The Odysseus Procedural Query Language (PQL) Framework

This document describes the basic concepts of the Procedural Query Language (PQL) of Odysseus and extend the language. In contrast to languages SQL based languages like the Continuous Query Language (CQL) or StreamSQL, PQL is more procedural and functional than declarative. The document is intended for developers who want to extend PQL.

- Extension of PQL – Make New Logical Operators Available
  - Extension through Annotations of Logical Operators
    - Announce Operators – The @LogicalOperator Annotation
    - Announce Parameters – The @Parameter Annotation
  - Extension through Operator Builders
    - Create the Operator Builder
    - Register the Operator Builder
  - Adding New Parameter Types

Extension of PQL – Make New Logical Operators Available

This section describes how your new created operators get available in PQL. There are at least two possibilities. The simplest way is to annotate the logical operator, so that Odysseus builds automatically all necessary things. Since this is suitable and useful for most but not in all cases, there is an alternative by implementing an operator builder. Finally, if new parameters are needed, the last section shows how to introduce new types of parameters.

Extension through Annotations of Logical Operators

The annotation framework exists of two parts. First, the operator must be annotated with @LogicalOperator. Secondly, the parameters of the operator must be defined via a @Parameter annotation. These parts are described in the following.

Announce Operators – The @LogicalOperator Annotation

A PQL annotation allows an operator to be automatically loaded for PQL. For that, the logical operator (normally your developed class that inherits AbstractLogicalOperator or implements the interface ILogicalOperator and ends with "AO") is used. It is necessary that this operator is in a package that contains the term logicaloperator in the package name, for example:

```
package de.uniol.inf.is.odysseus.core.server.logicaloperator;

import de.uniol.inf.is.odysseus.core.predicate.IPredicate;
```

The next step is the annotation of the class. The annotation is called "LogicalOperator" and has three parameters:
minInputPorts This is the minimal number of ports that the operator needs.
maxInputPorts This is the maximal number of ports that the operator is can handle
name It describes the (case insensitive) name of the operator.
For the selection you can see in the example, that the selection needs at least one input and is only able to handle one input.

```
@LogicalOperator(maxInputPorts=1, minInputPorts=1, name="SELECT')
public class SelectAO extends UnaryLogicalOp {
    private int rate;
```

Furthermore, the name is "SELECT". Just this declaration (and the part of the package name) provides all necessary things, so that the operator is loaded and is available in PQL. Thus, you can use it as follows:
SELECT(inputoperator)
As it is described in the previous section, there could be more than one input operator. This number directly depends on the values of maxInputPorts and minInputPorts. Since the example asks for exactly one input operator ((maxInputPorts – minInputPorts)+1 = 1), the query-command SELECT(inputoperator) has also exactly one input operator. So, if you have other values for maxInputPorts and minInputPorts, for example, minimal 0 and maximal 2, there are the following possibilities in PQL:
SELECT()SELECT(inputoperator1)SELECT(inputoperator1, inputoperator2)

Announce Parameters – The @Parameter Annotation

It is also possible to define parameters for the operator. If the operator is annotated with @LogicalOperator, you can define parameters that are also available through PQL. For that, you can use the @Parameter annotation. This annotation is put above a method:

```
@Parameter(type = IntegerParameter.class, name = "heartbeatrate", optional
= true)
public void setHeartbeatrate(int rate) {
    this.rate = rate;
}
```

The method itself should be a setter-method, so that it should be normally void and has exactly one parameter (e.g. int rate in the example). The annotation has seven possible values where only one is necessary, the type:
typeThe type is necessary and describes the kind of the parameter you want to provide. The type is provided via a class that implements IParameter (or AbstractParameter in most cases). For example, the class IntegerParameter asks for an integer. As you may see, the class IntegerParameter is defined as follows:

```
public class IntegerParameter extends AbstractParameter<String> {
```

The generic type of AbstractParameter (in this case Integer) describes the Java class that the Parameter provides. Therefore, the parameter IntegerParameter is responsible for translating a user defined value (in PQL) into a Java-based class. This class is exactly the same type of the setter (in our example int rate). Remember in this example, that Java provides auto boxing between classes and primitive data types and converts an Integer object automatically to int. Summing up, this means, that the following data types have to be equal (or at least convertible):

```
public class IntegerParameter extends AbstractParameter<Integer> {

@Parameter(type = IntegerParameter.class, name = "heartbeatrate", optional
= true)
public void setHeartbeatrate(int rate) {
    this.rate = rate;
}
```

Thus, you have to choose an appropriate IParameter based class for your setter. There are already some existing types. Besides some for basic data types like BooleanParameter for a Boolean or DoubleParameter for a double, there are also some special that are described in the following:
PredicateParameter tries to parse a predicate that is normally defined like param = RelationalPredicte('1 < x') for a relational predicate. It uses a predicate builder to build the predicate of type IPredicate before the predicate is passed to the setter, which therefore needs the data type IPredicate. The RelationalPredicate, for example, is built by the Relational-PredicateBuilder that is additionally installed into the OperatorBuilderFactory. Thus, if you have other predicates than Relational-Predicate, you have to create your own IPredicateBuilder, which has to be registered to the OperatorBuilderFactory.

```
@Parameter(type=PredicateParameter.class)
public void setPredicate(IPredicate predicate) {
    super.setPredicate(predicate);
}
```

SDFExpressionParameter needs a string as input which is passed to the math expression parser (MEP). MEP converts the string into a SDFExpression that reflects a mathematical expression like (x*3)+y /5 or similar. Thus, the data type of the setter should be SDFExpression.
CreateSDFAttributeParameter creates a SDFAttribute. It takes a two valued list of strings where one value is the name of the attribute and the other value is the data type. For example, the value param= ['example', 'integer'] in PQL would be interpreted by CreateSDFAttributeParamater to create a new SDFAttribute with name example and the data type integer. This is normally used by source operators where new SDFAttributes are declared. Since this method delivers an SDFAttribute, your setter function of the logical operator must have SDFAttribute as its parameter.
ResolvedSDFAttributeParameter is similar to the previous one but is used to lookup an existing attribute instead of creating a new one. If you have an operator that needs one of the attribute from the input schema this type can look up the attribute. For example, if you have an input schema like (time, value, bid, id) this parameter allows the use to write parameters like param='bid' to choose one attribute from the schema. Like the previous one, it also delivers an SDFAttribute for your setter method.

```
@Parameter(name = "x", type=ResolvedSDFAttributeParameter.class)
public void setAttributeX(SDFAttribute attributeX) {
    this.attributeX = attributeX;
}
```

ListParameter is more a wrapper than a real parameter. It allows you to ask for a list of IParameters. This is, for example, useful if you want to have subset of schema and want to use more than one ResolvedSDFAttributeParameter. However, you do not have to use ListParameter as type for the annotation. To use this, you have to use the annotation parameter isList.

isList

isList can be set to true, if you want to provide lists. It only wraps each value into a single list. Thus, the data type of the setter must be List. The generic class of the list is equal to the data type that is defined via type (see the previous section). In the following example, the data type is string and the isList option is set to true.

```
@Parameter(name = "options", type=StringParameter.class, isList = true,
optional = true)
public void setOptions(List<String> options) {
    this.options = options;
}
```

As you can see, the data type of the setter is List<String>, because StringParameter delivers a string and isList encapsulates all into List. Remember that the user has to define the parameter as a list in PQL (see above in the first part of this document). For the previous example, the following part should be used in PQL:
options=['now', 'batch', 'small']
This would be passed as a List of these three strings. It is also possible to have other IParameter as type to bundle them into a list. For example, to define a subset of a schema, you can use ResolvedSDFAttributeParameter as a list like in this example:

```
@Parameter(name = "ATTRIBUTES", type=ResolvedSDFAttributeParameter.class,
isList = true)
public void setOutputSchemaWithList(List<SDFAttribute> outputSchema) {
    setOutputSchema(new SDFSchema("", outputSchema));
}
```

isMap

This allows the user to define key value pairs for a parameter like a Map (e.g. a HashMap) in Java. Like in isList, the data type of the value is provided via the type declaration. Additionally, there is also a similar construct for the data type of the key. Thus, you have to define the key through the keytype parameter of the annotation. Since all entries are mapped to a Map, the data type of the setter must be a Map and the generics are the data type that is provided by keytype and by type. For example, if you have pairs of SDFAttribute and a String, the data type of the setter is Map<SDFAttribute, String> like in the following example:

```
@Parameter(type=StrinParameter.class,
keytype=ResolvedSDFAttributeParameter.class, isMap = true)
public void setDefault(Map<SDFAttribute, String> defaults) {
    this.defaults = defaults;
}
```

As describes in the first part of the document, this would be allow the user to define a key value pairs like:
default=['hair' = 'false', 'feathers' = 'true']
Remember, that the keytype is ResovledSDFAttributeParamter so that hair and feathers are attributes from the input schemas! Furthermore, it is also possible to use the isList option as well. This would allow you to have also lists as the value part of the pair

```
@Parameter(type=StrinParameter.class,
keytype=ResolvedSDFAttributeParameter.class, isMap = true, isList = true)
public void setDefault(Map<SDFAttribute, List<String>> nominals) {
    this.nominalsAttribute = nominals;
}
```

According to the other example, this allows the users the following example in PQL:
nominal=['hair' = ['false', 'true'], 'feathers' = ['true', 'false']]

keytype

This is similar to type and is used if isMap is also used. See isMap for further details. The default is StringParameter.class

name

This value describes the name of the parameter which is used by PQL. Normally the attribute name of the setter/getter is used. If you have the following example

```
@Parameter(name = "x", type=ResolvedSDFAttributeParameter.class)
public void setAttributeX(SDFAttribute attributeX) {
    this.attributeX = attributeX;
}
```

The attribute name would be AttributeX (the set/get is omitted!). Thus, the name of the parameter would be attributex (case insensitive). Since the name is not appropriate in most cases, the name parameter is used to define a new name. In the example, the parameter is named to X, so that the user has to use X='id' instead of attributex='x' in PQL.

optional

It marks the parameter as optional so that the parameter must not be defined by the user. The default is false, so that each parameter has to be defined by the user unless this option is not set to true.

deprecated

Marks the parameter as deprecated and is used when the parameter is not needed anymore or is replaced by another parameter and is going to be removed in the future. The default value is false.

Extension through Operator Builders

This part shows how to extend PQL with new operators, if annotating is not really possible, for example, if you have to build special parameters or if you have to use the constructor. There are two steps necessary. First you need an operator builder and secondly, you have to register the operator builder via a service.

Create the Operator Builder

This step is the creation of the operator builder. Since most steps have nearly the same semantic like the annotation, this description references some things. The steps for creating an operator builder are as follows:
Create a new class and inherit the AbstractOperatorBuilder.

```
package de.uniol.inf.is.odysseus.mining.logicaloperator;

import de.uniol.inf.is.odysseus.core.logicaloperator.ILogicalOperator;

public class ExampleOperatorBuilder extends AbstractOperatorBuilder {
    public ExampleOperatorBuilder(int minPortCount, int maxPortCount){
        super(minPortCount, maxPortCount);
    }

    public IOperatorBuilder cleanCopy() {
        return null;
    }

    protected boolean internalValidation() {
        return false;
    }

    protected ILogicalOperator createOperatorInternal() {
        return null;
    }
}
```

To remove the warning, you can generate a serial version id. Then you have to implement three methods and a constructor, which are described in more detailed in the following.

Constructor

The constructor asks for two integers: minPortCount and maxPortCount. These integers are equal to *minInputPorts* and *maxInputPorts* like they are described in the annotation (see previous section). Thus, they indicate the minimum and maximum number of incoming data ports (e.g. a selection has exactly min=max=1 port). Thus, you should pass your numbers in you constructor to the super constructor, for example, super(1, 1) if you have exactly one port. The next step is the definition of parameters. A common way is to declare and initialize each parameter of type IParameter as a class field and finally to announce the parameter by using the addParameter method in the constructor. This is shown in the following for a StringParameter called sourceName.

```
package de.uniol.inf.is.odysseus.mining.logicaloperator;

import de.uniol.inf.is.odysseus.core.logicaloperator.ILogicalOperator;

public class ExampleOperatorBuilder extends AbstractOperatorBuilder {

    private final StringParameter sourceName = new StringParameter
("SOURCE", REQUIREMENT.MANDATORY);

    public ExampleOperatorBuilder(){
        super(1, 1);
        addParameters(sourceName);
    }
}
```

According to the definition of parameters that is described in the annotation-part of this document, each parameter may have several options. Besides the name (see *name* for annotations), in most cases you also have to set the *requirement* and the *usage*. The *usage* is equal to *optional* for annotations so that you can define if the user must provide this parameter or not. *Usage* is similar because it reflects if the parameter is recent or deprecated (see *deprecated* for annotations). Since the default of usage is recent, the parameter sourceName of the example is a recent parameter that is mandatory (not optional). Of course, the exactly initializations strongly depends on the class that implements IParameter. To define lists (see the *isList* option for annotations), you have to use the ListParameter. You also have to pass the kind of IParameter that is provided by the class. For example, if you want to provide a list of strings, you have to use ListParameter in combination with StringParameter:

```
public class ExampleOperatorBuilder extends AbstractOperatorBuilder {

    private final StringParameter sourceName = new StringParameter
("SOURCE", REQUIREMENT.MANDATORY);
    private final ListParameter<String> attributes = new ListParameter<>
("ATTRIBUTES", REQUIREMENT.OPTIONAL, new StringParameter());

    public ExampleOperatorBuilder(){
        super(1, 1);
        addParameters(sourceName, attributes);
    }
}
```

Notice, the type of the generic type of ListParameter must be equal to the type that is returned by the inner parameter (called singleParameter). In the example above, the generic type is String because the defined StringParamter also delivers a String (see type for *annotations* for further details).

cleanCopy

This method is used to create a new, fully clean instance of the operator builder. Thus, you can simply return a new instance:

```
public IOperatorBuilder cleanCopy() {
    return new ExampleOperatorBuilder();
}
```

internalValidation

This method is invoked during the parsing of a PQL query right before the operators are built. It allows the class, for example, to check whether all necessary things are provided (each mandatory parameter was used by the user) or if the parameters have correct values. The AbstractOperatorBuilder already checks some things. For example, if there are (according to minPortCount) enough input operators and (according to maxPortCount) not too much input operators. It also calls the validate-method for each parameter. Thus, each parameter itself is already validated before internalValidation is called. Therefore, you normally do not have to check, if the parameter is set correctly. Notices, that this parameter based validations are not operator dependent. The ResolvedSDFAttributeParameter, for example, checks in the validate method, if one of the input schemas has the attribute that was defined by the user, but it does not check if—according to the operator—the correct attribute was chosen. The IntegerParameter may only check, if the given value is an integer, but not if it is between 100 and 150.
So, the internalValidation method could be used for validating operator based things or if you need to combine two or more parameters, e.g. if the value of one parameter depends on the value of another parameter. For example, we check if the sourceName exists in the data dictionary:

```
protected boolean internalValidation() {
    String sourceName = this.sourceName.getValue();
    if (getDataDictionary().containsViewOrStream(sourceName, getCaller()) {
        return true;
    }
    return false;
}
```

You can also see that you can access the value of the parameter through the method getValue(). Furthermore, each parameter has a method called hasValue()that is used to check if the parameter was used by the user and has a value.

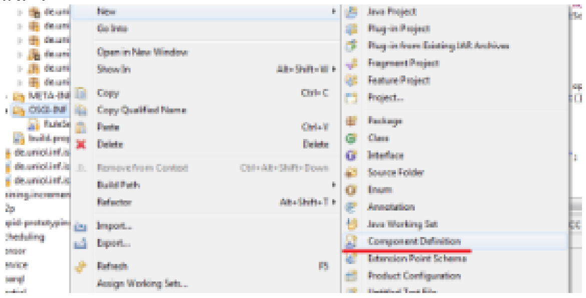
createOperatorInternal

Finally, this method is used to create the operator itself. In our case, we simply use the parameters and create the logical operator.

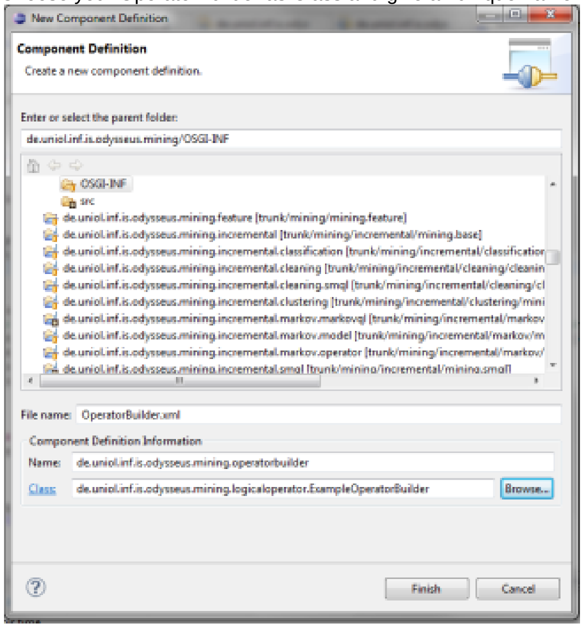
```
protected ILogicalOperator createOperatorInternal() {
    ILogicalOperator op = new ExampleAO(this.sourceName.getValue(), this.
attributes.getValue());
    return op;
}
```

Register the Operator Builder

The next step is to register the operator builder to the operator builder factory. This is done by a service declaration. First, you have to create a component definition, which is normally in a folder called "OSGI-INF".



Choose your OperatorBuilder as Class and give an unique name



New Component Definition

Component Definition

Create a new component definition.

Enter or select the parent folder:

de.uniol.inf.is.odysseus.mining/OSGI-INF

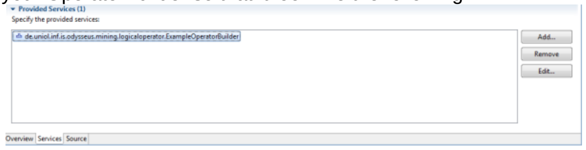
File name: OperatorBuilder.xml

Component Definition Information

Name: de.uniol.inf.is.odysseus.mining.operatorbuilder

Class: de.uniol.inf.is.odysseus.mining.logicaloperator.ExampleOperatorBuilder

In the next step, you have to add (under the "services" tab) the provided service. Click "Add…" and add your OperatorBuilder so that it look like the following:



Provided Services (1)

Specify the provided services:

de.uniol.inf.is.odysseus.mining.logicaloperator.ExampleOperatorBuilder

Take care that your bundle is loading so that the operator builder can be registered.

Adding New Parameter Types

Since there are already a lot of parameter types, firstly check if there is already a suitable parameter. However, if you need a new parameter type, you have to implement the interface IParameter. We recommend using the existing abstract implementation AbstractParameter. We show how to create a new parameter type with the help of an example.
In our example we want to have a parameter that resolves a file from a given path. We first create a new class called FileParameter and inherit the AbstractParamter class. The generic type T that is provided by the AbstractParameter is used to declare the java class that is returned by the Parameter. In our case, this should be the class java.io.File.

```
package de.uniol.inf.is.odysseus.core.server.logicaloperator.builder;

import java.io.File;

public class FileParameter extends  AbstractParameter<File> {
    protected void internalAssignment() {
    }
}
```

You may also generate a serial version uid and create the method stub for the one needed method called internalAssignment(). This method is responsible for transforming the value that is given by the user into the output format (in our case File). We assume that the user provides a string that holds a path. We can access this value via the inputValue variable, so that we are casting this value to string:

```
protected void internalAssignment() {
    String path = inputValue.toString();
}
```

Now, we can create a File object and set this object as the resulting value:

```
protected void internalAssignment() {
    String path = inputValue.toString();
    File file = new File(path);
    setValue(file);
}
```

As you can see, the output-value is set via the setValue() method. That's all. Notice, that we assume a String so that the user should use a string-based declaration (using apostrophes) like fileparam='C:
/example.csv'.
If you need further functionalities, it is possible to override some methods. For example, if you want to validate the input, you can overwrite internalValidateion(). In our example, we can, for example, check whether the file exists or not:

```
public boolean internalValidation() {
    File file = new File(inputValue.toString());
    return file.exists();
}
```

```
attributes = ['auction', 'bidder']
```