LinkOut XML API Development and Documentation.

Version April 1 2009

authors: Cees Wesseling & Willem van Deursen, PCRaster Environmental Software

Table of Contents

1Introduction

This document gives some background on the development of the API and some detailled information on the XML schema as not yet within in the PCRaster.xsd presented. For development purposes it is adviced to read this document once and then work from the API documentation, examples and the notes with PCRaster.xsd.

With the PCRaster XML interface (API) it is possible to request the PCRaster modelling engine to do a

series of computations as expressed in an XML document or a plain PCRaster ascii formatted script (here named as an ascii script).

Use cases of the API are:

- get information (reflection) on an ascii script. For example what data does it expect as input and what data will it compute as an output result.
- define a computation completely as an XML document and pass it to PCRaster.
- change input and output scenario's of an ascii script or XML document.
- set up in-memory data exchange between the API user and the modelling engine.

The design goal of the PCRaster XML interface is to have a cross platform interface with the PCRaster modelling engine. Every platform that is able to call simple C-linked external functions and can process XML in an easy way can benefit from this approach.

2Schema Development Requirements

In support of this XML API a XML-Schema is developed named PCRaster.xsd. The schema should contain:

- description of the computations required. Computations are in PCRaster known as statements such as "a=b*3" or "a = lookupscalar(table,b,3)"
- a binding mechanism as known from ascii scripts but more flexible; one should be able to rewrite the bindings easily.
- definition of auxiliary data besides grid maps such as lookup tables (and in the future time series). It must be possible to define the table "in-line" in the XML document.
- all possible messages needed to inform the API user correctly.

It should be possible to generate the XSD data bindings automatically from PCRaster.xsd by using:

- the xsd tool for C++ from http://codesynthesis.com/projects/xsd
- the xsd tool from Visual Studio .Net
- castor for Java

In case of conflicts that can not be resolved by reworking the schema, support for the xsd C++ tool will prevail.

3The general idea

The idea is to "round trip" information encoded in XML without assuming any state of the PCRaster model engine until the execution of computation is started. A script definition is a string or file containing the contents of an ascii script or XML document.

In the most simple case a a script definition is passed to the engine and executed:

```
s = pcr_createScriptFromTextString(string);
pcr_ScriptExecute(s);
pcr_destroyScript(s);
```

To get information from a script definition, alter it and then execute it the sequence is:

```
s1 = pcr_createScriptFromTextString(string);
xmlDocumentFromS1 = pcr_ScriptXMLReflection(s1);
pcr_destroyScript(s1);
// alter the xmlDocumentFromS1 and name it xmlDocumentToS2
s2 = pcr_createScriptFromXMLString(xmlDocumentToS2);
pcr_ScriptExecute(s2);
pcr_destroyScript(s2);
```

pcr_ScriptXMLReflection() will return an XML Document with all information made explicitly as defined in the input script definition. If the script definition is an ascii script, the return of pcr_ScriptXMLReflection is a XML document with the script information made explicit. (See Example 1). Otherwise the script definition is an XML document and the returned XML document is the "annotated" version of the script definition (See Example 2). An annotated version of an XML input (by pcr_createScriptFromTextString) document means that all elements of the input document are retained but informational elements are added.

4Description of the Schema

In this chapter we describe the schema on a global level and from a user perspective. Detailed documentation is given in the schema in the documentation elements.

4.1script

The script element is the generic container for a PCRaster script with the following sub elements described in separate sections below:

- executionOptions
- areaMap
- computationMask
- definition
- model
- textStatistics

4.2executionOptions

This element sets a number of options to alter a number of execution settings. The outputMapFormat sub element defines in which format the 2D maps are written. Note that the modelling engine supporting this XML API can detect multiple input formats in addition to the PCRaster format. All of pcrcalc's global options do have an element in executionOptions.

4.3areaMap

The areaMap element has a similar goal as the area map section in a PCRaster script. This element is required if no input maps (scriptInput in a field definition) are specified. This is the case if all I/O is done through memory exchange.

4.4computationMask

A coordinates sub-element can be specified to mask all input maps and hence perform all computations at a sub-area of the data.

4.5definition

A definition element introduces an unique symbol name in the document. Sub elements of definition describe various properties of the symbol. The following example shows the trivial parts of definition:

```
<definition name="FireStat">
 <description>
  <text>Fire station locations</text>
 </description>
</definition>
```

A functional grouping of the other definition sub elements is:

- symbol type: field or relation
- data exchange: input and output

Each group is presented in its own paragraph below.

4.5.1Type

The symbols that can be used in the model contents as a computation entity are fields and relations.

A field is the central computation unit used in PCRaster scripts. It can be either a spatial (a map) or a nonspatial (a number). The dataType element describe the possible PCRaster datatypes this symbol may or must have. To define a symbol as numeric (nonspatial) constant use the number sub element.

A relation is a relation in the database sense. It includes the PCRaster lookup table concept. The table can be specified in 2 forms:

- lookupTable element containing table in XML form (see use case for classification)
- indexedArray element describing only some meta data of an array that will be used in the memory transfer.

Both field and relation have sub elements that do give the symbol a constant value, since the value is defined within the model XML:

- field: number
- relation: lookupTable

If the symbol is given a constant value, the scriptInput and scriptOutput requests are ignored.

4.5.2Exchange

Script input and output is defined by attaching respectively the scriptInput and scriptOutput elements to a definition.

The presence of a scriptInput element defines the definition as an input interface element. The external sub element of scriptInput states that the input can be found externally. For example:

```
<definition name="a">
 <scriptInput>
  <external>a.bil</external>
 </scriptInput>
</definition>
```

This binds a to the map a.bil and states that a.bil MUST be present. If memory exchange is required the exchange index is given:

```
<definition name="a">
 <scriptInput>
  <memoryExchange>1</memoryExchange>
 </scriptInput>
</definition>
```

Having a scriptInput element with no sub elements is valid. It can be used to mark a symbol as input that must be supplied.

The presence of a scriptOutput element defines the definition as an output interface element. The external sub element of scriptOutput states that the output should be written to some external entity (e.g. a file). If memory exchange is required the exchange index is given. Having a scriptOutput element with no sub elements is valid. It can be used to mark a symbol as potential output, but omitting a sub element will skip the output action.

If a definition contains no scriptInput and scriptOutput elements the definition is considered to refer to an "internal" model parameter. A definition can also have both the scriptOutput and scriptInput element. This allows for example to let the model engine read an external map and pass it to the user in memory.

4.6timer

The timer elements sets the start and end time if the model has a dynamic section.

4.7model

The model element contains the actual computations requested. It contains a series of PCRaster statements. . Computations can only have field's or relation's as input. For example:

```
a = b * 3;
c = lookupscalar(inLineTable,a);
```

Note that the model element is optional. This is relevant if only some statistics (textStatistics) from certain input definitions should be generated.

Current limitations:

- the report keyword should not be used.

4.8Result Sampling Schemes

On the model result fields (or the input definitions) a number of result sampling schemes can be applied. Result sampling schemes is a new concept that combines a number of features applicable to PCRaster scripts (envisioned only textStatistics is materialized):

- write the (average) value at point (or areas) resulting in a time series.
- selective reports: write only maps at certain time steps.
- write a statistical summary of a field or 2 fields crossed. This is not possible in a PCRaster ascii script, and in a limited form with the PCRaster table application.

4.8.1textStatistics

A textStatistics element requests for calculating certain statistics of a field value or the spatial related cross occurrence of 2 fields. In the current implementation a default set of statistics is computed. The set differs on the type field dataType:

- classified (boolean,nominal,ordinal,ldd):
  - per class value: the map area covered by that class
- continuous (scalar,directional), per defined interval:
  - the map area covered having values in that interval,
  - the sum, minimum, maximum, average, standard deviation and median of the values in that interval.

If no interval is specified for continuous statistics the whole field (map) is considered to fall in a single interval. Intervals can be specified by referring to a relation element. The first column of the relation is

interpreted as the interval specification. See the examples for statistics.

5Examples

The apiExamples subdirectory of distribution contain a number of working examples. In this chapter some additional examples are presented

5.1Define a Constant

In a text script:

```
binding
 pi=scalar(3.14);
```

XML notation:

```
<definition name="pi">
 <field>
  <dataType><scalar></dataType>
  <number>3.14</number>
 </field>
</definition>
```

5.2Statistics

This example (see *statisticsMask.xml*) generates a statistics file for the area defined in its coordinateMask. The result file will look like:

	suitability.bil							
ecoClasses.bil		area	sum	minimum	maximum	average	standard deviation	median
3	[0]	0						
3	<0,0.2]	300	0,60	0,20	0,20	0,20	0,00	0,20
3	<0.2,0.4]	0						
3	<0.4,0.6]	0						
5	[0]	900	0,00	0,00	0,00	0,00	0,00	0,00
5	<0,0.2]	900	0,90	0,10	0,10	0,10	0,00	0,10
5	<0.2,0.4]	0						
5	<0.4,0.6]	0						

The table of statistics can also be passed by using MemoryExchange (see *memoryOnlyIO_StatisticsAsString.xml*). The layout of this type of output transfer is described in the DataTransferArray docs section: textString Buffer Memory Layout.

Limitation: textStatistics can only have static (non dynamic) subjects; a variable created in the dynamic section of a script can not be used as subject in a textStatistics element.

5.3Script with Input Data That Can Be Overwritten

A script might contain a default reference to data that can be overwritten. This allows in the OpenMI case to ship a script with some default data and overwrite the default if preferred. For example a script might contain:

```
<definition name="a">
 <scriptInput>
  <external>a.bil</external>
 </scriptInput>
</definition>
```

In the model configuration phase the scriptInput element might be rewritten to allow input at runtime:

```
<definition name="a">
 <scriptInput>
  <memoryExchange>1</memoryExchange>
 </scriptInput>
</definition>
```

5.4Reflection Example 1

This example demonstrates how the API generates interface elements in the reflection document. All information is deduced from the computations: what is input and output and what the required data types are:

```
s = pcr_createScriptFromTextString("zone.map = spreadzone(firestat.map,0,1);");
example1XMLReflection = pcr_ScriptXMLReflection(s);
// see example1XMLReflection.xml
```

5.5Reflection Example 2

This example does the same as example 1 but the computation is expressed in XML and the input and output parameters have a description element.

```
s = pcr_createScriptFromXMLFile("example2.xml");
example2XMLReflection= pcr_ScriptXMLReflection(s);
// see example2XMLReflection.xml
// now can "bind" the inputs and outputs
// the description element could be served to the user to inform what he
// must supply and what he can expect as output
// see example2XMLReflectionAltered.xml
pcr_destroyScript(s)
s = pcr_createScriptFromXMLFile("example2XMLReflectionAltered.xml");
pcr_ScriptExecute(s)
```

5.6Dynamic Model and memory transfer

See example in LinkOut\Examples\PCRasterLinkOutTest.py: testDynamicModel() (Executable from demo.bat or demo.sh)

Compared to dynamic pcrcalc models the following API memory transfer constructs are possible, all fragments shown are in the dynamic section

1) a map (Spatial)
pcrcalc script: `c = a - timeinput(mapStack)`
memory transfer: `c = a - d` # d will get new value transferred each timestep
2) a number (NonSpatial)
pcrcalc script: `c = a - timeinputscalar(tss, 1)` # tss with 1 column
memory transfer: `c = a - d` # d will get new value transferred each timestep

3) multi column / relation
pcrcalc script: `c = a - timeinputscalar(tss, idMap)` # tss with 1 column
memory transfer: c = a - lookupscalar(memRelation, idMap)

The layout of memRelation is described in the DataTransferArray docs section: IndexedArray Buffer Memory Layout.

4) A timeoutput construction, identical syntax, but transfer happens per row. See example memoryOnlyIO_Timeoutput.xml. The layout of the output transfer is described in the DataTransferArray docs section: timeoutput Buffer Memory Layout.