International
Virtual
Observatory
Alliance

Implementing Note of the Framework built around PDL
Version 0.1

IVOA Working Draft July 13, 2012

This version:
0.1: July 13, 2012 (SVN Revision 123)

Latest version:
N/A

Previous versions:

Working Group:
GWS / Theory

Editor(s):

Authors:
Carlo Maria Zwölf

Contents

1 Introduction

BLA BLA

2 The package structure

3 Java Objects corresponding to the PDL data Model

They are all generated using JAX-B on the PDL XML schema. The generated classes corresponds to the PDL

4 The CommonsObject package

This package contains the classes

- GeneralParameter.java
- GeneralPatameterAlgebra.java

4.1 The GeneralParameter.java class

This class is the core PDL and is used for handling all the parameters. It is a way for encapsulating the usual basic types (integer, double, boolean, string) into an object. Moreover the modular design of this class allow users to easily define new types.

This attributes of this class are

- a field *value* of String type,
- a field *type* of String type,
- a field *description* of String type,
- a field *visitor* of Ivisitor type (cf. paragraph 5 for the definition of this interface).

The constructor of this class takes as arguments the quadruplet *value, type, description, visitor* and invoke the method *visit(GeneralParameter)* on the object we are creating. If all the test contained in the visitor (typically this methods verify if the value provided could be cast to the type contained into the *type* field) pass with no problem, then the object is created. If a problem is encountered, the constructor throws an *InvalidParameterException* exception explaining the reasons of the problem.

With this mechanisms, the validation of a GeneralParameter is automatically performed at its own construction. All the existing instances of GeneralParameter are natively validated. Moreover, just by modifying the content of the visitor class passed as arguments, developers could easily add support for new types.

4.2 The GeneralParameterAlgebra.java class

This class is implemented using the singleton pattern. In the following, let g_i be a family of GeneralParameter instances. This class contains the methods for computing:

- the sum $g_r = g_i + g_j$;
- the multiplication $g_r = g_i \cdot g_j$;
- the difference of $g_r = g_i - g_j$. ;
  For these last three items, if the both g_i and g_j are integer, the resulting GeneralParameter will be of integer type too. If one of the two is a real, then the result will be a real (this is internally hold using double types).
- the power $g_r = g_i^{g_j}$. Since the native Java Math.pow method provide a double, the result will always be a GeneralParameter of real type;
- the absolute value $|g_r|$. Since the native Math.abs method provide a double, the result will always be a GeneralParameter of real type;
- the function $f(g_i)$, $f() \in \{\sin(), \cos(), \tan(), \sin^{-1}(), \cos^{-1}(), \tan^{-1}(), \exp(), \log()\}$. The result wil be a GeneralParamter of real type;
- the sum of all the components $g_{i,j}$ of a vector of general parameters $\vec{g}_i : g_r = \sum_j g_{i,j}$. By the choice of our implementation, the result is a GeneralParameter of real type;
- the product of all the components $g_{i,j}$ of a vector of general parameters $\vec{g}_i : g_r = \prod_j g_{i,j}$. By the choice of our implementation, the result is a GeneralParameter of real type;
- the size of a vector $\vec{g}_i$. In this case the result is a generalParameter of type Intger and the incapsulated value will be the size of the vector.

All these operations (excepted the last one) are performed only if g_i (and g_j too when it appears) is (are) numerical. In the other cases, this method will throw an InvalidExpression exception explaining the reasons of the problem;

Moreover this class contains methods for characterizing instances of GeneralParameter:

- *areGeneralParameterEqual* takes g_i and g_j and return a boolean. This last is true if
  - both g_i, g_j are numerical and the difference of the values $v(g_i)$ and $v(g_j)$ encapsulated in these objects is smaller than $\epsilon = 0.0000001$ ($v(g_i) - v(g_j) < \epsilon$),

or

  - the type of g_i is equal (in the Java equalsIgnoreCase String sense) to the type of g_j and the value if g_i is equal (again, in the Java equalsIgnoreCase String sense) to the value of g_j.

- *isFirstGreaterThanSecond* takes g_i, g_j and a boolean r. The result is true if
  - r is true and $v(g_i) \geq v(g_j)$, or
  - r is false and $v(g_i) > v(g_j)$.
- *isFirstSmallerThanSecond* takes g_i, g_j and a boolean r. The result is true if
  - r is true and $v(g_i) \leq v(g_j)$, or
  - r is false and $v(g_i) < v(g_j)$.
- *isGeneralParameterInteger* takes g_i and returns a boolean. This last is true if $v(g_i) \in \mathbb{N}$.
- *isGeneralParameterReal* takes g_i and returns a boolean. This last is true if $v(g_i) \in \mathbb{R}$.

5 The visitors package

This package contains the visitors classes used by the *GeneralParameter* class constructor (cf. paragraph 4.1).

5.1 The visitors objects

The interface *Ivisitor* describe the method *visit* which takes as argument the *GeneralParameter* instance to inspect.
The abstract *AbstractVisitor* class implements the *Ivisitor* interface. It defines

- the abstract method *buildCriteriaList* which return a list of *Icriteria* (cf. par. 5.2). Developer has to implement this function in order to define his own list of criteria. By default, we provide the *GeneralParameterVisitor* class (cf. the end of this paragraph).
- the *visit* function: the previous method is invoked and, for every Icriteria object returned, we verify if the couple (type, value) of the *GeneralParameter* instance passed as argument verify the criterion. If at least one criterion is satisfied, than the visit methods end positevely. If no criterion is satisfied, the methods throws an *InvalidParameterException* explaining the reasons of the problem.

The *GeneralParameterVisitor* class is the concrete implementation of *AbstractVisitor*. It defines the method *buildCriteriaList*. The list returned contains a *RealCriteria*, an *IntegerCriteria*, a *BooleanCriteria* and a *StringCriteria* (cf. par. 5.2).

5.2 The criteria objects

All the criteria we are going to define implements the *Icriteria* interface. This last describe the methods:

- *getAuthorizedCriteriaType*, which returns the String containing the type authorized by the concrete criterion implementing the interface.
- *verifyCriteria*, which takes as argument a couple (type,value) characterizing a *GeneralParameter*. It returns the boolean true if the criterion is satisfied and false in the other case.

This interface is implemented by *RealCriteria*, an *IntegerCriteria*, a *BooleanCriteria* and a *StringCriteria*.

5.2.1 The RealCriteria

In this class, the method *getAuthorizedCriteriaType* returns the String ('real') **used in the PDL grammar** for specifying that a parameter is of real type.
The *VerifyCriteria* method returns the boolean true if the type of the *GeneralParameter* is 'real' and if the value of the same *GeneralParameter* could be cast to a double type with no errors. If the type of the *GeneralParameter* is not 'real', this methods return false. Finally if the type of the *GeneralParameter* is 'real' and the value cannot be casted to a double, this method throws an *InvalidParameterException* explaining the reasons of the problem.

5.2.2 The IntegerCriteria

In this class, the method *getAuthorizedCriteriaType* returns the String ('Integer') **used in the PDL grammar** for specifying that a parameter is of Integer type.
The *VerifyCriteria* method returns the boolean true if the type of the *GeneralParameter* is 'Integer' and if the value of the same *GeneralParameter* could be cast to an Integer type with no errors. If the type of the *GeneralParameter* is not 'Integer', this methods return false. Finally if the type of the *GeneralParameter* is 'Integer' and the value cannot be casted to an Integer, this method throws an *InvalidParameterException* explaining the reasons of the problem.

5.2.3 The BooleanCriteria

In this class, the method *getAuthorizedCriteriaType* returns the String ('Boolean') **used in the PDL grammar** for specifying that a parameter is of Boolean type.
The *VerifyCriteria* method returns the boolean true if the type of the *GeneralParameter* is 'Boolean' and if the value of the same *GeneralParameter* is equal to 'true' or 'false'. If the type of the *GeneralParameter* is not 'Boolean', this methods return false. Finally if the type of the *GeneralParameter* is 'Boolean' and the value is not 'true' or 'false', this method throws an *InvalidParameterException* explaining the reasons of the problem.

5.2.4 The StringCriteria

In this class, the method *getAuthorizedCriteriaType* returns the String ('String') **used in the PDL grammar** for specifying that a parameter is of String type.
The *VerifyCriteria* method returns the boolean true if the type of the *GeneralParameter* is 'String' and the boolean false in the other cases.

6 The pdl.interpreter.expression package

In this package are contained all the classes for interpreting and parsing PDL expressions.
The entry point for understanding the expression interpreting mechanism is the abstract class *ExpressionParser*. It describes the abstract method *parse*, which interprets the expression invoking the method. The result of this method is a list of *GeneralParameter* instances. Since in PDL expressions are vectorial, the i-th element of that list corresponds to the result of the interpretation of i-th component of the vector expression. The abstract class *ExpressionParser* is used, jointly with the polymorphism mechanism for building the *ExpressionParserFactory*

6.1 The ExpressionParserFactory class

This class is built by implementing the *singleton* pattern and is, as its name indicates, an implementation of the *factory* pattern.
The method *buildParser* takes as argument an instance of the PDL *Expression* object and returns an *ExpressionParser*. More precisely, using introspection, this function analyze the instance of the PDL Expression:

- if the *Expression* is an instance of *AtomicConstantExpression* then the methods returns a *AtomicConstantExpressionParser*;
- if the *Expression* is an instance of *AtomicParameterExpression* then the methods returns a *AtomicParameterExpressionParser*;
- if the *Expression* is an instance of *FunctionExpression* then the methods returns a *FunctionExpressionParser*;
- if the *Expression* is an instance of *Function* then the methods returns a *FunctionParser*;
- if the *Expression* is an instance of *ParenthesisContent* then the methods returns a *ParenthesisContentParser*;
- if the *Expression* is not an instance of what seen in the previous items, the function throws an *InvalidExpression* exception.

It is important to note that, since in PDL expressions are recursive, this factory will be invoked inside every concrete class implementing *ExpressionParser*.
In what follows, we are going to describe all these concrete classes returned by the *buildParser* method (and implementing the *ExpressionParser*).

6.2 The ExpressionWithPowerParser class

This abstract class inherits from *ExpressionParser* and is used for factoring the code of all the classes interpreting PDL expressions involving powers. It specifies the method *evaluatePower* which takes as arguments two lists of *GeneralParameter*, one list for the base (let us note g_i^{base} its elements) and the other for the exponent (let us note g_i^{exp} its elements). This method returns:

- the base list of g_i^{base}, if the expos ant list is null;
- the list whose elements are $\left(g_i^{base}\right)^{g_i^{exp}}$, if both lists have the same size. The power operation is performed in the *GeneralParameter* sense, using the methods described in paragraph 4.2.
- the list whose elements are $\left(g_i^{base}\right)^{g_0^{exp}}$, if the exponent list contains only one element g_0^{exp}.

In the other cases, the *evaluatePower* method throws and *InvalidExpression* exception, specifying the reasons of the problem.

6.3 The AtomicParameterExpressionParser class

This class implements the *ExpressionWithPowerParser* and is used for interpreting the PDL AtomicParameterExpression instances. Indeed this class has an attribute of type *AtomicParameterExpression* whose value is initialized by the class constructor.
In the overridden method *parse*, the following actions are performed:

- the instance of the SingleParameter referenced by the expression, is retrieved using the *Utilities* class methods.
- if the power of the expression is not null, then we convert the power expression into a List of *GeneralParameter* by building (using the *ExpressionParserFactory*) the had hoc parser and invoking its *parse* method.
- The expression (without the operation part) is evaluated: Using the *Utilities* class we get the list of *GeneralParameter* corresponding to the user provided input vector for the SingleParameter contained into the AtomicConstantExpression. We verify that the dimension of that list corresponds to the value put in the dimension field of the SingleParameter. If this test is negative we throws an *InvalidExpression* exception. If the test is positive, we invoke the method *evaluatePower* inherited from the *ExpressionWithPowerParser* superclass.
- If the *Operation* contained into the *AtomicParameterExpression* is not null, we evaluate the results of this operation (cf 6.8, the result of the expression without the operation is in this case our first operand).
- Finally, the methods returns the list of *GeneralParameter* resulting from the previous stages.

6.4 The AtomicConstantExpressionParser class

This class implements the *ExpressionWithPowerParser* and is used for interpreting the PDL AtomicConstantExpression instances. Indeed this class has an attribute of type *AtomicConstantExpression* whose value is initialized by the class constructor.
In the overridden method *parse*, the following actions are performed:

- If the power of the expression is not null, then we convert the power expression into a List of *GeneralParameter* by building (using the *ExpressionParserFactory*) the had hoc parser and invoking its *parse* method.
- The expression (without the operation part) is evaluated: first we recover the set of constant values provided in the description of the AtomicConstantExpression. Then we try to instantiate a list of *GeneralParameter* whose values are the previously retrieved and the type is the one contained into the ConstantType attribute (of type ParameterType) contained into the AtomicConstantExpression. If no exception is thrown during this process, the method *evaluatePower* (inherited from the *ExpressionWithPowerParser* superclass) is invoked.
- If the *Operation* contained into the *AtomicConstantExpression* is not null, we evaluate the results of this operation (cf 6.8, the result of the expression without the operation is in this case the first operand).
- Finally, the methods returns the list of *GeneralParameter* resulting from the previous stages.

6.5 The ParenthesisContentParser class

This class implements the *ExpressionWithPowerParser* and is used for interpreting the PDL *ParenthesisContent* expression instances. This class has an attribute of type *ParenthesisContent* whose value is initialized by the class constructor.
In the overridden method *parse*, the following actions are performed:

- The expression 'contained into the parenthesis', i.e. with the highest priority, is first evaluated by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*.
- If the power of the expression is not null, then we convert the power expression into a List of *GeneralParameter* by building (using the *ExpressionParserFactory*) the had hoc parser and invoking its *parse* method.
- The expression (without the operation part) is evaluated: the method *evaluatePower* (inherited from the *ExpressionWithPowerParser* superclass) is invoked using as base the list of *GeneralParameter* corresponding to the expression contained into the parenthesis and as exponent the list of *GeneralParameter* corresponding to the power.
- If the *Operation* contained into the *ParenthesisContent* is not null, we evaluate the results of this operation (cf 6.8, the result of the expression without the operation is in this case the first operand).

- Finally, the methods returns the list of *GeneralParameter* resulting from the previous stages.

6.6 The FunctionParser class

This class implements the *ExpressionParser* and is used for interpreting PDL *Function* expressions.
This class has an attribute of type *Function* whose value is initialized by the class constructor.
In the overridden method *parse*, the following actions are performed:

- The argument of the function is interpreted by invoking by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*;
- According to the function type contained in the attribute *functionName* of the *Function* object, we invoke the good correspondent method contained into the GeneralParameterAlgebra class (cf. 4.2) passing the list of *GeneralParameter* (built during the previous step) as argument. The result of this invocation is returned by the function.

6.7 The FunctionExpressionParser class

This class implements the *ExpressionWithPowerParser* and is used for interpreting the PDL *FunctionExpression* expression instances. This class has indeed an attribute of type *FunctionExpression* whose value is initialized by the class constructor.
In the overridden method *parse*, the following actions are performed:

- The *Function* contained into the FunctionExpression is interpreted by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*;
- If the power of the expression is not null, then we convert the power expression into a List of *GeneralParameter* by building (using the *ExpressionParserFactory*) the had hoc parser and invoking its *parse* method.
- The expression (without the operation part) is evaluated: the method *evaluatePower* (inherited from the *ExpressionWithPowerParser* superclass) is invoked using as base the list of *GeneralParameter* corresponding to the *Function* expression contained into the *FunctionExpression* expression and as exponent the list of *GeneralParameter* corresponding to the power.
- If the *Operation* contained into the *ParenthesisContent* is not null, we evaluate the results of this operation (cf 6.8, the result of the expression without the operation is in this case the first operand).
- Finally, the methods returns the list of *GeneralParameter* resulting from the previous stages.

6.8 The OperationParser class

This class is used for interpreting the results of operations expressed in PDL syntax. This class has indeed an attribute of *Operation* type, whose value is initialized by the class constructor. The only method of this class is *processOperation*, which takes as argument a list of *GeneralParameter* representing the first operand of the operation. In this last method, the following operations are performed:

- The second operand of the operation (i.e. the *Expression* instance contained into the *Operation* object) is interpreted by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*;
- We get the *OperationType* from the *Operation* object and invoke the good corresponding method contained into the GeneralParameterAlgebra class (cf. 4.2) passing the *GeneralParameter* list corresponding to the first and second operands as arguments. The function returns the result of this last invokation.

7 The pdl.interpreter.condition package

In this package we put all the classes used for interpreting the instances of all the PDL objects implementing *AbstractCondition*.
The entry point for understanding the mechanisms of this package is the abstract class *ConditionInterpreter*. This last describe the abstract method *isConditionVerified* which a boolean value. True id the condition is verified, false in the opposite case. This method could throws three kind of exceptions: *InvalidExpression, InvalidParameterException, InvalidCondition*.
The abstract class *ConditionInterpreter* is used, jointly with the polymorphism mechanism for building the *ConditionInterpreterFactory*

7.1 The ConditionInterpreterFactory class

This class is built by implementing the *singleton* pattern and is, as its name indicates, an implementation of the *factory* pattern.
The method *buildConditionInterpreter* takes as argument an instance of the PDL *AbstractCondition* object and returns a *ConditionInterpreter*. More precisely, using introspection, this function analyze the instance of the PDL Expression:

- if the *AbstractCondition* is an instance of *ValueLargerThan* then the methods returns a *ValueLargerThanInterpreter*;
- if the *AbstractCondition* is an instance of *ValueSmallerThan* then the methods returns a *ValueSmallerThanInterpreter*;
- if the *AbstractCondition* is an instance of *ValueInRange* then the methods returns a *ValueInRangeInterpreter*;
- if the *AbstractCondition* is an instance of *ValueDifferentFrom* then the methods returns a *ValueDifferentFromInterpreter*;
- if the *AbstractCondition* is an instance of *BelongToSet* then the methods returns a *BelongToSetInterpreter*;
- if the *AbstractCondition* is an instance of *DefaultValue* then the methods returns a *DefaultValueInterpreter*;

- if the *AbstractCondition* is an instance of *IsInteger* then the methods returns a *IsIntegerInterpreter*;
- if the *AbstractCondition* is an instance of *IsReal* then the methods returns a *IsRealInterpreter*;
- if the *AbstractCondition* is an instance of *IsNull* then the methods returns a *IsNullInterpreter*;
- if the *AbstractCondition* is not an instance of what seen in the previous items, the function throws an *InvalidCondition* exception, explaining the origin of the problem.

In what follows, we are going to describe all these concrete classes returned by the *buildConditionInterpreter* method (and implementing the *ConditionInterpreter*).

7.2 The ValueLargerThanInterpreter class

This class implements the *ConditionInterpreter* and is used for interpreting the *ValueLargerThan* PDL condition. Indeed this class has an attribute of type *ValueLargerThan* whose value is initialized by the class constructor.
In the overridden *isConditionVerified* method, the following actions are performed:

- We build $\vec{g}^{exp}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* passed as argument to the function (by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*).
- We build $\vec{g}^{cond}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* contained into the field *Value* of the *ValueLargerThan* condition.
- If the lists built during the two previous steps has different sizes, we throw a *InvalidCondition* expression expelling the problem origin.
- For every couple (g_i^{exp}, g_i^{cond}) we call the method *isFirstGreaterThanSecond* of *GeneralParameterAlgebra* (by setting the boolean field reached according to the value contained in the attribute *reached* in the *ValueLargerThan* condition, cf. paragraph 4.2). The method returns the true boolean value if and only if all the results of this calls are true for all the couples. It returns false otherwise.

7.3 The ValueSmallerThanInterpreter class

This class implements the *ConditionInterpreter* and is used for interpreting the *ValueSmallerThan* PDL condition. Indeed this class has an attribute of type *ValueSmallerThan* whose value is initialized by the class constructor.
In the overridden *isConditionVerified* method, the following actions are performed:

- We build $\vec{g}^{exp}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* passed as argument to the function (by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*).
- We build $\vec{g}^{cond}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* contained into the field *Value* of the *ValueLargerThan* condition.

- If the lists built during the two previous steps has different sizes, we throw a *InvalidCondition* expression expelling the problem origin.
- For every couple (g_i^{exp}, g_i^{cond}) we call the method *isFirstSmallerThanSecond* of *GeneralParameterAlgebra* (by setting the boolean field reached according to the value contained in the attribute *reached* in the *ValueSmallerThan* condition, cf. paragraph 4.2). The method returns the true boolean value if and only if all the results of this calls are true for all the couples. It returns false otherwise.

7.4 The ValueInRangeInterpreter class

This class implements the *ConditionInterpreter* and is used for interpreting the *ValueInRange* PDL condition. Indeed this class has an attribute of type *ValueSmallerThan* whose value is initialized by the class constructor.
In the overridden *isConditionVerified* method, the following actions are performed:

- We interpret the *Sup* field of the *ValueInRange*, which is of type *ValueSmallerThan* (cf. par. 7.3)
- We interpret the *Inf* field of the *ValueInRange*, which is of type *ValueLargerThan* (cf. par. 7.2)
- We returns true if both the previous interpretations returned true, false otherwise.

7.5 The BelongToSetInterpreter class

This class implements the *ConditionInterpreter* and is used for interpreting the *BelongToSet* PDL condition. Indeed this class has an attribute of type *BelongToSet* whose value is initialized by the class constructor.
In the overridden *isConditionVerified* method, the following actions are performed:

- We build $\vec{g}^{exp}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* passed as argument to the function (by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*).
- For every expression $value_j$ appearing in the field *Value* of the *BelongToSet* object
  - We interpret $value_j$, by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*. Let $\vec{g}^{value_j}$ be this result.
  - If the list $\vec{g}^{value_j}$ resulting from this interpretation is equal to $\vec{g}^{exp}$ (i.e. $\forall i$, $g_i^{exp} = g_i^{value_j}$ in the sense defined by the method *areGeneralParameterEqual* of *GeneralParameterAlgebra*, cf. par. 4.2), then the function returns true. False otherwise.

7.6 The ValueDifferentFromInterpreter class

This class implements the *ConditionInterpreter* and is used for interpreting the *ValueDifferentFrom* PDL condition. Indeed this class has an attribute of type *ValueDifferentFrom* whose value is initialized by the class constructor.
In the overridden *isConditionVerified* method, the following actions are performed:

- We build $\vec{g}^{exp}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* passed as argument to the function (by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*).
- We build $\vec{g}^{value}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* contained into the field *Value* of the *ValueDifferentFrom* object.
- The methods return the boolean true if, at least for a given of i, we have $g_i^{exp} \neq g_i^{value}$.

7.7 The IsRealInterpreter class

This class implements the *ConditionInterpreter* and is used for interpreting the *IsReal* PDL condition. In the overridden *isConditionVerified* method, the following actions are performed:

- We build $\vec{g}^{exp}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* passed as argument to the function (by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*).
- The method returns the boolean true if for all i, g_i^{exp} is real (the test is performed by calling the method *isGeneralParameterReal* of the *GeneralParameterAlgebra* class, cf. par. 4.2).

7.8 The IsIntegerInterpreter class

This class implements the *ConditionInterpreter* and is used for interpreting the *IsInteger* PDL condition. In the overridden *isConditionVerified* method, the following actions are performed:

- We build $\vec{g}^{exp}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* passed as argument to the function (by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*).
- The method returns the boolean true if for all i, g_i^{exp} is integer (the test is performed by calling the method *isGeneralParameterInteger* of the *GeneralParameterAlgebra* class, cf. par. 4.2).

7.9 The IsNullInterpreter class

This class implements the *ConditionInterpreter* and is used for interpreting the *IsNull* PDL condition.
It is important to remark that, due to the signification of the *IsNull* condition, this last could be applied only on *AtomicParameterExpression* (because it has sense to say parameter p_1 is null, but has no sense to say $(p_1 \cdot p_2/p_3)$ is null.
In the overridden *isConditionVerified* method, the following actions are performed:

- If the *Expression* passed as parameter is not an *AtomicParameterExpression* we throw an *InvalidCondition* exception, explaining the problem source.

- We get the *SingleParameter* instance from the *ParameterReference* contained into the *Expression*.
- We look[1] for the value (or values for vector Parameter cases) provided by user for the *SingleParameter*. If no value is provided, the method return true. Otherwise, it returns false.

7.10 The DefaultValueInterpreter class

This class implements the *ConditionInterpreter* and is used in association *DefaultValue* PDL condition.
Since this particular condition don't need to be interpreted as the previous ones (it is used only for saying that the default value of a parameter is a given value) the overridden method *isConditionVerified* always returns true.

8 The pdl.interpreter.criterion package

In this package we put all the classes used for interpreting instances of all the PDL objects implementing *AbstractCriterion*. The entry point for understanding the mechanism of this package is the abstract class *AbstractCriterionInterpreter*. This last describe the abstract method *isCriterionSatisfied* which returns a boolean value: true if the criterion is satisfied, false in the opposite case. This method could throw four kind of exceptions: *InvalidExpression, InvalidParameterException, InvalidCondition, InvalidCriterion*.
The class *AbstractCriterionInterpreter* is used, jointly with the polymorphism mechanism for building the *CriterionInterpreterFactory*.

8.1 The CriterionInterpreterFactory class

This class is built by implementing the *singleton* pattern and is, as its name indicates, an implementation of the *factory* pattern.
The method *buildCriterrionInterpreter* takes as argument an instance of the PDL *AbstractCriterion* object and returns a *AbstractCriterionInterpreter*. More precisely, using introspection, this function analyze the instance of the PDL Expression:

- if the *AbstractCriterion* is an instance of *Criterion* then the methods returns a *CriterionInterpreter*;
- if the *AbstractCriterion* is an instance of *ParenthesisCriterion* then the methods returns a *ParenthesisCriterion*;
- if the *AbstractCriterion* is not an instance of what seen in the previous items, the function throws an *InvalidCriterion* exception, explaining the origin of the problem.

In what follows, we are going to describe these two concrete classes returned by the *buildCriterrionInterpreter* method (and implementing the *AbstractCriterionInterpreter*).

[1]by calling the *getuserProvidedValuesForParameter* of the *Utilities* class

8.2 The CriterionInterpreter class

This class implements the *AbstractCriterionInterpreter* and is used for interpreting the *Criterion* PDL criterion.
Indeed this class has an attribute of type *Criterion* whose value is initialized by the class constructor.
In the overridden *isCriterionSatisfied* method, the following actions are performed:

- We interpret the condition *ConditionType* contained into the *Criterion* object. For this we call the method *buildConditionInterpreter* of *ConditionInterpreterFactory* (cf. par 7.1). We call the method *isConditionVerified* (cf. par. 7) contained in the object returned by *buildConditionInterpreter* by passing as argument the *Expression* contained into the *Criterion* object. This returns the boolean value b_1.
- If the *Criterion* object has no *LogicalConnector*, we return the boolean obtained at the previous item.
- If the *Criterion* object has a *LogicalConnector*, we interpret it (as explained in paragraph 8.4), by passing to the method *interpret* of *LogicalConnectorInterpreter* b_1 as parameter. Then the method returns the result of this last interpretation.

8.3 The ParenthesisCriterionInterpreter class

This class implements the *AbstractCriterionInterpreter* and is used for interpreting the *ParenthesisCriterion* PDL object. Indeed it contains an attribute of type *Parenthesis-Criterion* whose value is initialized by the class constructor. We recall that in PDL syntax, the *ParenthesisCriterion* are used for defining complex criteria, with arbitrary evaluation priority fixed by the user.
In the overridden method *isCriterionSatisfied* the following actions are performed:

- The *AbstractCriterion* contained into the *ParenthesisCriterion* is interpreted by calling the method *isCriterionSatisfied* contained into the object returned by the method *buildCriterionInterpreter* (of the class *CriterionInterpreterFactory*, cf. par. 8.1). Let b_1 be the boolean result of this interpretation.
- If the *ParenthesisCriterion* has no *ExternalLogicalCriterion*, the method returns b_1.
- If the *ParenthesisCriterion* has no *ExternalLogicalCriterion*, the methods return the boolean resulting from the call of the method

8.4 The LogicalConnectorInterpreter class

This class is used for interpreting the abstract *LogicalConnector* objects. Indeed, it contains an attribute of *LogicalConnector* type. We recall that the two concrete classes implementing *LogicalConnector* are *And* and *Or*.
The method *interpret* takes as argument a boolean value b_1 (which is typically the result of the interpretation of a first Criterion, which contains the *LogicalConnector* we are trying to interpret) and returns a boolean value. The following actions are performed inside this method:

- The second criterion (i.e. the criterion contained into the LogicalConnector instance) is interpreted by calling the method *isCriterionSatisfied* contained into the object returned by the method *buildCriterionInterpreter* (of the class *CriterionInterpreterFactory*, cf. par. 8.1). Let b_2 be the result of this interpretation.
- If the instance of *LogicalConnector* is of type *And*, the boolean (b_1 and b_2) is returned.
- If the instance of *LogicalConnector* is of type *Or*, the boolean (b_1 or b_2) is returned.

9 The pdl.interpreter.conditionalStatement package

In this package we put all the classes used for interpreting the instances of all the PDL objects implementing *ConditionalStatement*.
The entry point for understanding the mechanisms of this package is the abstract class *ConditionalStatementInterpreter*. This last describe two abstract methods

- *isStatementSwitched* which returns a boolean value: true if the statement is switched (i.e. if we are in a case where the statement has to be considered), false otherwise.
- *isValidStatement* which returns a boolean value: true if the statement is valid, false otherwise.

The abstract class *ConditionalStatementInterpreter* is used, jointly with the polymorphism mechanism for building the *ConditionalStatementInterpreterFactory*.

9.1 The ConditionalStatementInterpreterFactory class

This class is built by implementing the *singleton* pattern and is, as its name indicates, an implementation of the *factory* pattern.
The method *buildInterpreter* takes as argument an instance of the PDL *ConditionalStatement* object and returns a *ConditionalStatementInterpreter*. More precisely, using introspection, this function analyze the instance of the PDL Expression:

- if the *ConditionalStatement* is an instance of *AlwaysConditionalStatement* then the methods returns a *AlwaysConditionalStatementInterpreter*;
- if the *ConditionalStatement* is an instance of *IfThenConditionalStatement* then the methods returns a *IfThenConditionalStatementInterpreter*;
- if the *ConditionalStatement* is not an instance of what seen in the previous items, the function throws an *InvalidConditionalStatement* exception, explaining the origin of the problem.

In what follows, we are going to describe these two concrete classes returned by the *buildInterpreter* method (and implementing the *ConditionalStatementInterpreter*).

9.2 The AlwaysConditionalStatementInterpreter class

9.3 The IfThenConditionalStatementInterpreter class