the technical aspects of this description grammar.

3 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,
- access to smaller sub-products images, spectra and/or data generated on the fly,
- the entry point to a database listing the results of complex and compute-intensive 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.

Interoperability with other services and the immediacy of use are two key factors in the success of a service: in general, a service will not be used by the community if users do not know how to call it, the inputs it needs, or what it does and how. However, other issues may have influence in the user decision e.g. Who has implemented it? Who is the service provider? Does it implement a well known technique? Is there a paper to support the research behind the service? Can it be used as a standalone application and can it be used together with other services? A new service will be more useful for some users if it can be released easily as an interactive and standalone application whereas for other users the interoperability with other services and applications is more important. This standard is focused on the needs of the second group, as the ease of the distribution of web services is the primary concern of service providers. Indeed, service description and interoperability are two key points for building efficient and useful ecosystem of services.

PDL aims to provide a solution to the problems of description and interoperability of services. With PDL, service providers will be able to share with users (either as humans or as computer systems) the knowledge of what the service does (and how). Moreover this new service will be immediately interactive and well integrated with other services.

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

3.1 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 pre-existing service description languages. The most well known for their high expression level and their wide use are the *W3C WSDL* and *WADL*.
Since both *WSDL* and *WADL* support *XML-schema*, one could include in these descriptions complex and highly specialized XML objects for expressing conditions and/or restrictions. However, the process for building these ad-hoc XML extension types is not standard[1]: a service provider could only describe, using the native standard feature of WADL or WSDL, primitive-typed parameters. It thus serves a roughly similar purpose as a method-signature in a programming language, with no possibility for defining restrictions, semantics and criteria to satisfy. PDL proposes a way for expressing these features in a unified way.

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

[1]For example, for expressing that a given parameter must be greater and smaller than arbitrary values, we could define a *bounded* type containing an *inf* field and a *sup* field. If another user defines a similar object calling these two fields *inf-bound* and *sub-bound*, the instances of these two types could not interoperate straightforwardly. The theory of types is not sufficient to ensure the interoperability of the object defined.

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

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

To our knowledge, no existing description language meets these exacting requirements of scientific services. This leads us naturally to work on a new solution and consider developing a new description language.

Remark: The PDL descriptions for the two examples above are provided respectively in paragraphs 13.2 and 13.3.

3.2 Interoperability issues

Nowadays, with the massive spread and popularity 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 the 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.

Remark: We will present further considerations on the workflows aspects in paragraph 3.4.2, once we have exposed some basic concepts about PDL in the following paragraph.

3.3 Astronomical and Astrophysical use-cases needing PDL's fine description capabilities

PDL was originally designed for meeting requirements coming from the community members wishing to expose their code online as public services. One of the difficulty they often mentioned is that online codes are often complex to use and users may do mistake with online simulations. For example, they could use them outside of their validity domain. The description of parameters with PDL allows to constrain those ones in validity domains, and so PDL answers this fear of the theorist community.

In order to build a grammar like PDL, we could go two ways: in the first we would have built a monolithic solution for meeting the vast majority of astronomical and astrophysical needs. In the other we would have to provide community with a flexible enough tool (modular and extensible) to fit the majority of use-cases: if the parameters (of a given service) are decomposed with the finest granularity, PDL is a good tool for performing *a priori verification*, notifying errors to user before submitting jobs to a server system. This has, for example, an immediate consequence on how we deal, in PDL, with sky coordinates: we don't have particular fields/entries for ascensions and declinations.

For us this parameters could be stored in *double* parameters. The associated unit will precise if the angle will be considered in degrees or radiant and the associated *SKOS* concepts (http://www.w3.org/TR/skos-reference/) will provide further information. If a service provider has to define particular conditions on the angular distance between two coordinates (asc_1, dec_1) and (asc_2, dec_2) (e.g. $|asc_1 - asc_2| + |dec_1 - dec_2| < \epsilon$) he/she may use the expression capabilities of PDL (cf. paragraph 9)

During the PDL development, close cooperation naturally born with the Workflow community. PDL indeed allow the real *scientific* interoperability (not only based on computer types) required by the Astronomic and Astrophysics workflow community.

The following sections of this document could seems complex at first reading. This is because we present all the features and the descriptive richness of PDL. Nevertheless this does not mean that all PDL descriptions are necessarily complex. They could be complex in case of services with many parameters linked by many constraints. But PDL description could be very simple in case of simple services. For example the PDL description associated with a common cone search service is very simple. It could be consulted at the following URL:

http://www.myexperiment.org/files/999/versions/4/
download/AMIGA-PDL-Description.xml.

3.4 A new Parameter Description Language: a unique solution 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 *interpreted* by human beings (we could say *understood* for the simpler description cases),
- could be parsed and handled by a computer,
- complex relations and constraints involving parameters could be formulated unambiguously. Indeed we would like to express
  - mathematical laws/formulas,
  - conditional relationships (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 object structure.

With PDL one could build a mathematical expression (respectively conditional sentences) assembling the base-element described in section 9 (resp. section 10).

If a particular expression (or condition) could not be expressed using the existing features, this modular grammar could be extended by introducing an ad hoc syntactic element into the object DM.

For describing the physical scientific concept or model behind a given parameter, the idea is to use *SKOS* concepts and, if more complexity is required by the use case, a richer ontology.

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 reasonably 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 (cf paragraph 13.4).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, it is possible to verify if the instance of a given parameter (i.e. the value of the parameter that a user sends 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 condition composed with PDL. By this we mean that one must replace the referenced parameters (in the PDL expressions/conditions) by the set of values provided to the service by the user. The replacement mechanisms will be explained in detail, case by case.

3.4.1 PDL in the IVOA architecture

Within the IVOA Architecture of figure 1, PDL is a VO standard for richly describing parameters with a fine grained granularity, allowing to introduce constraints and mathematical formulae.
If PDL describes the nature, the hierarchy of parameters and their constraints, **it does not describe** how this parameters are transmitted to a service, nor how these parameters will be processed by the described service. For example, PDL does not prescribe whether to transfer parameters through a SOAP envelope or through a REST post, nor what will be the phases that the submitted job will pass through. In the context of the IVOA, this means that the separation between PDL and UWS is clear and one can be used without the other without infringing any rules of those standards.
Indeed, PDL could be seen a supplementary layer, for explaining in a unified way the physical/computational meaning of every parameter, whereas UWS has only a description of the values of parameters.

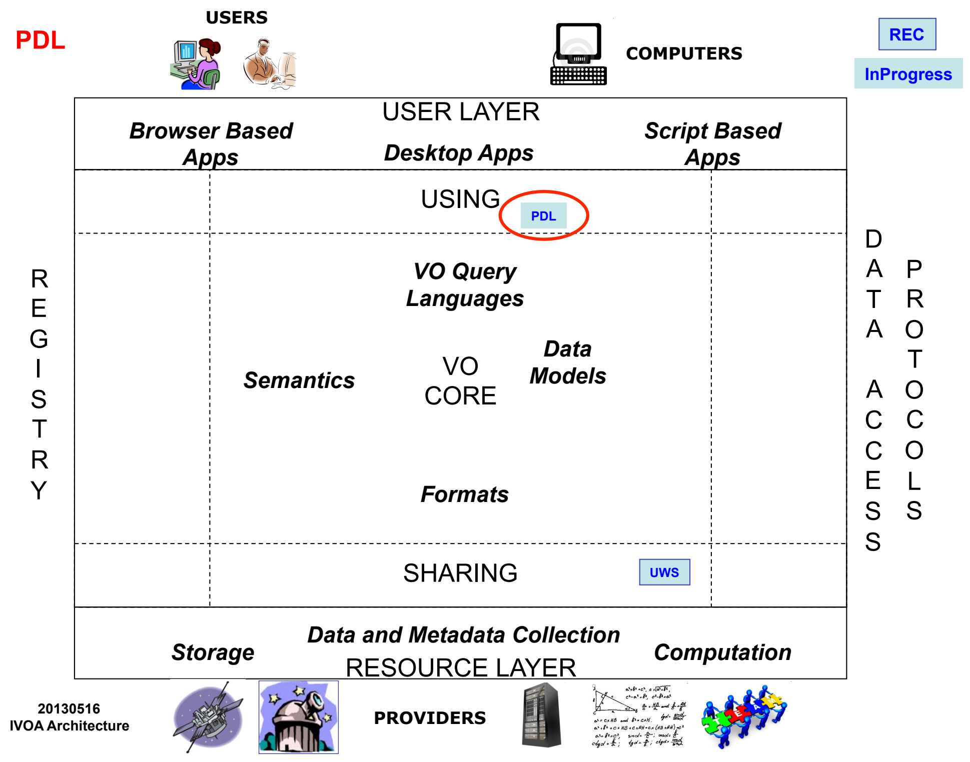


Figure 1: *The IVOA Architecture, with PDL highlighted. As pointed out in paragraph 3.4.1, the domains and scopes of PDL and UWS are well separated: one can be used without the other without infringing any rules of those standards. Of course they could work in synergy. In this case PDL could be seen a supplementary layer (explaining the physical/computational meaning of every parameter), whereas UWS has only a description of the values of parameters.*

PDL could be plugged as an additional layer to every existing IVOA service and is suitable for solving issues not covered by other IVOA standards and is particularly indicated for workflows.

3.4.2 Some consideration on PDL and Workflows

The orchestration of services define a Scientific Workflow, and services interoperability is key in the process of designing and building workflows. An important consideration in this process of orchestration is the control of parameters constraints at the moment of the workflow execution. Even if interoperability is assured at the phase of workflow design, a control at the execution phase has to be implemented by workflow engines as service clients. As we suggested in the remark of the previous paragraph, testing for valid parameters provided to a service could be automatically generated starting from the PDL description. This automation facility could be used to perform the verification both client side and server side:

- verifications made client-side will avoid sending the wrong set of parameters to a server, reducing the load on the latter,
- verifications server-side will avoid running jobs with wrong set of parameters. Indeed a server does not know if the job is sent by a client implementing the verifications or not. Therefore it must behave as if the data had never been checked.

Verification of non-standard errors (e.g. network issues) are out of the scope of PDL.

4 The Service object

The root element of the PDL description of a generic service is the object *Service* (see figure 2). This **must contain**

- A single *ServiceName*. This field is a String containing the name of the service.
- A *ServiceId*. This field is a String containing the IVOA id of the service. It is introduced for a future integration of PDL into the registries: each service in the registry will be marked with its own unique id.
- A *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 *Parameters* field which is a list of *SingleParameter* object types (cf. paragraph 5). 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.
- An *Inputs* field of type *ParameterGroup* (cf. paragraph 8). This object contains the detailed description (with constraints and conditions) of all the input parameters.
- An *Outputs* field of type *ParameterGroup*. This object contains the detailed description (with constraints and conditions) of all the output parameters.

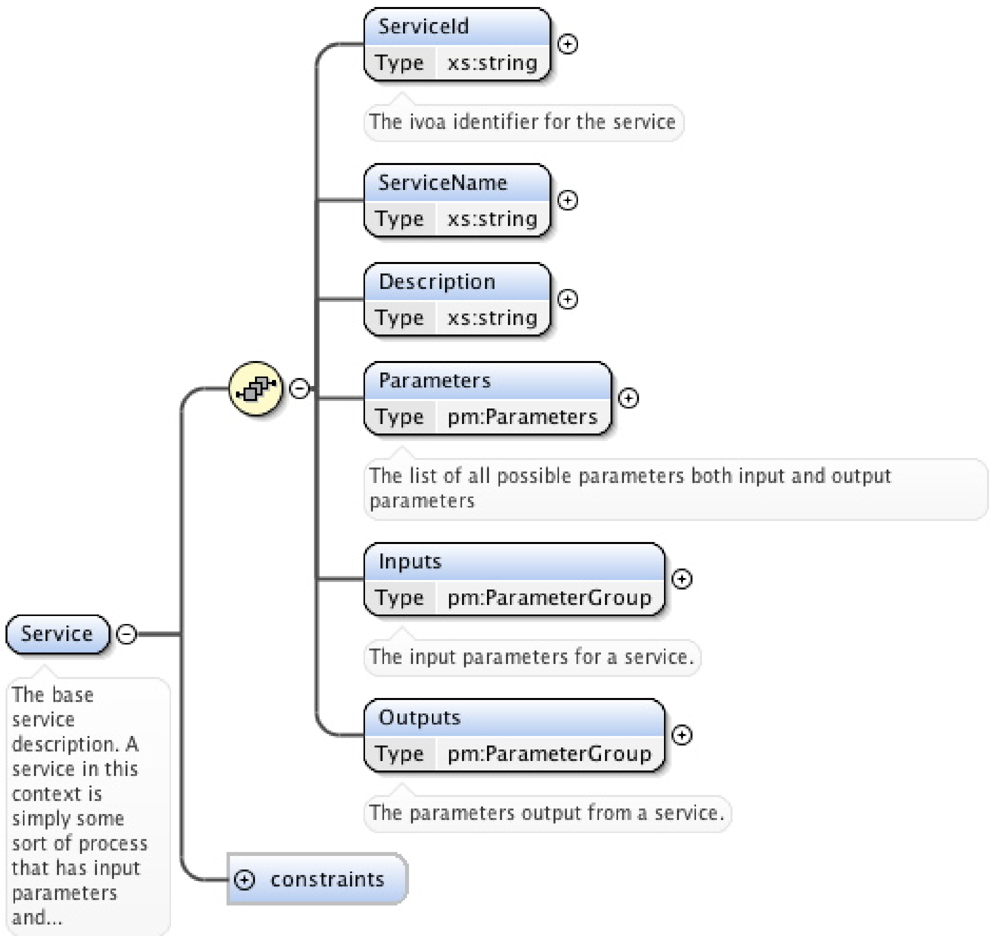

ServiceId
Type xs:string
The ivoa identifier for the service
ServiceName
Type xs:string
Description
Type xs:string
Parameters
Type pm:Parameters
The list of all possible parameters both input and output parameters
Inputs
Type pm:ParameterGroup
The input parameters for a service.
Outputs
Type pm:ParameterGroup
The parameters output from a service.
Service
The base service description. A service in this context is simply some sort of process that has input parameters and...
constraints

Figure 2: Graphical representation of the Service object

5 The SingleParameter Object

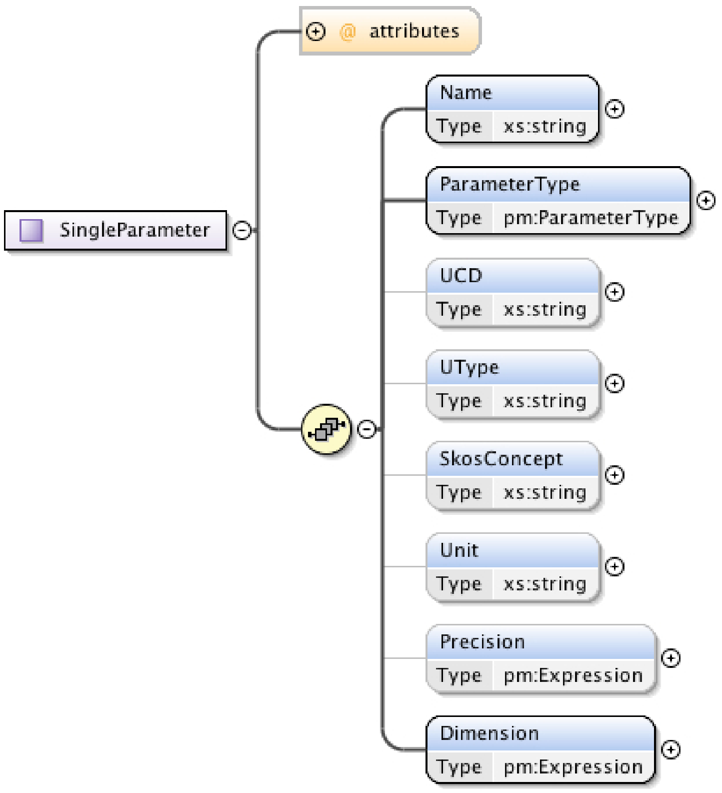


Figure 3: Graphical representation of the Parameter object

The *SingleParameter* object (see figure 3) is the core element for describing jobs. Every object of this type must be characterized by:

- A name, which is the Id of the parameter. In a given PDL description instance, two parameters cannot have the same name;
- A single parameter type, which explains the nature of the current parameter. The allowed types are : boolean, string, rational, complex, integer, real, date;
- A 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 9). The result of the expression that appears in this *SingleParameter*-field object **must be integer**.[2]

Remark on the vector aspect: It could seem unnecessarily complex to have the parameter dimension into an *Expression*. This feature has been introduced for meeting

[2]This is obvious, since this value corresponds to a vector size.

some particular needs: consider for example a service computing polynomial interpolations. Let the first parameter d (an integer) be the degree of the interpolation and the second parameter $\vec{p}$ be the vector containing the set of points to interpolate. For basic mathematical reasons, these two parameters are linked by the condition $(\text{size}(\vec{v})-1) = d$. By defining dimensions as *Expressions* we can easily include this kind of constraints into PDL descriptions.
Vectors in PDL are intended as one dimensional arrays of values. Further significations should be documented using the UCD, Utype or the Skos concept fields. Moreover, if one wish to define an *Expression* using the scalar product of two vectors (cf. paragraph 9) he/she has to pay attention that the involved vectors are expressed in the same orthonormal basis.

■

The 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 UCD : which is a text reference to an existing UCD for characterizing the parameter;
- a Utype : which is a reference to an existing Utype for characterizing the parameter (the reference is typically a text string);
- a Skos Concept (the reference is typically a text string containing the valid URL of a Skos concept).
- a Unit (which is a string reference to a valid VOUnits element).
- a precision. This field must be specified only for parameter types where the concept of precision makes sense. It has indeed no meaning for integer, rational or string. It is valid, for instance, for a real type. To understand 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 9). 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.

6 The ParameterReference object

This object, as it name suggests, is used to reference an existing parameter defined in the *Service* context (cf. paragraph 4). It contains only an attribute *ParameterName* of type String which must corresponds to the *Name* field of an existing *SingleParameter* (cf. paragraph 5).

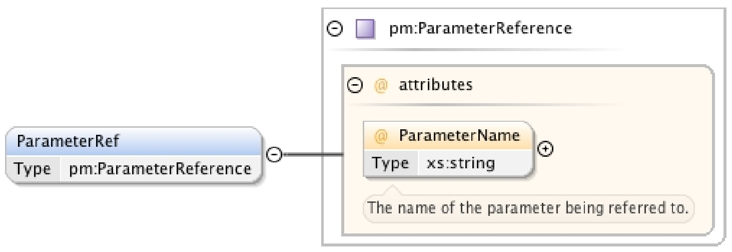

pm:ParameterReference
attributes
ParameterName
Type xs:string
The name of the parameter being referred to.
ParameterRef
Type pm:ParameterReference

Figure 4: Graphical representation of the Parameter Reference object

7 The ParameterType object

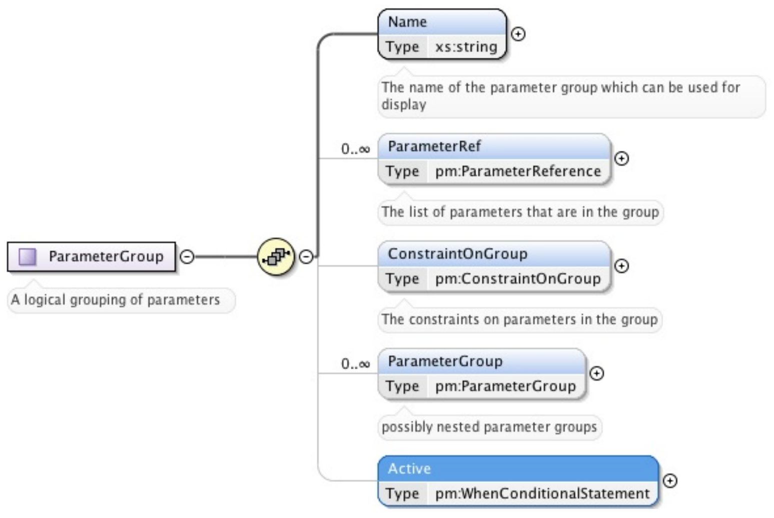
This object is used to explain the type of a parameter (cf. paragraph 5) or an expression (cf. paragraph 9.2). The allowed types are :

- Boolean. The allowed values for parameters of this type are *true* / *false*, non case sensitive.
- String. Any String (UTF8 encoding recommend).
- Integer. Numbers (positive and negatives) composed of [0-9] characters.
- Real. Two formats are allowed for parameters of this type:
  - numbers (positives and negatives) composed of [0-9] characters, with dot as decimal separator,
  - scientific notation: number composed of [0-9] characters, with dot as decimal separator, followed by the *E* character (non case sensitive), followed by an integer.
- Date. Parameters of this type are dates in ISO8601 format.

8 The ParameterGroup object

The *ParameterGroup* object (see figure 5) is used for grouping parameters according to a criterion of relevancy arbitrarily chosen by service provider (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 10.1).
This object **must contain** a single Name. This name is a String and is the identification label of the ParameterGroup, and two groups cannot have the same Name in a given PDL description instance.
Optional fields are

- the references to the parameters (cf. paragraph 6) one want to include into the group;



Name
Type xs:string
The name of the parameter group which can be used for display
0..∞
ParameterRef
Type pm:ParameterReference
The list of parameters that are in the group
ParameterGroup
A logical grouping of parameters
ConstraintOnGroup
Type pm:ConstraintOnGroup
The constraints on parameters in the group
0..∞
ParameterGroup
Type pm:ParameterGroup
possibly nested parameter groups
Active
Type pm:WhenConditionalStatement

Figure 5: Graphical representation of the ParameterGroup object

- a *ConstraintOnGroup* object (cf. paragraph 10.1). This object is used for expressing the complex relations and constraints involving parameters.
- an *Active* field of type *WhenConditionalStatement* (cf. paragraph 10.2.3). If there is no *Active* element the group will be considered active by default (e.g. in case of a graphical representation it will be displayed). Otherwise, the activations depends on the result of the evaluation of the Criterion contained into the When conditional statement (cf. paragraphs 10.2.3 and 10.7).
- the *ParameterGroup* object 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: 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.

9 The Expression Objects

The *Expression* is the most versatile component of the PDL. It occurs almost everywhere: in defining fields for *SingleParameters* (cf. paragraph 5) 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).

9.1 The AtomicParameter expression

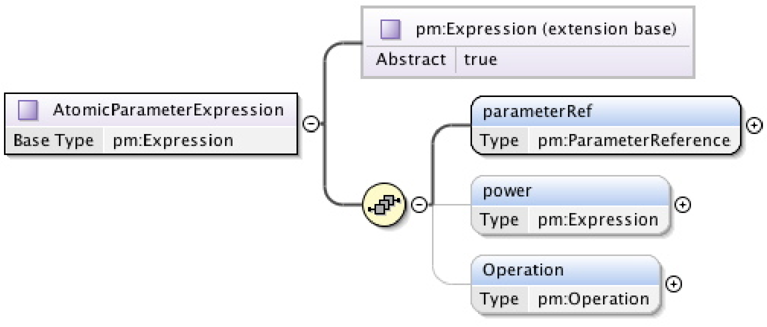


Figure 6: Graphical representation of the AtomicParameter expression object

The *AtomicParameterExpression* (extending *Expression*, see figure 6) 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 :

- A **numerical** power *Expression* object;
- An *Operation* object (cf. paragraph 9.4).

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

$$\underbrace{p^{exp}\underbrace{\overbrace{\begin{pmatrix} + \\ - \\ * \\ \cdot \\ \div \end{pmatrix}}^{\text{Operation type}}\overbrace{(\text{AnotherExpression})}^{\text{expression contained in operation}}}_{\text{Operation object}}}_{\text{Atomic Parameter Expression}} \tag{3}$$

To evaluate a given *AtomicParameterExpression*, one proceeds as follows: Let d_p, d_{exp} 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 $\forall\ i = 1, ..., d_p$ 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, $\forall\ i = 1, ..., d_p$ 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. We recall 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 9.4, by setting there $b = ep$.

9.2 The AtomicConstant expression

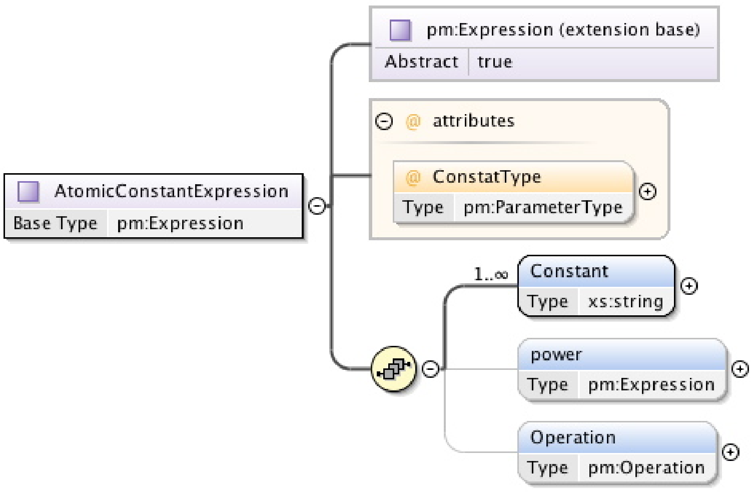


Figure 7: Graphical representation of the AtomicParameter expression object

The *AtomicConstantExpression* (extending *Expression*, see figure 7) 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 single 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 $d_c > 1$.
- An attribute *ConstantType* of type *ParameterType* (cf. paragraph 7) which defines the nature of the constant expression. The allowed types are the same as in the field *parameterType* of the object *SingleParameter*.

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 **numerical** power *Expression* object;
- An *operation* object (cf. paragraph 9.4).

Let s_i $(i = 1, ..., d_c)$ 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:

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

To evaluate a given *atomicConstantExpression*, one proceeds as follows: let d_c, d_{exp} be respectively the dimension of the vector constant (d_c is equal to one in case of scalar constant), 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 $(s_1, ..., s_{d_c})^{exp}$ is a d_c size vector expression and $\forall i = 1, ..., d_c$ 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, $\forall i = 1, ..., d_c$ 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 9.4, by substituting there $b = ep$.

9.3 The parenthesisContent expression

The *parenthesisContent* object (extending *Expression*, see 8) is used to explicitly denote precedence by grouping the expressions that should be evaluated first. This object **must contain** a single **numerical** object *Expression* (referred to hereafter as exp_1).
Optional fields are

- A **numerical** power *expression* object (referred to hereafter as exp_2);

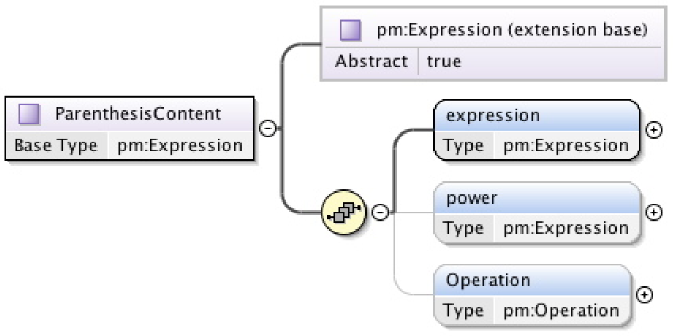


Figure 8: Graphical representation of theParenthesisContent expression object

- An *Operation* object (cf. paragraph 9.4).

This composite object could be presented as follows:

$$\underbrace{\underbrace{(exp_1)}_{\text{Priority term}}^{exp_2} \underbrace{\overbrace{\begin{pmatrix}+\\-\\*\\\cdot\\\div\end{pmatrix}}^{\text{Operation type}} \overbrace{(\text{AnotherExpression})}^{\text{expression contained in operation}}}_{\text{Operation object}}}_{\text{Parenthesis Expression}} \tag{5}$$

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 9.1 (after the equation (3)) by substituting $p = exp_1$ and $exp = exp_2$.

9.4 The Operation object

The *Operation* object (see figure 9) is used for expressing operations involving two **numerical** expressions. This object **must contain**:

- an *operationType* attribute. 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 $+, -, *, \cdot, \div$.
- an *Expression* object.

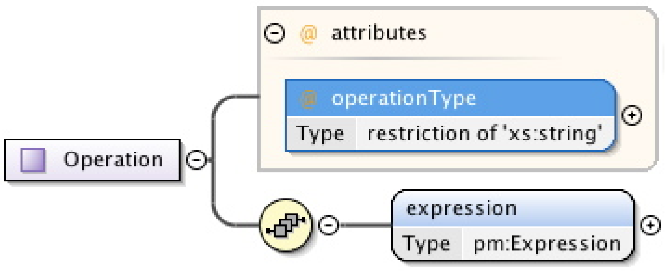


Figure 9: Graphical representation of Operation object

$$\underbrace{\overbrace{\begin{pmatrix} + \\ - \\ * \\ \cdot \\ \div \end{pmatrix}}^{\text{Operation type}} \overbrace{(\text{ContaindedExpression})}^{\text{expression contained in operation}}}_{\text{Operation object}} \tag{6}$$

The *Operation* object is always contained within a **numerical** *Expression* (cf. paragraph 9) and could not exist alone. Let a be the result of the evaluation of the expression object containing the operation[3] let b 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:

- $d_a = d_b$ and operation type (i.e. the operator) $op \in \{+, -, *, \div\}$. In this case $a\,op\,b$ is a vector expression of size d_a and $\forall\ i = 1, ..., d_a$ the i component of this vector is equal to $(a_i\,op\,b_i)$ (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 $a\,op\,b$ is a vector expression of size d_a and $\forall\ i = 1, ..., d_a$ the i component of this vector is equal to $(a_i\,op\,b)$.
- Or $d_a = 1$ and operation type (i.e. the operator) $op \in \{+, -, *, \div\}$. This case in symmetric to the previous one.

[3]this came from the evaluation of parameterRef field in case of an *AtomicParameterExpression* (cf. paragraph 9), from the evaluation of constant field in the case of a *AtomicConstantExpression* (cf. paragraph 9.2), from the evaluation of the Expression field in case of an *parenthesisContentExpression* (cf. paragraph 9.3) and from the evaluation of the Function object in case of a *FunctionExpression* (cf. par. 9.7)

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

9.5 The FunctionType object

This object is used for specifying the mathematical nature of the function contained within a *Function* object (cf. paragraph 9.6). 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 9.6 it is explained how these different function types are used and handled.

9.6 The Function object

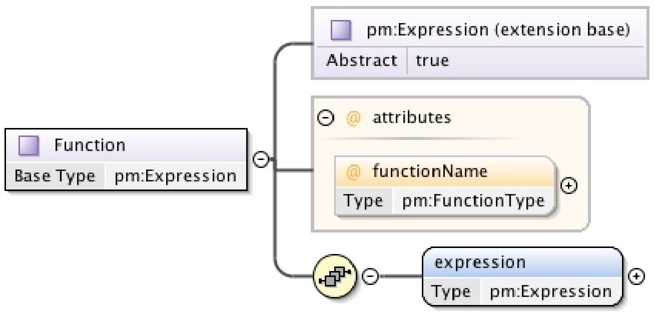


Figure 10: Graphical representation of Function object

The *Function* object (extending *expression*, see figure 10) is used for expressing a mathematical function on expressions. This object **must contain**

- A *functionName* attribute (of type *functionType* (cf. paragraph 9.5)) which specifies the nature of the function.
- An *Expression* object (which is the function argument).

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

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

- Or f =size. In this case the result is the scalar integer value d_{arg}.

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

9.7 The FunctionExpression object

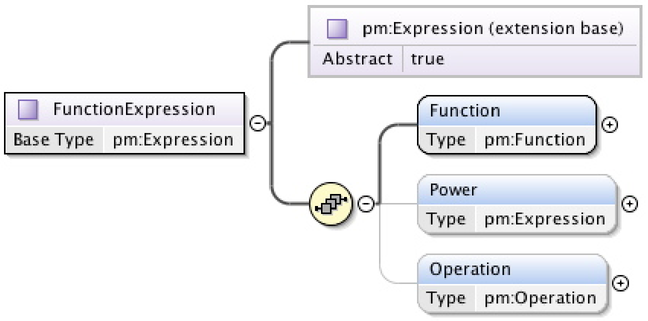


Figure 11: Graphical representation of FunctionExpression object

The *FunctionExpression* object (extending *Expression*, see figure 11) is used for building mathematical expressions involving functions.
This object **must contains** a single *Function* object (cf. paragraph 9.6).
Optional fields, valid only for numerical types, are :

- A **numerical** power *Expression* object;
- An *Operation* object (cf. paragraph 9.4).

This composite object could be presented as follows:

$$\underbrace{\underbrace{(\text{function})}_{\text{Function object}}^{exp} \underbrace{\overbrace{\begin{pmatrix} + \\ - \\ * \\ \cdot \\ \div \end{pmatrix}}^{\text{Operation type}} \overbrace{(\text{AnotherExpression})}^{\text{expression contained in operation}}}_{\text{Operation object}}}_{\text{FunctionExpression Object}} \tag{7}$$

In order to evaluate this object expression, one proceed as follows: first one evaluate the funtion expression as explained in paragraph 9.6. Then one proceed exactly as in paragraph 9.1 (after the equation (3)) by taking p =function.

10 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.

10.1 The ConstraintOnGroup Object

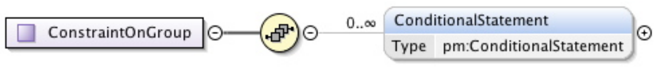


Figure 12: Graphical representation of ConstraintOnGroup object

The *ConstraintOnGroup* object (see figure 12) 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 10.2, for expressing the complex relations and constraints involving parameters.

10.2 The ConditionalStatement object

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

10.2.1 The AlwaysConditionalStatement

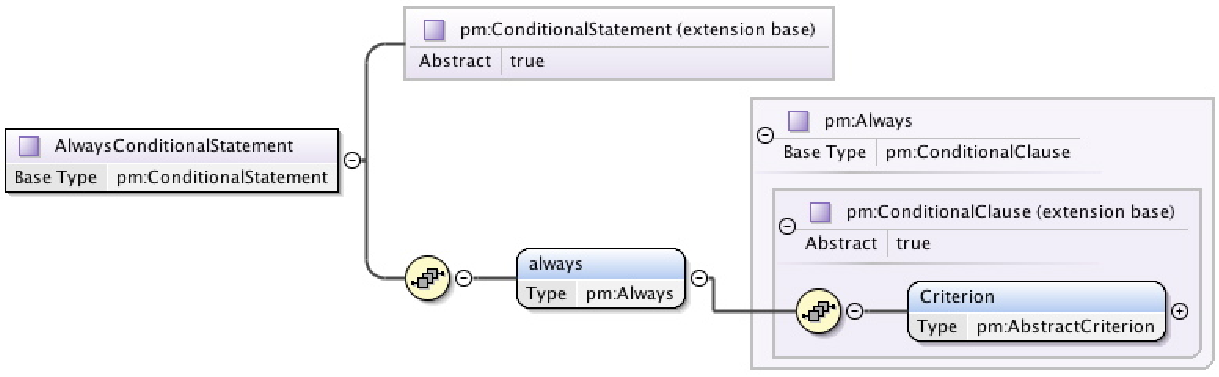

pm:ConditionalStatement (extension base)
Abstract true
AlwaysConditionalStatement
Base Type pm:ConditionalStatement
pm:Always
Base Type pm:ConditionalClause
pm:ConditionalClause (extension base)
Abstract true
always
Type pm:Always
Criterion
Type pm:AbstractCriterion

Figure 13: Graphical representation of AlwaysConditionalStatement object

This object (see figure 13), as it name suggests, is used for expressing statement that must always be valid. It **must contain** a single *Always* object (which extends

ConditionalClause, cf. paragraph 10.3).

10.2.2 The IfThenConditionalStatement

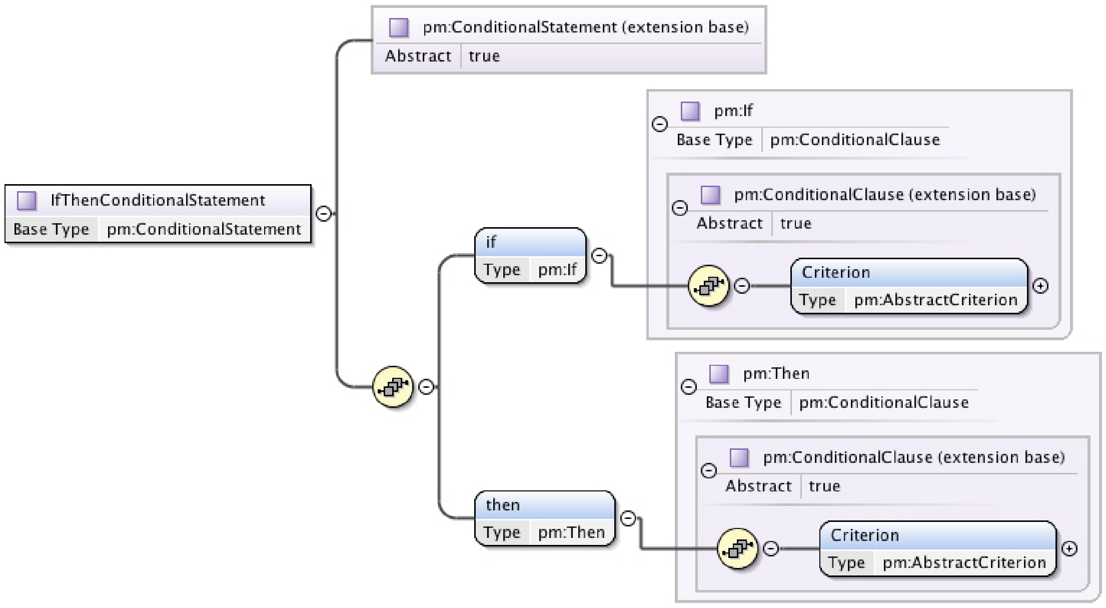

pm:ConditionalStatement (extension base)
Abstract true
IfThenConditionalStatement
Base Type pm:ConditionalStatement
if
Type pm:If
pm:If
Base Type pm:ConditionalClause
pm:ConditionalClause (extension base)
Abstract true
Criterion
Type pm:AbstractCriterion
then
Type pm:Then
pm:Then
Base Type pm:ConditionalClause
pm:ConditionalClause (extension base)
Abstract true
Criterion
Type pm:AbstractCriterion

Figure 14: Graphical representation of IfThenConditionalStatement object

This object (see figure 14), as it name suggests, is used for expressing statements that are valid only if a previous condition is verified. It **must contain**:

- an *If* object (which extends *ConditionalClause*, cf. paragraph 10.3).
- a *Then* object (which extends *ConditionalClause*, cf. paragraph 10.3).

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

10.2.3 The WhenConditionalStatement object

The when conditional statement is valid when the enclosed *When* conditional clause evaluates to true (cf. paragraph 10.7). It contains a unique field of *When* type (cf. paragraph 10.3). It was introduced for the purpose of dealing with the case of activating a ParameterGroup (cf paragraph 8): Thus When has the advantage of having a restricted form of ConditionalStatement that could have no *side effects* in the Then part.

10.3 The ConditionalClause object

The *ConditionalClause* object (see figure 16) is abstract. It **must contain** a single *Criterion* object of type *AbstractCriterion* (cf. paragraph 10.4).

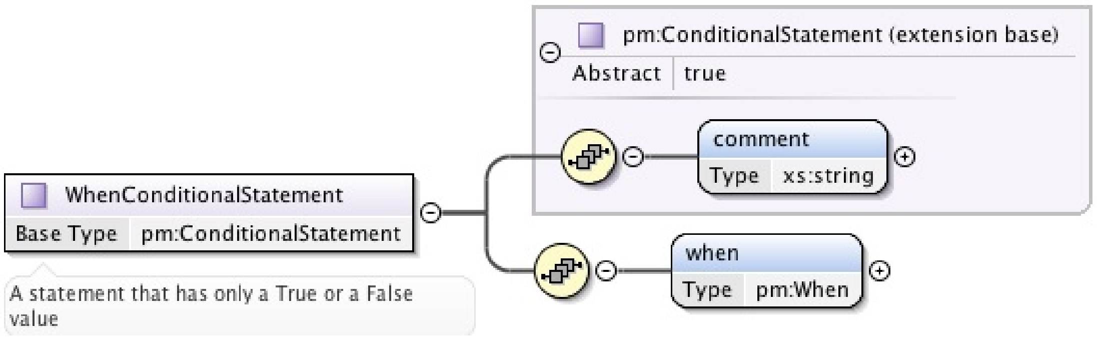

pm:ConditionalStatement (extension base)
Abstract true
comment
Type xs:string
WhenConditionalStatement
Base Type pm:ConditionalStatement
A statement that has only a True or a False value
when
Type pm:When

Figure 15: Graphical representation of a WhenConditionalStatement object

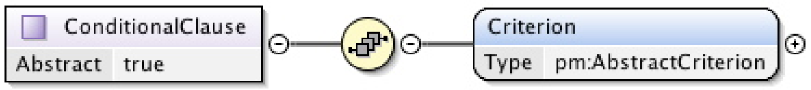


Figure 16: Graphical representation of ConditionalClause object

The four concrete objects extending the abstract *ConditionalClause* are (see figure 17):

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

The Criterion contained within a *Always* object must always evaluates to *true* (we will hereafter say it is valid, cf paragraph 10.2.1). With other words, this means that *it is good* only when the evaluation of the criterion contained into the *Always* object' evaluates to *true*. What if it is not good? It is wrong. *Wrong* evaluation is typically cached for notifying errors to users.

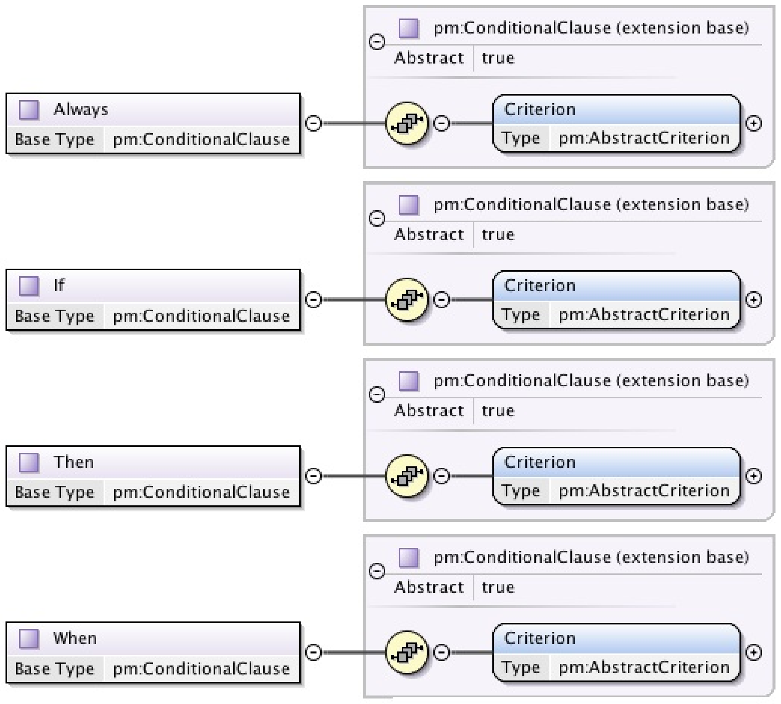
The Criterion contained within a *When* object will be valid only when the enclosed Expression evaluates to True (cf. paragraphs 10.2.3 and 10.7).

The *If* and *Then* objects work as a tuple by composing the *IfThenConditionalStatement* (cf. paragraph 10.2.2).

10.4 The AbstractCriterion object

The objects extending *AbstractCriterion* (see figure 18) are essentials for building *ConditionalStatemets* (cf. paragraph 10.2) since they are contained within the *Always*, *If* and *Then* objects (cf. paragraph 10.3). An *AbstractCriterion* object **must contain**:

- an *Expression* object (cf. paragraph 9);



Always
Base Type pm:ConditionalClause
If
Base Type pm:ConditionalClause
Then
Base Type pm:ConditionalClause
When
Base Type pm:ConditionalClause
pm:ConditionalClause (extension base)
Abstract true
Criterion
Type pm:AbstractCriterion

Figure 17: Graphical representation of Always, If and Then clauses

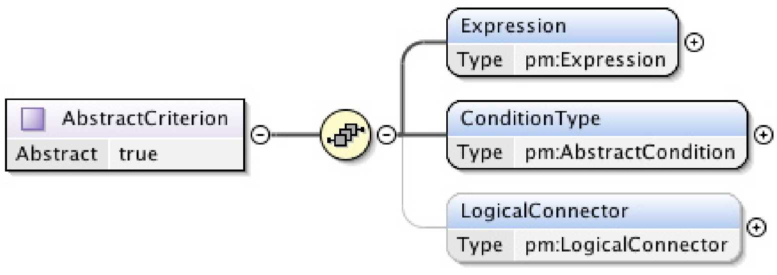


Figure 18: Graphical representation of AbstractCriterion object

- a *ConditionType* which is an object of type *AbstractCondition* (cf. paragraph 10.6). This object specify which condition must be satisfied by the previous *Expression*.

An optional field is the single *LogicalConnector* object (cf. paragraph 10.5) used for building logical expressions.
The two concrete objects extending *AbstractCriterion* are *Criterion* and *ParenthesisCriterion*. The latter of these two objects allows for assigning priority when interpreting and linking the criteria (cf. paragraph 10.7).

10.4.1 The Criterion object

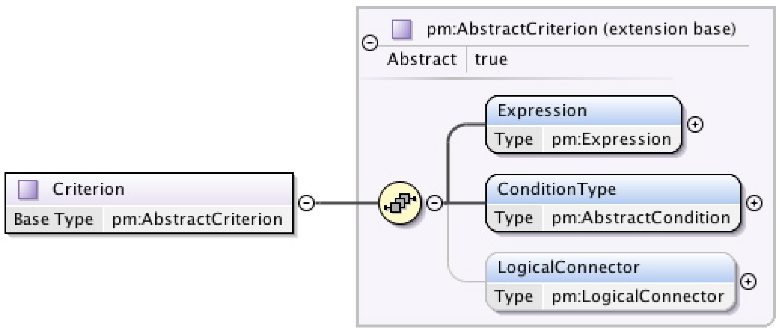


Figure 19: Graphical representation of Criterion object

This object (see figure 19) extends the *AbstractCriterion* without specializing it. It is indeed just a concrete version of the abstract type.

10.4.2 The ParenthesisCriterion object

This object (see figure 20) 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 an *ExternalLogicalConnector* object of type *LogicalConnector*. It is used for linking other criteria, out of the priority perimeter defined by the parenthesis (cf. paragraph 10.7).

10.5 The LogicalConnector object

The *LogicalConnector* object (see figure 21) is used for building complex logical expressions. It is an abstract object and it **must** contain a Criterion of type *AbstractCriterion* (cf. paragraph 10.4).
The two concrete objects extending *LogicalConnector* are:

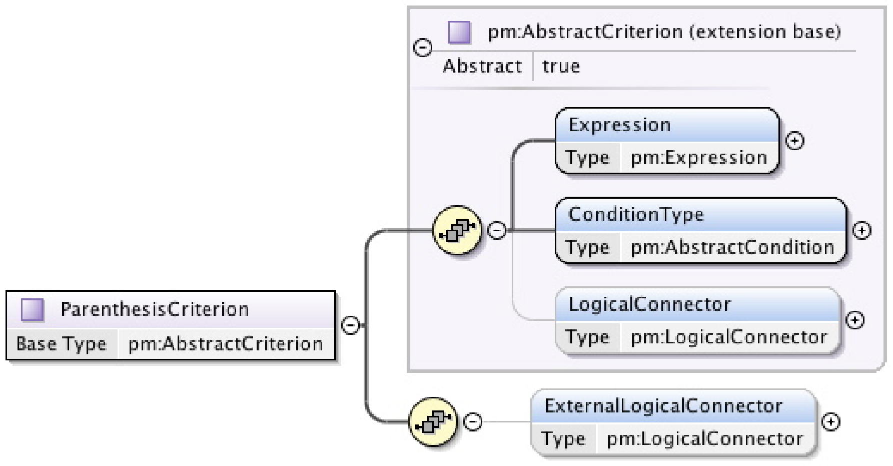


Figure 20: Graphical representation of ParenthesisCriterion object

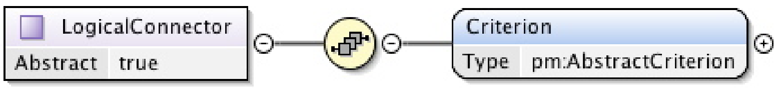


Figure 21: Graphical representation of LogicalConnector object

- 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.

10.6 The AbstractCondition object

AbstractCondition is an abstract object. The objects extending it always belong to an *AbstractCriterion* (cf. 10.4). 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.

10.6.1 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

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

a TRUE boolean value. Thus, in the case $\mathcal{CR}$ has no *LogicalConnector*, the criterion is true.

10.6.2 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.

10.6.3 The BelongToSet condition

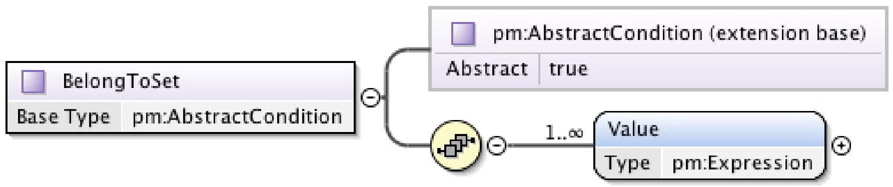

pm:AbstractCondition (extension base)
Abstract true
BelongToSet
Base Type pm:AbstractCondition
1..∞
Value
Type pm:Expression

Figure 22: Graphical representation of BelongToSet object

This object (see figure 22) 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 $\mathcal{E}_s^i =_t \mathcal{E}^i$, $\forall i = 1, ..., d_{\mathcal{E}}$, 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 $\mathcal{E}$.

10.6.4 The ValueLargerThan object

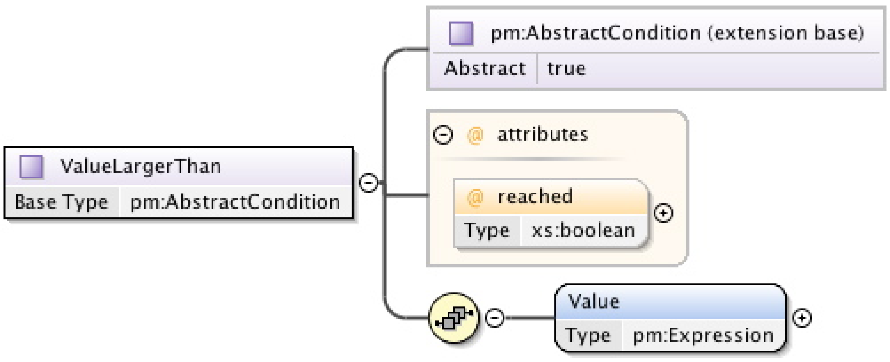


Figure 23: Graphical representation of ValueLargerThan object

This object (see figure 23) 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 **numerical** *Expression* $\mathcal{E}_c$.
- a *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.

10.6.5 The ValueSmallerThan object

This object (see figure 24) 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 **numerical** *Expression* $\mathcal{E}_c$.
- a *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 $\mathcal{E}_c$.

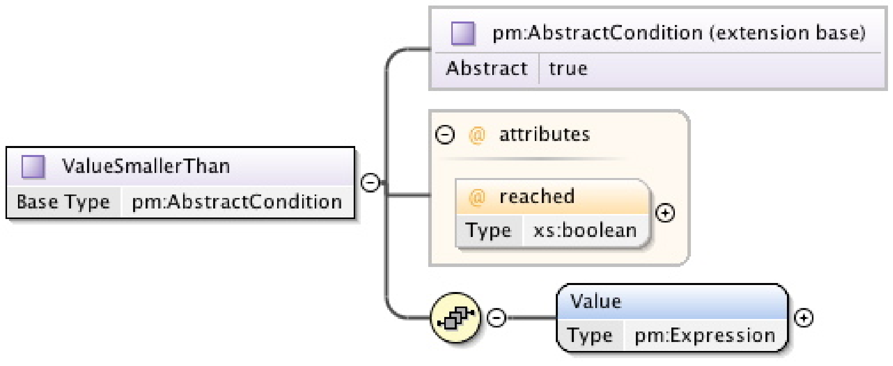


Figure 24: Graphical representation of ValueSmallerThan object

10.6.6 The ValueInRange object

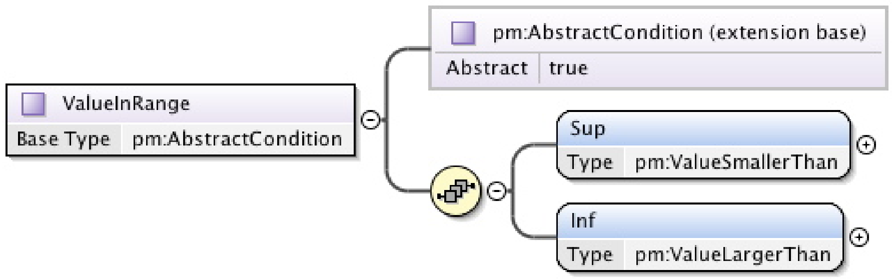


Figure 25: Graphical representation of ValueInRange object

This object (see figure 25) 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 using the *ValueLargerThan* *ValueSmallerThan* objects. Indeed, the *ValueInRange* object **must contain**:

- a *ValueLargerThan* object,
- a *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 ($\mathcal{E}$, *ValueSmallerThan*) and ($\mathcal{E}$, *ValueLargerThan*) lead to TRUE boolean values.

10.6.7 The ValueDifferentFrom object

This object (see figure 26) is used for specifying that the expression $\mathcal{E}$ must be different from a given value. It **must contain** an *Expression* $\mathcal{E}_c$.
In order to be compared, the two expressions $\mathcal{E}$ and $\mathcal{E}_c$ must have the same type. The

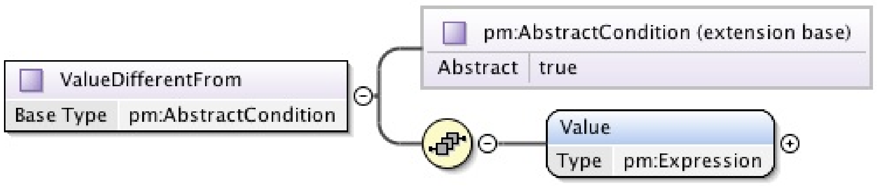

ValueDifferentFrom
Base Type pm:AbstractCondition
pm:AbstractCondition (extension base)
Abstract true
Value
Type pm:Expression

Figure 26: Graphical representation of ValueDifferentFrom object

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 10.6.3 (in the second point of the list).

10.6.8 The DefaultValue object

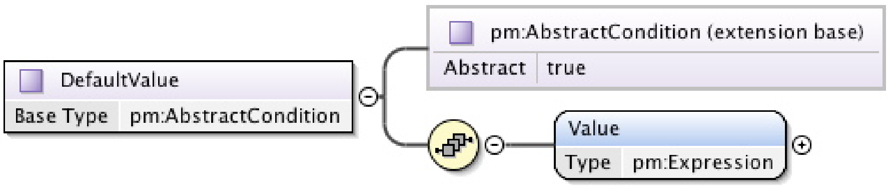

DefaultValue
Base Type pm:AbstractCondition
pm:AbstractCondition (extension base)
Abstract true
Value
Type pm:Expression

Figure 27: Graphical representation of DefaultValue object

This object (see figure 27) is used for specifying the default value of a parameter. It **must contain** an *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. 9.1)
- 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 10.3).

10.7 Evaluating and interpreting criteria objects

The evaluation of the criterion type objects (cf. paragraph 10.4) always leads to a boolean value (the only exception is what we saw in paragraph 10.6.8, where the criterion contains a *DefaultValue* condition).
We use hereafter the same notation introduced in 10.6: 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 10.6.1 to 10.6.8
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.

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

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 10.6.1 to 10.6.8 every tuple $(\mathcal{E}_i,\mathcal{C}_i)$, $i = 1,..,N$ could be evaluated (according to the concrete object involved) and leads to a boolean value $\mathcal{B}_i$. Thus the expression (8) become

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

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:

$$\Big\langle (\mathcal{E},\mathcal{C}) \xrightarrow{LC} \mathcal{CR}_c \Big\rangle_{\mathcal{CR}} \xrightarrow{ELC}, \tag{10}$$

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:

$$\Big\langle(\mathcal{E}_1,\mathcal{C}_1)\xrightarrow{LC_1}(\mathcal{E}_2,\mathcal{C}_2)\Big\rangle_{\mathcal{CR}_1}\xrightarrow{ELC_1}\cdots \\ \Big\langle(\mathcal{E}_{i-1},\mathcal{C}_{i-1})\xrightarrow{LC_{i-1}}\Big\langle(\mathcal{E}_i,\mathcal{C}_i)\xrightarrow{LC_i}(\mathcal{E}_{i+1},\mathcal{C}_{i+1})\Big\rangle_{\mathcal{CR}_i}\Big\rangle_{\mathcal{CR}_{i-1}}\xrightarrow{ELC_{i-1}} \\ \cdots\Big\langle(\mathcal{E}_{N-1},\mathcal{C}_{N-1})\xrightarrow{LC_{N-1}}(\mathcal{E}_N,\mathcal{C}_N)\Big\rangle_{\mathcal{CR}_{N-1}} \tag{11}$$

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

$$\Big\langle\mathcal{B}_1\xrightarrow{LC_1}\mathcal{B}_2\Big\rangle_{\mathcal{CR}_1}\xrightarrow{ELC_1}\cdots \\ \Big\langle\mathcal{B}_{i-1}\xrightarrow{LC_{i-1}}\Big\langle\mathcal{B}_i\xrightarrow{LC_i}\mathcal{B}_{i+1}\Big\rangle_{\mathcal{CR}_i}\Big\rangle_{\mathcal{CR}_{i-1}}\xrightarrow{ELC_{i-1}} \\ \cdots\Big\langle\mathcal{B}_{N-1}\xrightarrow{LC_{N-1}}\mathcal{B}_N\Big\rangle_{\mathcal{CR}_{N-1}} \tag{12}$$

and finally

$$\Big(\mathcal{B}_1\xrightarrow[\text{AND/OR}]{LC_1}\mathcal{B}_2\Big)\xrightarrow[\text{AND/OR}]{ELC_1}\cdots \\ \Big(\mathcal{B}_{i-1}\xrightarrow[\text{AND/OR}]{LC_{i-1}}\Big(\mathcal{B}_i\xrightarrow[\text{AND/OR}]{LC_i}\mathcal{B}_{i+1}\Big)\Big)\xrightarrow[\text{AND/OR}]{ELC_{i-1}} \\ \cdots\Big(\mathcal{B}_{N-1}\xrightarrow[\text{AND/OR}]{LC_{N-1}}\mathcal{B}_N\Big). \tag{13}$$

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.

11 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

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

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:

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

The PDL language grammar is not a tool with capabilities to perceive logical contradictions which may be contained within statements. This kind of consideration is outside of the scope of the present standard. For this reason, a validation system for PDL definitions is not required to implement the detection of contradictions, but may do if the services within its description realm make this feasible. The current PDL reference implementation does not perform such contradiction detection and thus any parameter p_1 provided by user would be rejected for this example.
In other words **people providing descriptions of services must pay great attention to their contents.**

12 Remarks for software components implementing PDL

Throughout this document we have described PDL as a *grammar*. If we consider it just as a grammar, then a specific description should be considered as an implementation.
We remember that, since a PDL description is detailed, it is a priori possible to write once for all generic software components. These components will be automatically *configured* by a PDL description thus becoming *ad hoc* implementation software for the described service. Moreover checking algorithms could also be generated automatically starting from a description instance. In our implementations we wanted to check practically that these concepts implied in the definition of PDL really works. The development of operational services (as the Paris-Durham shock code) also permits to ensure the coherence of the core grammar and to verify if the PDL's capabilities could meet the description needs of state of the art simulation codes.

At the present (Fall 2013) four software elements are implemented around PDL:

- the standalone dynamic client. It embeds the automatic generation of the verification layer (Google code repository at `https://code.google.com/p/vo-param/`). This development shows that a PDL description instance can be used for generating the checking algorithms and for generating a client with a dynamic-intelligent behavior helping the user in interacting with a service. This client could be used for interacting with services exposed using different job systems;
- a server for exposing any exiting code as a web services. It embeds the verification layer. This development was done for showing that a PDL description instance can be used for generating the *ad hoc* server, exposing the described service. A particular feature of this server is that it can generates grids of model starting from a single job submission, which indicates ranges for parameters (GitHub repository at `https://github.com/cmzwolf`);

- the Taverna Plugin. From one point of view this plugin could be seen as an alternate client to the standalone one. From another point of view it is strongly oriented towards general physical and scientific interoperability (discussed in paragraph 3.2) since it uses PDL description for validating the "chaining" of jobs composing a workflow. As the dynamic client, the Taverna plugin can be used for interacting with services exposed different job systems (GitHub repository for stable version at `https://github.com/wf4ever/astrotaverna`).
- the description editor, for editing PDL description from a Graphical User Interface. Since the key point for using PDL and take advantage of the software tools we have just described is a PDL description, we decided to provide the community with a tool for easily composing PDL description. In some sense this is the entry-point of the PDL software farmework (google code repository at `https://code.google.com/p/pdl-editor/`).

All these developments validate the concepts of automatic generation of algorithms and the possibility of configuring, with highly specialized individual behaviors, generic software components. This is very important since it reduces drastically the development time for building services based on the PDL grammar. This is essential in a scientific context where only few scientists have access to software engineer for their IVOA developments.

In further developments, PDL-client implementations will include a formal-logic module. This will permit finding contradictions inside the descriptions.
Such a module will also be required for implementing the automatic computation of *a priori interoperability graphs*. It will also permit checking interoperability in terms of semantic annotations: for example, let A be the concept that describes an input parameter of a service $\mathcal{S}$ and B the concept that describes an output parameter of a service $\mathcal{S}'$. If A and B are the same concept, then both services match the interoperability criterion. However, if A and B are not the same concept we need, for piping the result of $\mathcal{S}'$ to $\mathcal{S}$, to ask if the concept B is more specific than the concept A, in other words, if the concept B is generalized or subsumed by the concept A. If this happens then both services match again the interoperability criterion. Interoperability only makes sense when there is an application or infrastructure that allows communication and connection of different services. An example is the applications for orchestrating services by designing workflows (as described in section 2.2). Further developments for PDL include the implementation of interoperability mechanisms in Taverna.

13 Annex

13.1 A practice introduction to PDL (or dive in PDL)

In this section we present a practice approach to PDL. It is inspired by one of the first services we deployed using the PDL framework: the Meudon Stark-H broadening

computation service for Hydrogen (http://atomsonline.obspm.fr).
The exposed code take as input four parameters:

- A quantum number N_i, which corresponds to the upper energy level.
- A quantum number N_f, which corresponds to the lower energy level.
- A temperature T, which is the temperature of the simulated medium.
- A density ρ, which is an electron density.

With the existing exposure systems (mostly Soap, REST, servlet web services) the information about parameters is more or less limited to a basic *function signature*: the two first parameters are *Integer*, while the two last are *Double*. But this information is not sufficient for a user wishing to use the service without knowing a priori the code: what are the unit of these parameters? What are their physical meaning? PDL is a unified way for providing user with this information by hardcoding it directly in the software composing the service. With PDL service provider can easily express that

- N_i is *Integer*, it corresponds to the principal quantum number of the upper energy level and, as a quantum number, it has no dimension. The PDL *translation* of this sentence is:

```
<parameter dependency="required">
        <Name>InitialLevel</Name>
        <ParameterType>integer</ParameterType>
        <SkossConcept>skos.uri.to.definition.of.
                initialLevel</SkossConcept>
         <Unit>None</Unit>
         <Dimension xsi:type="AtomicConstantExpression"
         ConstantType="integer">
                <Constant>1</Constant>
        </Dimension>
</parameter>
```

  The PDL description points to the skos uri containing the definition of the physical concept. Moreover it says that the current parameter has 1 as dimension. This means that the parameter is scalar (a dimensions greater than one is for vector parameters). The required attribute indicate that the user must submit this parameter to the service, and it is not optional.
- N_f is *Integer*, it corresponds to the principal quantum number of the lower energy level and, as a quantum number, it has no dimension. The PDL *translation* of this sentence is:

```
<parameter dependency="required">
        <Name>FinalLevel</Name>
        <ParameterType>integer</ParameterType>
        <SkossConcept>skos.uri.to.definition.of.
                finalLevel</SkossConcept>
        <Unit>None</Unit>
        <Dimension xsi:type="AtomicConstantExpression"
        ConstantType="integer">
```

```
                <Constant>1</Constant>
        </Dimension>
</parameter>
```

- T is the thermodynamic temperature of the simulated medium and is expressed in Kelvin. The PDL *translation* for this sentence is:

```
<parameter dependency="required">
        <Name>Temperature</Name>
        <ParameterType>real</ParameterType>
        <SkossConcept>skos.uri.to.definition.of.
                Temperature</SkossConcept>
        <Unit>K</Unit>
        <Dimension xsi:type="AtomicConstantExpression"
        ConstantType="integer">
                <Constant>1</Constant>
        </Dimension>
</parameter>
```

- ρ is an electron density in cm^{-3}. The PDL version is:

```
<parameter dependency="required">
        <Name>Density</Name>
        <ParameterType>real</ParameterType>
        <SkossConcept>skos.uri.to.definition.of.
                Density</SkossConcept>
        <Unit>cm^-3</Unit>
        <Dimension xsi:type="AtomicConstantExpression"
        ConstantType="integer">
                <Constant>1</Constant>
        </Dimension>
        </parameter>
</Parameters>
```

Even with this information, it is not guaranteed that users will be able to correctly use the service. Indeed, two constraints involve parameters. The first comes from the definition of N_i and N_f: the condition

$$(N_i - N_f) > 1 \tag{16}$$

must always be satisfied. The second comes from the physical model implemented into the exposed code. The result has a physical meaning only if the Debey approximation hypothesis holds:

$$\frac{9\,\rho^{1/6}}{100\,T^{1/2}} < 1 \tag{17}$$

How to alert the user of these two constraints? A first solution consists in writing explanation (e.g. a code documentation) but it is not sure that users will read it. A more secure approach consists in writing checking algorithms. But this solution is time consuming, since you have to write *ad hoc* tests for every specific code. PDL answer this issues by providing a unified way for expressing the constraints. The PDL formulation of (16) is

```
<always>
       <Criterion xsi:type="Criterion">
              <Expression xsi:type="AtomicParameterExpression">
                     <parameterRef ParameterName="FinalLevel"/>
                     <Operation operationType="MINUS">
                            <expression
                            xsi:type="AtomicParameterExpression">
                                   <parameterRef
                                   ParameterName="InitialLevel"/>
                            </expression>
                     </Operation>
              </Expression>
                  <ConditionType xsi:type="ValueLargerThan" reached="true">
                     <Value xsi:type="AtomicConstantExpression"
                     ConstantType="real">
                            <Constant>1</Constant>
                         </Value>
                  </ConditionType>
       </Criterion>
</always>
```

whereas the formulation for (17) is

```
<always>
  <Criterion xsi:type="Criterion">
     <expression xsi:type="AtomicConstantExpression" ConstantType="real">
          <Constant>0.09</Constant>
          <Operation operationType="MULTIPLY">
              <expression xsi:type="AtomicParameterExpression">
                   <parameterRef ParameterName="Density"/>
                   <power xsi:type="AtomicConstantExpression"
                   ConstantType="real">
                     <Constant>0.16666666</Constant>
                   </power>
                   <Operation operationType="DIVIDE">
                         <expression xsi:type="AtomicParameterExpression">
                             <parameterRef ParameterName="Temperature"/>
                              <power xsi:type="AtomicConstantExpression"
                              ConstantType="real">
                                     <Constant>0.5</Constant>
                              </power>
                        </expression>
                   </Operation>
             </expression>
        </Operation>
     </Expression>
     <ConditionType xsi:type="ValueSmallerThan" reached="false">
       <Value xsi:type="AtomicConstantExpression" ConstantType="real">
              <Constant>1</Constant>
       </Value>
     </ConditionType>
 </Criterion>
</always>
```

These two last pieces of XML (composing a wider PDL description) are not intended for humans. They are parsed by the PDL framework for automatically generate the checking algorithms associated with the described constraints.

The key point to retain is that PDL is simple for simple services is flexible and powerful enough for meeting description requirements coming with the most complex scientific codes (of course the associated description won't be simple).

13.2 The PDL description of the example of equation (1)

The reader will find the xml file related to this example at the following URL:
http://vo-param.googlecode.com/svn/trunk/model/documentation/
PDL-Description_example01.xml

13.3 The PDL description of the example of equation (2)

The reader will find the xml file related to this example at the following URL:
http://vo-param.googlecode.com/svn/trunk/model/documentation/
PDL-Description_Example02.xml.

13.4 The PDL XSD Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:pm="http://www.ivoa.net/xml/Parameter/v0.1" elementFormDefault="qualified"
   targetNamespace="http://www.ivoa.net/xml/Parameter/v0.1">
   <!-- needs isActive property on group - need to be able to reference a group -->
   <xs:annotation>
      <xs:documentation> IVOA Description of the set of parameters for a service</xs:documentation>
   </xs:annotation>
   <xs:element name="Service">
      <xs:annotation>
         <xs:documentation> The base service description. A
            service in this context is simply some sort of process
            that has input parameters and produces output parameters.
         </xs:documentation>
```

```xml
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ServiceId" type="xs:string" minOccurs="1"
        maxOccurs="1">
          <xs:annotation>
            <xs:documentation>The ivoa identifier for the
            service</xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="ServiceName" type="xs:string" minOccurs="1"
        maxOccurs="1"/>
        <xs:element name="Description" type="xs:string" minOccurs="1"
        maxOccurs="1"/>
        <xs:element name="Parameters" type="pm:Parameters"
        minOccurs="1" maxOccurs="1">
          <xs:annotation>
            <xs:documentation>The list of all possible parameters
            both input and output parameters</xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="Inputs" type="pm:ParameterGroup"
        minOccurs="1" maxOccurs="1">
          <xs:annotation>
            <xs:documentation>The input parameters for a
            service.</xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="Outputs" type="pm:ParameterGroup"
        minOccurs="1" maxOccurs="1">
          <xs:annotation>
            <xs:documentation>The parameters output from a
            service.</xs:documentation>
          </xs:annotation>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
    <!-- keys to ensure that parameter names are unique -->
    <xs:unique name="KeyName">
      <xs:selector xpath="./pm:ParameterList/pm:parameter"/>
      <xs:field xpath="pm:Name"/>
    </xs:unique>
    <xs:keyref name="expressionKeyref" refer="pm:KeyName">
      <xs:selector xpath=".//pm:parameterRef"/>
      <xs:field xpath="pm:parameterName"/>
    </xs:keyref>

  </xs:element>
  <xs:complexType name="Parameters">
```

```xml
    <xs:annotation>
      <xs:documentation>The list of possible parameters both input and output.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="parameter" type="pm:SingleParameter" minOccurs="1" maxOccurs="unbounded">
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ParameterReference">
    <xs:annotation>
      <xs:documentation>A reference to a parameter</xs:documentation>
    </xs:annotation>
    <xs:attribute name="ParameterName" type="xs:string">
      <xs:annotation>
        <xs:documentation>The name of the parameter being referred to.</xs:documentation>
      </xs:annotation>
    </xs:attribute>
  </xs:complexType>
  <xs:complexType name="Description">
    <xs:sequence>
      <xs:element name="humanReadableDescription" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

  <xs:simpleType name="ParameterDependency">
    <xs:annotation>
      <xs:documentation>The types that a parameter may have.</xs:documentation>
      <xs:documentation>
        Flag for saying if a parameter is required or optional
      </xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:enumeration value="required">
        <xs:annotation>
          <xs:documentation>The parameter must be provided by user.</xs:documentation>
        </xs:annotation>
      </xs:enumeration>
      <xs:enumeration value="optional">
        <xs:annotation>
          <xs:documentation>The parameter is optional.</xs:documentation>
        </xs:annotation>
      </xs:enumeration>
    </xs:restriction>
```

```
</xs:simpleType>

<xs:simpleType name="ParameterType">
  <xs:annotation>
    <xs:documentation>The types that a parameter may
    have.</xs:documentation>
    <xs:documentation>
      Note that the types are made more specific by using the UCD
      attribute of the parameter definition.
      In particular it is expected that a Parameter Model library
      would be able to recognise the more specific types associated
      with the following UCDs
      <ul>
        <li>pos - to provide a suitable widget for positions</li>
        <li>time - to provide suitable widgets for times and
        durations</li>
      </ul>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:enumeration value="boolean">
      <xs:annotation>
        <xs:documentation>A representation of a boolean - e.g.
        true/false</xs:documentation>
      </xs:annotation>
    </xs:enumeration>
    <xs:enumeration value="string">
      <xs:annotation>
        <xs:documentation>Data that can be interpreted as
        text.</xs:documentation>
      </xs:annotation>
    </xs:enumeration>
    <xs:enumeration value="integer"/>
    <xs:enumeration value="real"/>
    <xs:enumeration value="date"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="FunctionType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="size"/>
    <xs:enumeration value="abs"/>
    <xs:enumeration value="sin"/>
    <xs:enumeration value="cos"/>
    <xs:enumeration value="tan"/>
    <xs:enumeration value="asin"/>
    <xs:enumeration value="acos"/>
    <xs:enumeration value="atan"/>
    <xs:enumeration value="exp"/>
```

```
      <xs:enumeration value="log"/>
      <xs:enumeration value="sum"/>
      <xs:enumeration value="product"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="OperationType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="PLUS"/>
      <xs:enumeration value="MINUS"/>
      <xs:enumeration value="MULTIPLY"/>
      <xs:enumeration value="DIVIDE"/>
      <xs:enumeration value="SCALAR"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="SingleParameter">
    <xs:sequence>
      <xs:element name="Name" type="xs:string" minOccurs="1"
      maxOccurs="1"> </xs:element>
      <xs:element name="ParameterType" type="pm:ParameterType"
      minOccurs="1" maxOccurs="1"> </xs:element>
      <xs:element name="UCD" type="xs:string" maxOccurs="1"
      minOccurs="0"> </xs:element>
      <xs:element name="UType" type="xs:string" maxOccurs="1"
      minOccurs="0"/>
      <xs:element name="SkosConcept" type="xs:string" minOccurs="0"
      maxOccurs="1"/>
      <xs:element name="Unit" type="xs:string" minOccurs="0"
      maxOccurs="1"/>
      <xs:element name="Precision" type="pm:Expression" minOccurs="0"
      maxOccurs="1"/>
      <xs:element name="Dimension" type="pm:Expression" maxOccurs="1"
      minOccurs="1"/>
    </xs:sequence>
    <xs:attribute name="dependency" type="pm:ParameterDependency">
    </xs:attribute>
  </xs:complexType>

  <xs:complexType name="ParameterGroup">
    <xs:annotation>
      <xs:documentation>A logical grouping of
      parameters</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="Name" type="xs:string" maxOccurs="1"
      minOccurs="1">
        <xs:annotation>
          <xs:documentation>The name of the parameter group which can be used for display</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="ParameterRef" type="pm:ParameterReference" minOccurs="0"
        maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>The list of parameters that are in the group</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="ConstraintOnGroup" type="pm:ConstraintOnGroup" maxOccurs="1"
      minOccurs="0">
        <xs:annotation>
          <xs:documentation>The constraints on parameters in the group</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="ParameterGroup" type="pm:ParameterGroup" minOccurs="0"
        maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>possibly nested parameter groups</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Active" type="pm:WhenConditionalStatement" maxOccurs="1" minOccurs="0">
        <xs:annotation>
          <xs:documentation>It the goup active? i.e. should it be displayed - The default is yes if there is no active element, otherwise it is the result of the evaluation of the When conditional statement.</xs:documentation>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="ConstraintOnGroup">
    <xs:annotation>
      <xs:documentation>The possible constraints on the parameters in a group</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="ConditionalStatement" type="pm:ConditionalStatement" minOccurs="0"
        maxOccurs="unbounded"/>
```

```xml
    </xs:sequence>
  </xs:complexType>

  <xs:complexType abstract="true" name="ConditionalStatement">
    <xs:sequence>
      <xs:element name="comment" type="xs:string" minOccurs="1"
      maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="IfThenConditionalStatement">
    <xs:complexContent>
      <xs:extension base="pm:ConditionalStatement">
        <xs:sequence>
          <xs:element name="if" type="pm:If" minOccurs="1"
          maxOccurs="1"/>
          <xs:element name="then" type="pm:Then" minOccurs="1"
          maxOccurs="1"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="AlwaysConditionalStatement">
    <xs:complexContent>
      <xs:extension base="pm:ConditionalStatement">
        <xs:sequence>
          <xs:element name="always" type="pm:Always" minOccurs="1"
          maxOccurs="1"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="WhenConditionalStatement">
    <xs:annotation>
      <xs:documentation>
        A statement that has only a True or a False value
      </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="pm:ConditionalStatement">
        <xs:sequence>
          <xs:element name="when" type="pm:When"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType abstract="true" name="LogicalConnector">
    <xs:sequence>
```

```
      <xs:element name="Criterion" type="pm:AbstractCriterion"
      minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="And">
    <xs:complexContent>
      <xs:extension base="pm:LogicalConnector"/>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="Or">
    <xs:complexContent>
      <xs:extension base="pm:LogicalConnector"/>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType abstract="true" name="ConditionalClause">
    <xs:sequence>
      <xs:element name="Criterion" type="pm:AbstractCriterion"
      minOccurs="1" maxOccurs="1">
      </xs:element>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Always">
    <xs:complexContent>
      <xs:extension base="pm:ConditionalClause"/>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="If">
    <xs:complexContent>
      <xs:extension base="pm:ConditionalClause"/>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="Then">
    <xs:complexContent>
      <xs:extension base="pm:ConditionalClause"/>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="When">
    <xs:complexContent>
      <xs:extension base="pm:ConditionalClause"/>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType abstract="true" name="AbstractCondition"/>
  <xs:complexType name="IsNull">
    <xs:complexContent>
```

```
      <xs:extension base="pm:AbstractCondition"/>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="IsInteger">
    <xs:complexContent>
      <xs:extension base="pm:AbstractCondition"> </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="IsReal">
    <xs:complexContent>
      <xs:extension base="pm:AbstractCondition"> </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="BelongToSet">
    <xs:annotation>
      <xs:documentation>The value must belong to a set</xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="pm:AbstractCondition">
        <xs:sequence>
          <xs:element name="Value" type="pm:Expression" minOccurs="1" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="ValueLargerThan">
    <xs:complexContent>
      <xs:extension base="pm:AbstractCondition">
        <xs:sequence>
          <xs:element name="Value" type="pm:Expression" maxOccurs="1" minOccurs="1"/>
        </xs:sequence>
        <xs:attribute name="reached" type="xs:boolean"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="ValueSmallerThan">
    <xs:complexContent>
      <xs:extension base="pm:AbstractCondition">
        <xs:sequence>
          <xs:element name="Value" type="pm:Expression" maxOccurs="1" minOccurs="1"/>
        </xs:sequence>
        <xs:attribute name="reached" type="xs:boolean"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

```
<xs:complexType name="ValueInRange">
   <xs:complexContent>
      <xs:extension base="pm:AbstractCondition">
         <xs:sequence>
            <xs:element name="Sup" type="pm:ValueSmallerThan"
            maxOccurs="1" minOccurs="1"/>
            <xs:element name="Inf" type="pm:ValueLargerThan"
            maxOccurs="1" minOccurs="1"/>
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
<xs:complexType name="ValueDifferentFrom">
   <xs:complexContent>
      <xs:extension base="pm:AbstractCondition">
         <xs:sequence>
            <xs:element name="Value" type="pm:Expression" maxOccurs="1"
            minOccurs="1"/>
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
<xs:complexType name="DefaultValue">
   <xs:complexContent>
      <xs:extension base="pm:AbstractCondition">
         <xs:sequence>
            <xs:element name="Value" type="pm:Expression" maxOccurs="1"
            minOccurs="1"/>
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>

<xs:complexType abstract="true" name="AbstractCriterion">
   <xs:sequence>
      <xs:element name="Expression" type="pm:Expression" minOccurs="1"
      maxOccurs="1"> </xs:element>
      <xs:element name="ConditionType" type="pm:AbstractCondition"
      minOccurs="1" maxOccurs="1"/>
      <xs:element name="LogicalConnector" type="pm:LogicalConnector"
      maxOccurs="1" minOccurs="0"
      />
   </xs:sequence>
</xs:complexType>

<xs:complexType name="Criterion">
   <xs:complexContent>
      <xs:extension base="pm:AbstractCriterion"> </xs:extension>
   </xs:complexContent>
```

```
</xs:complexType>

<xs:complexType name="ParenthesisCriterion">
   <xs:complexContent>
      <xs:extension base="pm:AbstractCriterion">
         <xs:sequence>
            <xs:element name="ExternalLogicalConnector"
            type="pm:LogicalConnector" maxOccurs="1"
               minOccurs="0"/>

         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>

<xs:complexType name="Function">
   <xs:complexContent>
      <xs:extension base="pm:Expression">
         <xs:sequence>
            <xs:element name="expression" type="pm:Expression"/>
         </xs:sequence>
         <xs:attribute name="functionName" type="pm:FunctionType"/>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
<xs:complexType name="Operation">
   <xs:sequence>
      <xs:element name="expression" type="pm:Expression" maxOccurs="1"
      minOccurs="1"/>
   </xs:sequence>
   <xs:attribute name="operationType" type="pm:OperationType">
   </xs:attribute>
</xs:complexType>
<xs:complexType abstract="true" name="Expression"> </xs:complexType>
<xs:complexType name="ParenthesisContent">
   <xs:complexContent>
      <xs:extension base="pm:Expression">
         <xs:sequence>
            <xs:element name="expression" type="pm:Expression"
            minOccurs="1" maxOccurs="1"/>
            <xs:element name="power" type="pm:Expression" maxOccurs="1"
            minOccurs="0"/>
            <xs:element name="Operation" type="pm:Operation"
            maxOccurs="1" minOccurs="0"/>
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
<xs:complexType name="AtomicParameterExpression">
```

```
    <xs:complexContent>
      <xs:extension base="pm:Expression">
        <xs:sequence>
          <xs:element name="parameterRef" type="pm:ParameterReference"
          maxOccurs="1"
            minOccurs="1"> </xs:element>
          <xs:element name="power" type="pm:Expression" maxOccurs="1"
          minOccurs="0"/>
          <xs:element name="Operation" type="pm:Operation"
          maxOccurs="1" minOccurs="0"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="AtomicConstantExpression">
    <xs:complexContent>
      <xs:extension base="pm:Expression">
        <xs:sequence>
          <xs:element name="Constant" type="xs:string"
          maxOccurs="unbounded" minOccurs="1"/>
          <xs:element name="power" type="pm:Expression" maxOccurs="1"
          minOccurs="0"/>
          <xs:element name="Operation" type="pm:Operation"
          maxOccurs="1" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="ConstantType" type="pm:ParameterType"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="FunctionExpression">
    <xs:complexContent>
      <xs:extension base="pm:Expression">
        <xs:sequence>
          <xs:element name="Function" type="pm:Function" maxOccurs="1"
          minOccurs="1"/>
          <xs:element name="Power" type="pm:Expression" maxOccurs="1"
          minOccurs="0"/>
          <xs:element name="Operation" type="pm:Operation"
          maxOccurs="1" minOccurs="0"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```