
AstroGrid-D

Deliverable



Distributed Database Access and Data Stream Management

Distributed Function Providers¹

Deliverable	D4.4
Authors	Tobias Scholl, Angelika Reiser
Editors	Tobias Scholl
Date	February 26th, 2007
Document Version	1.0.0
Current Version	1.0.0
Previous Versions	

A: Status of this Document

Deliverable D4 of working group 4.

B: Reference to project plan

Forth deliverable of working group *Distributed Database Access and Data Stream Management*.

¹This work is part of the AstroGrid-D project and D-Grid. The project is funded by the German Federal Ministry of Education and Research (BMBF).

C: Abstract

This deliverable describes how existing functionality is integrated into the data stream management system of the AstroGrid-D middleware.

We demonstrate how to integrate existing user-defined *operators* or modules into the data stream management system and publish them to the AstroGrid-D community using so-called *function providers*. Function providers are light-weight services to maintain published modules close to the developers and enable reuse within the whole community.

Using the Information service of working group 2 (Metadata management) existing modules can be discovered for cooperating researchers. Once a module is selected by users, they can integrate the module into their application with the help of the metadata provided by the function provider.

D: Change History

Version	Date	Name	Brief summary
0.1.0	27.12.2006	Tobias Scholl	Initial draft.
0.9.0	14.02.2007	Tobias Scholl	Version for the working group internal discussion.
0.10.0	26.02.2007	Tobias Scholl	Final editing for the first release.
1.0.0	19.03.2007	Tobias Scholl	First release.

E:

Contents

Abstract	2
Change History	3
1 Motivating Example: Astrometric Matching	5
2 Integration into the Data Stream Management	6
2.1 Coordinate Transformation	6
2.2 Implementing the StreamIterator Interface	7
2.3 Metadata Description	8
2.4 Publication at the Function Provider	10
3 Operator Discovery	11
4 Operator Reuse	12
4.1 Integration in Execution Plans	12
4.2 Plan Execution	13
5 Further Examples	13
5.1 Configurable Operators	13
5.1.1 Sigma Enricher	13
5.1.2 Reduced Chi Squared Filter	15
5.2 Multi-Stream Operators	17
5.2.1 Join Operator	17
References	19

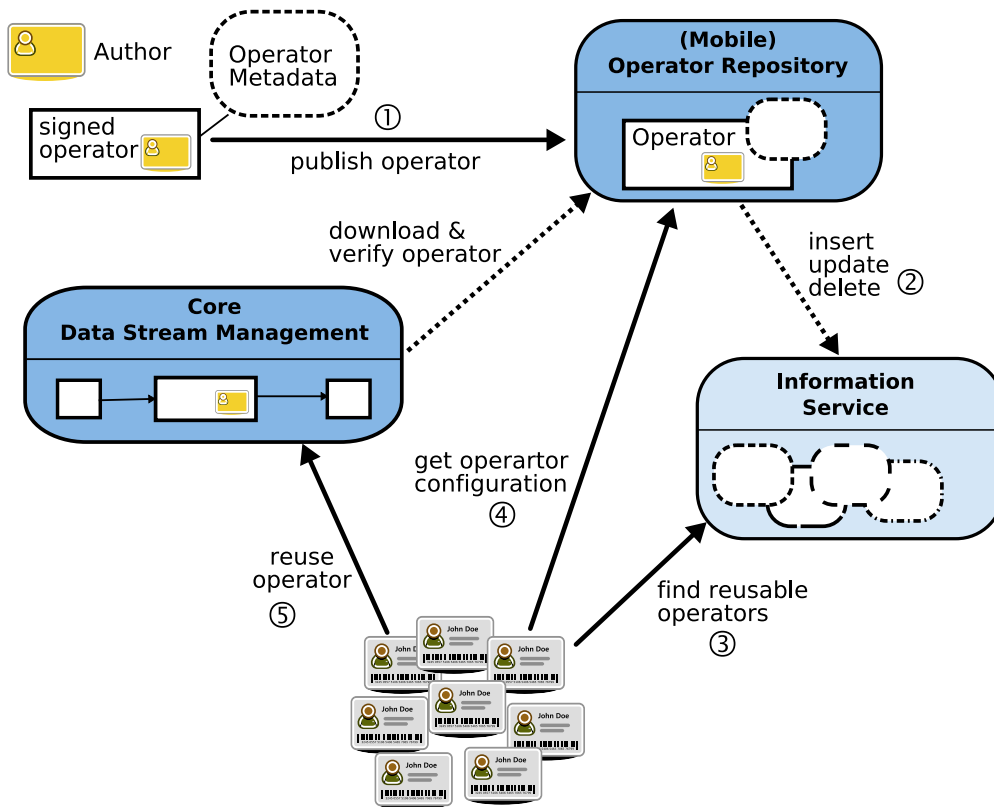


Figure 1: Overview of the integration process for user-defined operators.

1 Motivating Example: Astrometric Matching

Throughout this document we consider the *astrometric matching* use case as described in the AstroGrid-D use cases documentation², and at the eScience conference [2] as our running example to demonstrate the use of user-defined operators within our data stream management system.

The mechanism of function providers enables astronomers to load functionality during run-time into already deployed setups. So new libraries can be written on demand and seamlessly integrated.

Figure 1 shows the overall process steps necessary to integrate existing functionality into the data stream management system. Writing up the operator in order to fit into the data streaming environment, specifying its metadata, and publishing the metadata at the function provider is covered in Section 2 (steps 1-2). How reusable operators can be discovered using the information service is described in Section 3. Once having found a suitable operator (step 3), it can be configured (step 4) and reused in own processing pipelines (Section 4, step 5).

A general purpose data stream management does not provide all functionality needed by the astronomers. In the case of the astrometric matching use case,

²<http://mintaka.aip.de:8080/lenya/gac-grid/archive/TUM-AstrometricMatching.html>

several operators are user-defined operators:

- Coordinate transformation: converting spherical coordinates into Cartesian coordinates
- Sigma enricher: adding uncertainty information to the catalogs
- MergeJoin: for combining the sources of two primary match result streams. *primary match results* are the matches from a single catalog for the objects of interest.
- Chi-Squared filter: filtering matches from the merge join operator whose quality is below a configurable threshold.

In the following we concentrate on the integration of operators written in Java. Through its platform-independence and its support for dynamically loading libraries during run-time is very suitable for this purpose. Functionality written in other programming languages (Fortran, C, C++) can be also integrated as sub processes (e. g., using the *java.lang.ProcessBuilder*). However staging the appropriate binary or dependencies is beyond the scope of this deliverable.

2 Integration into the Data Stream Management

Astronomers who want to integrate one of their existing libraries into the data stream management system, need to find individual components, that can be plugged into the data stream processing. As running example, we use the coordinate transformation.

2.1 Coordinate Transformation

Given two polar coordinates α (right ascension, ra) and δ (declination, dec) on a sphere with radius R (resp. on the unit sphere, where $R = 1$), we get the Cartesian coordinates by the following equations.

$$r = R \cdot \cos \delta \quad (\cos \delta) \quad (1)$$

$$x = r \cdot \cos \alpha \quad (\cos \delta \cdot \cos \alpha) \quad (2)$$

$$y = r \cdot \sin \alpha \quad (\cos \delta \cdot \sin \alpha) \quad (3)$$

$$z = R \cdot \sin \delta \quad (\sin \delta) \quad (4)$$

The right ascension and declination are measured on the unit sphere³ and therefore we can calculate the coordinates using the special case of the equations above.

³using a certain reference system, currently *J2000*.

As an example, we transform the coordinates of an arbitrarily chosen point a ($\alpha_a = 20^h 48^m 13.3^\circ$ and $\delta_a = 33^\circ 26' 26''$) into degrees. As 24h of the right ascension coordinate correspond to 360° (15° per hour) we get approx. 312.05542 degrees right ascension and approx. 33.44056 degrees for declination. Inserted into the formulas above we get approximately $x_a = 0.55896$, $y_a = -0.61958$, $z_a = 0.55107$.

The class `org.gavo.util.math.CoordinateTransformation` realizes the above equation. It provides a method `toCartesian(double, double)` which expects the both spherical coordinates and returns a double array (`double[]`).

So the user-defined function has three parameters:

1. *RA* is the right ascension,
2. *DEC* is the declination,
3. and the array containing x, y, and z, the *cartesianCoordinates*.

2.2 Implementing the StreamIterator Interface

To integrate the user-defined functionality, a class needs to implement the `stream-globe.services.p2p.engine.operator.external.StreamIterator` interface. This interface consists of three methods `open(DynaBean, StreamWriter)`, `next(StreamIteratorEvent)`, and `close(String)`. The open-method is for configuration purposes, the next-method is used for data exchange between the embedding `StreamProcessor` and the `StreamIterator`. Finally, the close-method is called, if (one of) the input streams has terminated. For more details on the communication between `StreamIterator` and `StreamProcessor` please see the second deliverable of WG4 [4].

The implementation of the coordinate transformation is an ideal starting point to look at the functionality of a stream iterator. As the main functionality lies in the `next(StreamIteratorEvent)` method we only show a fragment of the implementation.

```
public class CoordinateStreamIterator implements StreamIterator {
    ...
    public synchronized void next(StreamIteratorEvent nextItem) {
        DynaBean params = nextItem.getParameters();
        Double ra = (Double) params.get(RA);
        Double dec = (Double) params.get(DEC);

        double[] cartesian = CoordinateTransformation.toCartesian(
            ra.doubleValue(), dec.doubleValue());

        for (int i = 0; i < cartesian.length; i++) {
            params.set(CARTESIAN, i, String.valueOf(cartesian[i]));
        }
        writer.write(nextItem);
    }
}
```

Listing 1: User-defined coordinate transformation.

The right ascension and declination are extracted from the data stream, and the Cartesian coordinates are written back to the data stream. The actual parsing and writing of the data stream is done by the embedding stream processor.

2.3 Metadata Description

Now we have a functioning StreamIterator, which performs the coordinate transformation from the two spherical coordinates to the Cartesian coordinates. Before we can publish the operator at a function provider, we define the metadata for the operator. For this metadata description we use the *eXtended Markup Language (XML)*. In the following we describe the structure of the metadata, and their representation as XML-elements.

The *name* of a stream operator is the executable of the stream operator. In case of operators implemented in Java, it is the complete class-name. The *description* of the stream operator offers space for documenting the operator with links to other resources.

The input parameters are described as *parameter* elements below a *input* element. Each input parameter has a *name*, a *type*, and a *description*. As types currently the wrapper classes (Integer, Long, ...) are supported. The input parameters are optional whenever an operator adds additional information to a data stream item regardless of the specific characteristics of the data stream item. The *Sigma Enricher* (see Section 5.1.1) is such an operator.

The structure of *output* parameters is similar, however as the results of the stream operator get serialized back into the data stream, we omitted the data type. Output parameters are optional, as some stream operators do not modify the structure of the data stream.

The description of *configuration* parameters is straightforward by using a *name* and *description* element. As not all stream operators need to be configured up-front, this part is optional.

The *jar* element describes the URL of the jar file at a function provider. The *authorizedby* tag contains the *Distinguished Name (DN)* of the author's certificate. The data stream management uses this information to verify, that the DN of the signer of the jar file and the *authorizedby* tag match and that the Certificate Authority (CA) of the author's certificate is trusted. You can retrieve it using the command `grid-cert-info -subject -file <usercert.pem>`.

Finally, developers of stream operators can specify *keywords* and optionally special *requirements* of the operator. Here you can specify dependencies like libraries, databases, or certain compilers.

The description of the coordinate transformation is as follows:

```

<streamoperator>
  <name>org.gavo.streamoperators.CoordinateStreamlterator</name>
  <description>
    Transforms spherical coordinates into cartesian coordinates.
    Spherical coordinates are specified as right ascension (ra) and
    declination (dec). The resulting cartesian coordinates consist of
    x, y, and z.
  </description>
  <input>
    <parameter>
      <name>RA</name>
      <type>double</type>
      <description>The right ascension.</description>
    </parameter>
    <parameter>
      <name>DEC</name>
      <type>double</type>
      <description>The declination.</description>
    </parameter>
  </input>
  <output>
    <parameter>
      <name>cartesianCoordinates[0]</name>
      <description>The x–coordinate.</description>
    </parameter>
    <parameter>
      <name>cartesianCoordinates[1]</name>
      <description>The y–coordinate.</description>
    </parameter>
    <parameter>
      <name>cartesianCoordinates[2]</name>
      <description>The z–coordinate.</description>
    </parameter>
  </output>
  <jar>http://www–db.in.tum.de/~scholl/starglobe–operators.jar</jar>
  <authorizedby>CN=Tobias Scholl,OU=Leibniz–Rechenzentrum,O=
    GridGermany,C=DE</authorizedby>
  <keywords>
    <keyword>coordinate</keyword>
    <keyword>transformation</keyword>
    <keyword>spherical coordinates</keyword>
    <keyword>cartesian coordinates</keyword>
  </keywords>
</streamoperator>

```

Listing 2: Metadata for coordinate transformation.

2.4 Publication at the Function Provider

Before publishing the operator to the function provider, authors need to sign their operators by using their Grid-certificates.

As each user needs a certificate in order to access Grid resources via Globus, they can use this certificate to sign the archive containing the operator. In case of operators written in Java, they can make use of the *jarsigner* tool provided by Sun to sign jar archives. Before that, the X.509 certificate needs to be transformed to a PKCS-key. A detailed description on how to sign the jar-file using the Globus certificate can be found on the web page by the GridLab project.⁴

After specifying the metadata as in Section 2.3, the XML document can be parsed into an XHTML page. This page can be then be uploaded to any standard web server and makes the information available to the AstroGrid-D community.

To generate the XHTML file for the coordinate transformation client, the following command could be issued, using the Apache Xalan parser:

```
java -jar xalan-2.6.0.jar -IN CoordinateStreamIterator.xml -XSL streamoperator.xsl -OUT CoordinateStreamIterator.html
```

Now the only thing left to share your operator with the AstroGrid-D community is to publish the metadata of the operator to the information service (step 1 in Figure 1).

For this purpose there exists the `streamglobe.fp.FunctionProvider` class. It extracts all data which will be published to the information service from the operator metadata (first parameter) and generates the RDF which can be published to the information service. Second parameter is an URI of the institution which provides the operator (e. g., for the Max-Planck-Institut für extraterrestrische Physik, MPE a suitable URI would be `http://www.mpe-garching.mpg.de/`). Third parameter is the URI to the generated web site.

An example call for the coordinate transformation is

```
java streamglobe.fp.FunctionProvider /home/scholl/astrogrid/deliverable/data/CoordinateStreamIterator.xml http://www-db.in.tum.de http://www-db.in.tum.de/~scholl/CoordinateStreamIterator.html
```

You can store the output of this command in an RDF file. Now we have all information to interact with the information service as shown in step 2 in Figure 1. We can use the `curl` utility to update the information service.

The third parameter to the function provider client is not only the link to the website⁵ where all details of the operator are described but also the identifier for the operator within the information service. Queries returning information about stream operators will return these URLs and can be easily copied into a web browser to read the full details.

⁴<http://www.gridlab.org/WorkPackages/wp-5/guide/signjar.html>

⁵in our example <http://www-db.in.tum.de/~scholl/CoordinateStreamIterator.html>

Assuming you stored the RDF in the file `coordinatetransformer.rdf` you can use the following curl command to update the information service:

```
curl --upload-file coordinatetransformer.rdf http://mintaka.aip.de:24000/
context/functionprovider/CoordinateStreamIterator
```

If you want to delete the operator, use the standard API of the information service to delete the context of your operator. At the time of the writing the following command would delete the metadata for the coordinate transformation operator:

```
curl -X DELETE http://mintaka.aip.de:24000/context/functionprovider/
CoordinateStreamIterator
```

Both the `streamoperator.xsl` XSLT-Stylesheet and the Java client (`streamglobe.fp.FunctionProvider`) are available from the AstroGrid-D Subversion repository at `svn://svn.gac-grid.org/software/functionprovider/trunk`.

3 Operator Discovery

AstroGrid-D users can discover published operators using the information service developed by the working group 2 [3]. This is step 3 in Figure 1.

The following information is directly available at the information service.

- name of the stream operator,
- its author,
- the institution that maintains the code for the operator,
- and keywords describing the functionality.

Based on this information, users are able to interrogate the information service whether there exist operators that already perform an important task.

To find all existing operators issue, users can issue the following SPARQL query:

```
PREFIX dsm:<http://dsm.gac-grid.org/functionprovider/>
SELECT ?op
WHERE { ?fp dsm:operator ?op .}
```

To find all function providers, this SPARQL query can be used:

```
PREFIX dsm:<http://dsm.gac-grid.org/functionprovider/>
SELECT DISTINCT ?fp
WHERE { ?fp dsm:operator ?op .}
```

Other users might be interested in finding operators which are suitable for x-ray data sets. Assuming the author of the stream operator has done this by specifying the keyword "x-ray", a suitable query could be:

```
PREFIX dsm:<http://dsm.gac-grid.org/functionprovider/>
SELECT ?op
WHERE { ?op dsm:keyword "x-ray" .}
```

If you want to select all relevant data from the above operator, the following SPARQL query can be used. It returns the function provider, the URL of the web page, the name of the operator, the author, and all keywords for operators that contain the keyword "x-ray".

```
PREFIX dsm: <http://dsm.gac-grid.org/functionprovider/>

SELECT ?site ?op ?name ?author ?keyword
WHERE {
  ?site dsm:operator ?op .
  ?op dsm:author ?author .
  ?op dsm:keyword ?keyword .
  ?op dsm:keyword "x-ray" .
  ?op dsm:name ?name .
}
```

4 Operator Reuse

Having found a suitable reusable operator, we now can integrate the operator into our data stream processing task.

4.1 Integration in Execution Plans

The information necessary to integrate the operator into the data stream management system is all available at the website of the operator (see Section 2.4). The *codebase* is the jar file of the operator, the *name* is the class name of the operator. Besides the *authorizedby* element, users need to specify all reminding parameters required by the operator such as configuration, input, and output parameters.

```
<streamoperator id="transform-1" codebase="http://www-db.in.tum.de/~
  scholl/starglobe-operators.jar" name="org.gavo.streamoperators.
  CoordinateStreamIterator" xsi:type="externalStreamoperatorType" >
  <dependencies>
    <stream id="stream-1" />
  </dependencies>
  <authorizedby>CN=Tobias Scholl,OU=Leibniz-Rechenzentrum,O=
    GridGermany,C=DE</authorizedby>
  <inputstreamdata id="stream-1" >
    <dtd/>
    <variable name="RA" select="._RAJ2000" type="Double"/>
    <variable name="DEC" select="._DEJ2000" type="Double"/>
```

```
</inputstreamdata>
<outputstreamdata>
  <dtd/>
  <variable name="cartesianCoordinates[0]" select="./cartesian/x" />
  <variable name="cartesianCoordinates[1]" select="./cartesian/y" />
  <variable name="cartesianCoordinates[2]" select="./cartesian/z" />
</outputstreamdata>
</streamoperator>
```

Listing 3: Stream operator configuration for the *Coordinate Transformation*.

4.2 Plan Execution

To execute the external plan, we need to setup the peer network. This can be achieved using the *servicegenerator-script* script. With this script also the data streams are installed into the network. To install the execution plan the client *install-plan* can be used. During the installation process the peer which executes an external operator, downloads the signed jar-file, verifies the signature and installs the operator.

5 Further Examples

Having gone through all steps of reusing a user-defined operator with the operator for coordinate transformation, we describe the remaining operators from the *astrometric matching* use case. The original descriptions are from [1] and slightly modified for this deliverable.

5.1 Configurable Operators

Configurable operators are such operators that have specified the *configuration* element in their metadata. This configuration data is passed to the stream operator during its initialization phase.

In the *astrometric matching* use case the *sigma enricher* and the *reduced chi-squared filter* are configurable operators.

5.1.1 Sigma Enricher

The purpose of this operator is to add uncertainty information to data streams in a consistent way. The uncertainty information is stored in a Java XML properties file `GlobalCatalogueUncertainties.properties` and is included in the JAR file of

the operator. In addition to this standard behavior, the following parameters can be specified:

- **BASEFILE:** the user additionally has the ability to specify a path to the file in the query plan containing global catalog uncertainties. The iterator will then search for the file at this location instead of using the file included in the JAR.
- **CATALOGUE:** the appropriate uncertainty is selected by specifying the catalog which this sigma enricher operates on. This maps directly to the keys given in the Java properties file.
- In the `<outputstreamdata>` element, a new XML element **SIGMA** is declared which will be included in the output stream. Each data stream item is enriched with a `<SIGMA>` element containing the corresponding uncertainty.

In Listing 4, an example configuration for the sigma enricher is shown. As the operator does not extract information from the data stream no *variable* is defined in the *inputstreamdata* element.

```

<streamoperator id="sigma-0"
  codebase="http://www-db.in.tum.de/~scholl/starglobe-operators.jar"
  name="org.gavo.streamoperators.SigmaEnricherStreamIterator"
  xsi:type="externalStreamoperatorType" >
  <dependencies>
    <streamreference id="transform-0" />
  </dependencies>
  <authorizedby>CN=Tobias Scholl,OU=Leibniz-Rechenzentrum,O=
    GridGermany,C=DE</authorizedby>
  <inputstreamdata id="transform-0" >
    <dtd/>
  </inputstreamdata>
  <outputstreamdata>
    <dtd/>
    <variable name="SIGMA" select="./SIGMA" />
  </outputstreamdata>
  <parameter>
    <key>BASEFILE</key>
    <value>
      etc/streamglobe/GlobalCatalogueUncertainties.properties
    </value>
  </parameter>
  <parameter>
    <key>CATALOGUE</key>
    <value>RASS-BSC</value>
  </parameter>
</streamoperator>

```

Listing 4: Configuration options of the *Sigma Enricher*.

5.1.2 χ_{reduced}^2 Filter

The χ_{reduced}^2 filter is a user-defined operator which has two behaviors:

- early χ_{reduced}^2 -filtering
- final χ_{reduced}^2 -filtering

The only difference is that with early χ_{reduced}^2 -filtering a static upper limit of counterparts is configured to ensure that no match candidate is accidentally dropped. This has to be the maximum number of counterparts a match candidate can have according to the scenario which represents the number of catalogs to be matched. When including the input list in the fuzzy match, this has to be taken into account, too.

When omitting the COUNTER_PARTS parameter, the module does a final filtering by computing the actual χ_{reduced}^2 value based on the effective number of counterparts per match candidate.

REDUCED_CHI_SQUARED_THRESHOLD is the threshold which decides, whether a match candidate is forwarded to the output stream, or filtered out and dropped. It can be any positive double value. When setting REDUCED_CHI_SQUARED_THRESHOLD to -1, the filter iterator is bypassed and all stream iterator events are forwarded. This is convenient for testing purposes.

To be able to calculate the χ_{reduced}^2 value this operator has to extract all Cartesian coordinates from its data stream sub-elements as well as the uncertainty σ_i attached to each data stream sub-element i . Each *sub-element* represents a counterpart within a match candidate. Since all original tag names have been renamed during join processing, the coordinates and uncertainties cannot be addressed via an XPath expression in the <inputstreamdata> element. Therefore, path selections have to be specified in <parameter> elements as one can see from Listing 5. In fact, selection is based on the structure of the original streams, i. e., before structural modifications by join operators and transformations into data stream sub-elements. It is possible because the χ_{reduced}^2 filter operator can depend on a consistent notation of Cartesian coordinates and the associated uncertainties σ_i . These are uniformly attached to streams by preceding operators (CoordinateStreamIterator and SigmaEnricherStreamIterator). The selection path relative to the original data stream element is used for a string matching process on the renamed XML element names of the actual data.

For debugging purposes, OUTPUT_REDUCED_CHI_SQUARED can be set to true and the location of the output stream element specified in <outputstreamdata>. This results in an inclusion of the χ_{reduced}^2 value in the output stream. In Listing 5, this feature is activated.

```
<streamoperator id="filter-1"
  codebase="dist/gavo.jar"
  name="org.gavo.streamoperators.ChiSquaredFilterStreamIterator"
  xsi:type="externalStreamoperatorType" >
  <dependencies>
    <streamreference id="join-1" />
  </dependencies>
  <authorizedby>CN=Tobias Scholl,OU=Leibniz-Rechenzentrum,O=
    GridGermany,C=DE</authorizedby>
  <inputstreamdata id="join-1" >
    <dtd/>
  </inputstreamdata>
  <outputstreamdata>
    <dtd/>
    <variable name="REDUCED_CHI_SQUARED" select="./chi"
      position="FIRST" />
  </outputstreamdata>
  <parameter>
    <key>REDUCED_CHI_SQUARED_THRESHOLD</key>
    <value>3</value>
  </parameter>
  <parameter>
    <key>COUNTER_PARTS</key>
    <value>11</value>
  </parameter>
  <parameter>
    <key>CARTESIAN_X</key>
    <value>./cartesian/x</value>
  </parameter>
  <parameter>
    <key>CARTESIAN_Y</key>
    <value>./cartesian/y</value>
  </parameter>
  <parameter>
    <key>CARTESIAN_Z</key>
    <value>./cartesian/z</value>
  </parameter>
  <parameter>
    <key>SIGMA</key>
    <value>./SIGMA</value>
  </parameter>
  <parameter>
    <key>OUTPUT_REDUCED_CHI_SQUARED</key>
    <value>>true</value>
  </parameter>
</streamoperator>
```

Listing 5: Configuration options of the $\chi_{reduced}^2$ filter

5.2 Multi-Stream Operators

5.2.1 Join Operator

The merge-join iterator (see Listing 6) can be configured to perform a left outer, right outer, full outer, or inner (equi-)join on any join attribute accessible by a simple XPath expression. So, configuration of the `<inputstreamdata>` elements is the first step. The join-ID is also mandatory to be included in the output stream if further joins have to be applied (e. g., in an n-way join). Other parameters are:

- `JOIN_TYPE` can be `LEFT_OUTER_JOIN`, `RIGHT_OUTER_JOIN`, `FULL_OUTER_JOIN`, or `INNER_JOIN`. The IDs of the streams taking part in the join have to be specified on the desired side (left and right outer joins are not commutative). The syntax is similar to the one used in a `FROM` clause of an SQL statement.
- `JOIN_CONDITION` describes the condition on which the join attributes are evaluated. In case of a merge-join, this can only be equality. The join attribute of a stream is specified by its stream-ID followed by the simple XPath expression already declared in the `<inputstreamdata>` elements.
- `PREFIX_SEPERATOR` and `SUFFIX_SEPERATOR`, respectively, specify the delimiter between an original XML element name within a data stream element and the attached prefix (suffix) after renaming. Only one out of these two parameters can be specified.

```
<streamoperator id="join-1"
  codebase="dist/gavo.jar"
  name="org.gavo.streamoperators.MergeJoinIterator"
  xsi:type="externalStreamoperatorType" >
  <dependencies>
    <streamreference id="stream-0" />
    <streamreference id="stream-1" />
  </dependencies>
  <authorizedby>CN=Tobias Scholl,OU=Leibniz-Rechenzentrum,O=
    GridGermany,C=DE</authorizedby>
  <inputstreamdata id="stream-0" >
    <dtd/>
    <variable name="JOIN_ATTRIBUTE" select="./id" type="Integer" />
  </inputstreamdata>
  <inputstreamdata id="stream-1" >
    <dtd/>
    <variable name="JOIN_ATTRIBUTE" select="./id" type="Integer" />
  </inputstreamdata>
```

```
<outputstreamdata>
  <dtd/>
  <variable name="JOIN_ATTRIBUTE" select="./id" />
</outputstreamdata>
<parameter>
  <key>JOIN_TYPE</key>
  <value>stream-0 LEFT_OUTER_JOIN stream-1</value>
</parameter>
<parameter>
  <key>JOIN_CONDITION</key>
  <value>stream-0./id = stream-1./id</value>
</parameter>
<parameter>
  <key>PREFIX_SEPERATOR</key>
  <value>_</value>
</parameter>
</streamoperator>
```

Listing 6: Configuration options of the join operator

F: References / Bibliography**References**

- [1] Sebastian Huber. Grid-based Processing of Multiple Data Streams in E-Science. Master's thesis, Technische Universität München, Germany, 2005.
- [2] R. Kuntschke, T. Scholl, S. Huber, A. Kemper, A. Reiser, H.-M. Adorf, G. Lemson, and W. Voges. Grid-based Data Stream Processing in e-Science. In *Proc. of the IEEE Intl. Conf. on e-Science and Grid Computing*, Amsterdam, The Netherlands, December 2006.
- [3] M. Högvist and T. Röblitz. AstroGrid-D Information Service: Requirements Specification and Architectural Design. Technical Report D2.1, AstroGrid-D project, July 2006.
- [4] T. Scholl and A. Carlson and A. Reiser. Distributed Database Access and Data Stream Management: Prototype for Manual Query Execution Plans. Technical Report D4.2, AstroGrid-D project, August 2006.