Open GeoSpatial Consortium Inc.

Date: 2019-06-05

Reference number of this Open GeoSpatial Consortium © Project Document: **OGC 08-068r3**

Version: 1.1.0

Category: Open Geospatial Consortium© Interface Standard

Editor: Peter Baumann

Web Coverage Processing Service (WCPS) Language Interface Standard

Document type:	Open Geospatial Consortium ® Interface Standard
Document subtype:	Extension Package
Document stage:	Adopted
Document language:	English

Warning

OGC official documents use a triple decimal-dot notation (i.e. MM.xx.ss). This document may be identified as MM.xx (Major.minor) and may include increments to the third dot series (schema changes) without any modification to this document, or the version displayed on the document. This means, for example, that a document labelled with versions 1.1.0 and 1.1.1 or even 1.1.9 are exactly the same except for modifications to the official schemas that are maintained and perpetually located at: http://schemas.opengis.net/. Note that corrections to the document are registered via corrigendums. A corrigendum will change the base document and notice will be given by appending a c# to the version (where # specifies the corrigendum number). In corrigendums that correct both the schemas and the base document, the third triplet of the document version will increment and the 'c1' or subsequent identifier will be appended, however the schemas will only increase the third triplet of the version.

This document is an OGC Standard. Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Contents

Page

1 Scope **1**

2 Compliance **1**

3 Normative references **1**

4 Terms and definitions **2**

5 Conventions **2**
5.1 Symbols (and abbreviated terms) **2**
5.2 UML notation **2**
5.3 Platform-neutral and platform-specific specifications **2**

6 Conceptual coverage model **3**
6.1 Coverage model **3**
6.1.1 Coverages ..... Error! Bookmark not defined.
6.1.2 Dimensions ..... Error! Bookmark not defined.
6.1.3 Locations ..... Error! Bookmark not defined.
6.1.4 Domain ..... Error! Bookmark not defined.
6.1.5 Range values and types **6**
6.1.6 Null and interpolation **7**
6.2 Coverage probing functions summary **7**

7 WCPS coverage processing language **9**
7.1 Expression syntax **10**
7.1.1 processCoveragesExpr **11**
7.1.2 processingExpr **13**
7.1.3 storeCoverageExpr **13**
7.1.4 encodedCoverageExpr **13**
7.1.5 scalarExpr **14**
7.1.6 booleanScalarExpr **14**
7.1.7 numericScalarExpr **14**
7.1.8 stringScalarExpr **14**
7.1.9 coverageExpr **14**
7.1.10 getComponentExpr ..... Error! Bookmark not defined.
7.1.11 setComponentExpr ..... Error! Bookmark not defined.
7.1.12 coverageIdentifier **18**
7.1.13 inducedExpr **19**
7.1.14 unaryInducedExpr **20**
7.1.15 unaryArithmeticExpr **20**
7.1.16 trigonometricExpr **22**
7.1.17 exponentialExpr **24**
7.1.18 booleanExpr **25**
7.1.19 castExpr **26**
7.1.20 fieldExpr **27**

7.1.21 binaryInducedExpr 28
7.1.22 rangeConstructorExpr 33
7.1.23 subsetExpr 34
7.1.24 trimExpr 35
7.1.25 extendExpr 36
7.1.26 sliceExpr 38
7.1.27 scaleExpr 40
7.1.28 crsTransformExpr 41
7.1.29 coverageConstructorExpr 45
7.1.30 coverageConstantExpr 47
7.1.31 condenseExpr 49
7.1.32 generalCondenseExpr 49
7.1.33 reduceExpr 52
7.2 Expression evaluation 53
7.2.1 Evaluation sequence 53
7.2.2 Nesting 53
7.2.3 Parentheses 53
7.2.4 Operator precedence rules 54
7.2.5 Range type compatibility and extension 54
7.3 Evaluation exceptions 56
7.4 processCoveragesExpr response 57

8 xWCPS 58
8.1 Overview 58
8.2 xWCPS language elements 58
8.2.1 letExpr 58
8.2.2 xpathExpr 60
8.2.3 xWcpsCoverageConstructorExpr 61
8.2.4 decodeCoverageExpr 67
8.2.5 switchExpr 68
8.2.6 Additional atomic types ..... Error! Bookmark not defined.
8.3 Coordinate Reference Systems (CRSs) 5
8.4 Evaluation response 70
8.5 Metadata 70
8.5.1 Operator precedence rules 71
8.6 Character encoding 71
8.7 Evaluation exceptions 72

Annex A (normative) Abstract Test Suite 73

Annex B (normative) WCPS Expression Syntax 77

Tables

Page

Table 1 – Coverage domain dimension types. ..... **Error! Bookmark not defined.**

Table 2 – Coverage range field data types. 7
Table 3 – Coverage probing functions. 8
Table 4 – reduceExpr definition via generalCondenseExpr 52
Table 5 – Type extension sequence. 55
Table 6 – Additional coverage range field data types. ...**Error! Bookmark not defined.**

i. Abstract

The OGC Web Coverage Processing Service (WCPS) defines a protocol-independent language for on-demand extraction, processing, and analysis of multi-dimensional gridded coverages ("datacubes") representing – among others – spatio-temporal sensor, image, simulation, or statistics data.

ii. Submitting organizations

The following organizations have submitted this Interface Standard to the Open Geospatial Consortium, Inc.

- Jacobs University Bremen
- rasdaman GmbH

iii. Document Contributor Contact Points

Name	Organization
Peter Baumann	Jacobs University Bremen, rasdaman GmbH

iv. Revision history

Date	Release	Author	Paragraph modified	Description
2008-04-29	0.0.1	Peter Baumann		created from 07-151r1 and 08-059r3
2008-09-01	0.0.2	Peter Baumann	Many	Final version following adoption
2008-09-22	0.0.3	Peter Baumann	Result type specs	More accurate phrasing, not just table reference
2009-01-08	1.0.0	Peter Baumann	many	Reworked scalar operations; incorporated e-vote comments; fixed syntax inconsistencies; final editorial brush-up
2019-05-30	1.1.0	Peter Baumann	many	Extended to allow processing of OGC CIS 1.1
2020-08-03	1.1.0	Peter Baumann	Several	Proofreading fixes

v. Changes to the Open Geospatial Consortium ® Abstract Specification

The Open Geospatial Consortium ® Abstract Specification does not require any changes to accommodate the technical contents of this (part of this) document.

vi. Future Work

This WCPS framework will be enhanced and extended incuding the following features:

1) Add support for further coverage types.

2) Add support for inserting, updating, and deleting coverages through expressions (harmonized with WCS-T).

Foreword

This OGC WCPS Language Standard 1.1 defines a datacube analytics language based on OGC Coverage Implementation Schema (CIS) version 1. This document supersedes WCPS version 1.0; it contains WCPS 1.0 functionality and extends it in a backwards compatible manner.

This document includes two normative Annexes, A and B.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium Inc. shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

Introduction

The OGC Web Coverage Processing Service (WCPS) defines a geo datacube analytics language for server-side retrieval, filtering, processing, and fusion of multi-dimensional geospatial grid coverages representing, for example, spatio-temporal sensor, image, simulation, or statistics data. Services implementing this language provide access to original or derived sets of grid coverage information, in forms that are useful for client-side consumption.

WCPS relies on the coverage model as defined in OGC Abstract Specification Topic 6 “Schema for Coverage Geometry and Functions “ [OGC 07-011] and the OGC Coverage Implementation Schema (CIS) Standard [OGC 09-146rX] where coverages are defined as “digital geospatial information representing space-varying phenomena”, practically speaking: regular and irregular grids, of which the regular and irregular grids are supported by WCPS currently..

The WCPS language is independent from any particular request and response encoding, as no concrete request/response protocol is assumed. For setting up a WCPS instance, therefore, a separate, additional specification establishing the concrete protocol is required. This allows embedding of WCPS into different target service frameworks. One such target framework is the WCS Processing Extension [OGC 08-059r3] of the Web Coverage Service (WCS) version 2 Standard [OGC 17-089rX].

Open Geospatial Consortium Interface: Web Coverage Processing Service (WCPS)

1 Scope

This document defines a protocol-independent language for retrieving and processing geospatial coverage datacubes. Version 1.1 keeps version 1.0 and adds handling of the coverage tpes defined in the OGC Coverage Implementation Schema (CIS) 1 [09-146rX]. Further, this version 1.1 takes into account changed and new rules OGC has established since the original version 1.0 adoption.

NOTE Following OGC's rules of compatibility among minor release numbers this WCPS 1.1 standard actually applies to all CIS 1.x versions.

2 Compliance

Annex A (normative) specifies compliance tests which shall be tested by any service claiming to implement WCPS.

3 Normative references

The following normative documents contain provisions that, through reference in this text, constitute provisions of this standard. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. For undated references, the latest edition of the normative document referred to applies.

IETF RFC 2616, *Hypertext Transfer Protocol -- HTTP/1.1*

ISO 8601:2000, *Data elements and interchange formats — Information interchange — Representation of dates and times*

OGC 07-092r1, *Definition identifier URNs in OGC namespace*, version 1.1.2

OGC 09-146r6, *Coverage Implementation Schema (CIS)*, version 1.1

OGC 17-089r1, *OpenGIS® Web Coverage Service (WCS) Core*, version 2.1

OGC 08-059r4, *WCS Processing Extension*, version 2.1

W3C XQuery 3.1. W3C Recommendation 21 March 2017, https://www.w3.org/TR/2017/REC-xquery-31-20170321/

W3C XML Path Language (XPath) Version 3.1. W3C Recommendation 21 March 2017, http://www.w3.org/TR/xpath/

4 Terms and definitions

For the purposes of this document, the terms and definitions given in the above references (in particular: WCS 2 [OGC 17-089rX]) apply.

5 Conventions

5.1 Symbols (and abbreviated terms)

This document assumes familiarity with the terms and concepts of WCS [OGC 17-089rX].

5.2 UML notation

All the diagrams that appear in this standard are presented using the Unified Modeling Language (UML) static structure diagram.

5.3 Platform-neutral and platform-specific specifications

WCPS is a high-level language independent from any client/server transmission protocol and server-side implementation paradigm. In terms of Clause 10 of OGC Abstract Specification Topic 12 “OpenGIS Service Architecture”, this document includes only Distributed Computing Platform-neutral specifications. This document specifies each operation request and response in platform-neutral fashion. This is done using a semi-formal approach to recursively specifying the language syntax and semantics. The specified platform-neutral data could be encoded in many alternative ways, each appropriate to one or more specific DCPs. One service embedding and encoding is defined in the WCS Processing Extension [08-059r4]. Other encodings may specify an API (Application Programming Interface approach) with actually no networks communication involved between “client” and “server”.

6 Conceptual coverage model

6.1 Overview

The coverage model of WCPS relies on the OGC Coverage Implementation Schema (CIS). In this Clause the coverage definitions of CIS are recapitulated. For the purpose of this standard, "coverage" means "grid coverage" unless expressed otherwise.

The semantics of WCPS expressions is defined via so-called probing functions which extract information from a coverage. Subclause 6.9 describes the constituents of a coverage by defining a set of coverage probing functions.

6.2 Coverage model

A coverage is a set of locations, each one bearing some value, together with some constraints – in the case of gridded coverages, which WCPS focuses on, all these locations ned to sit on some multi-dimensional grid. Following the mathematical notion of a function that maps elements of a domain (here: spatio-temporal coordinates) to a range (here: "pixel", "voxel", … values), a coverage consists of (Figure 1):

- a *domain set* of coordinate points: "*where in the multi-dimensional space can I find values?*"
- a *range set*: "*what are the values?*"
- a *range type*: "*what do those values mean?*"
- optional *metadata*: "*what else should I know about these data?*"

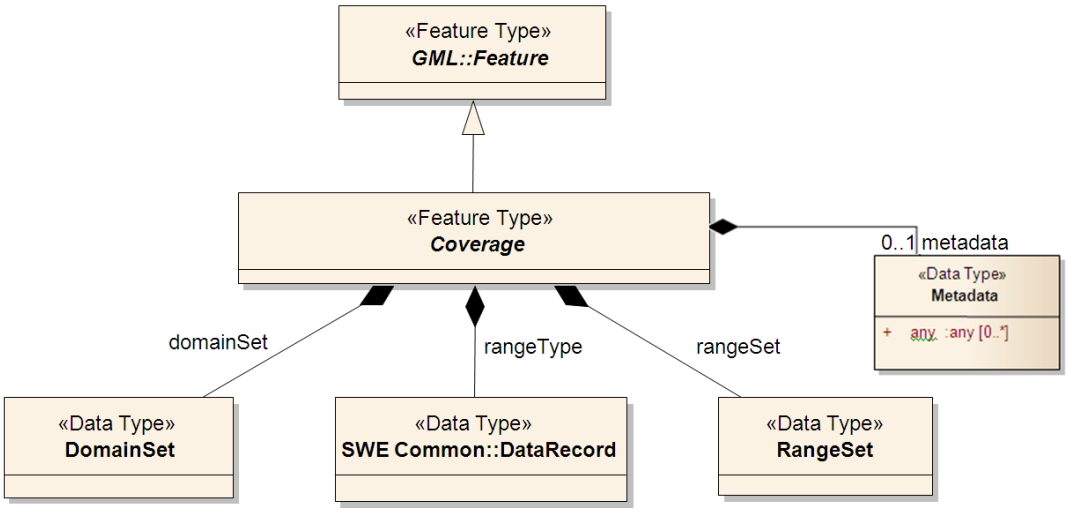

Figure 1 – Schematic coverage UML diagram

While detailing the coverage constituents we introduce a series of auxiliary functions, referred to as probing function, which will be needed for the semantics definition lateron.

6.3 Coverage Identifier

Every coverage has an identifier ("name") which is used in a WCPS query to address this coverage. Therefore – and as per WCS rule, for example – coverage identifiers must be unique within some coverage offering. Obviously this does not hold for coverages extracted and delivered somewhere. Further, coverages can be created during a query; these may have any name, including an empty string.

6.4 Domain Set, Grid, Direct Positions

The domain set contains the locations, called *direct positions*, which are expressed as coordinate tuples. For some given coverage *C*, function *domain*() delivers the domain set *D*:

$D = domain(\ C\ )$

All coordinates $c=(x_1,\ldots,x_d)$ in a domain set *D*, called the coverage's *direct positions*, share the same dimension $d>0$. These coordinates are described through some multi-dimensional *Coordinate Reference System* (CRS) combining one-dimensional axes in an ordered list:

$axisList(D) = (a_1,\ldots,a_d)$

A CRS defines an unambiguous sequence of axes, each identified in the CRS through a unique name, such as "Lat", "Long", or "date". The mathematical definition of CRSs and axes is out of the scope of this standard but defined in ISO 19111. For this standard it is sufficient that each axis contributes coordinates from a nonempty, totally ordered set of values which can be numeric or, in the general case, strings (such as "2020-08-05T"). Let c_i be the set of coordinate values axis a_i contributes in the domain set space. Then, the set of direct positions in the domain can be written mathematically as

$D = \{\ p \mid p=(x_1,\ldots,x_d),\ x_i \in c_i,\ 1 \le i \le d\ \} \subset c_1 \times \ldots \times c_d$

The gridded coverages considered in this standard impose an additional constraint: all direct positions must sit on a multi-dimensional grid. Informally speaking this means that every direct position inside the grid has exactly one next neighbor in both directions of every axis and every direct position at the rim of the grid has exactly one neighbor in each axis. Figure 2 shows some regular and irregular grid examples.

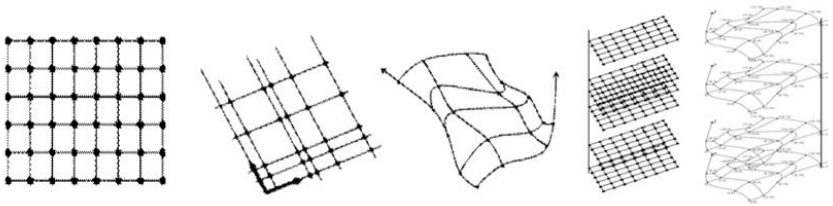

Figure 2 – Sample regular and irregular grid structures

The grid description depends on the complexity of the grid. As a grid is composed from an ordered sequence of axes the resulting complexity is determined by the types of axes (such as integer versus Lat versus time) as well as the rules determining the direct positions along these axes. The following axis / grid types are supported by WCPS, based on CIS 1.1:

- A Cartesian ("index") grid just requires lower and upper bound (which are of type integer).
- A grid which is still regular but uses some other unit of measure can be described by lower and upper bounds together with the resolution, all expressed in these units.
- An irregular grid which has individual distances along each axis can be described by a sequence of the grid points for each axis.
- More complex types, as foreseen by CIS 1.1, are not supported by this version of WCPS.

CIS 1.1 grid type *GeneralGrid* allows modelling any multi-dimensional grid based on the above notions. The specialized CIS 1.0 grid types can be described through these general terms: a *RectifiedGrid* is a grid where all axes are of type Cartesian or regular; a *ReferenceableGrid* is a grid where at least one axis is not of these types.

6.5 Coordinate Reference Systems (CRSs)

As per CIS, a coverage always has two CRSs: its Native CRS allows georeferencing, it can be of one of the axis types above. The underlying grid is specified by some Index CRS. Both CRSs have the same dimension.

NOTE The special case that the Native CRS is a pure Index CRS, in addition to the grid's Index CRS, is possible. Practical applications include abstract matrices or tensors that are not georeferenced.

The Native CRS of a domain set is obtained through function *crs*(*D*), its Index CRS through function *indexCrs*().

This standard does not define the syntax of CRS parameters. Following a change of course in OGC, URLs are generally preferred, and OGC provides a resolver[1] for CRSs, axes, etc. at URL www.opengis.net/def/crs, www.opengis.net/def/axis/, etc. Composition of CRSs can be done through a special constructor, *crs-compound*. A typical space/time CRS is given by

[1] This resolver is operated by Jacobs University on behalf of OGC. It is based on an open-source implementation available from www.rasdaman.org.

http://www.opengis.net/def/crs-compound?
1=http://www.opengis.net/def/crs/EPSG/0/4326&
2=http://www.opengis.net/def/crs/OGC/0/AnsiDate

In practice, however, this leads to lengthy CRS URLs; while this is no problem for machine-to-machine communication it is less convenient for human readers. Therefore, starting CIS 1.1 implementations may also support other well-known representations, such as WKT or GDAL-style CRS composition, for example: “EPSG:4326+OGC:Date”.

6.6 Interpolation

Starting CIS 1.1, a coverage may carry a list of admissible interpolation methods. If there are no such interpolations indicated then the overage is said to be *discrete*: only at the direct positions range values are available. Otherwise; values may also be obtained in between the direct positions by using one of the interpolation methods. Such coverages are called *continuous*.

NOTE If there is more than one interpolation method indicated then the different interpolations will usually result in different values for the same in between position, hence the coverage as a whole becomes non-deterministic.

Probing function *interpolationSet*(*c*) returns the set interpolation types applicable to the coverage, as per CIS 1.1 *GeneralGridCoverage*. This set can be empty, and will be empty if the coverage type is not *GeneralGridCoverage* ore a subtype thereof.

6.7 Range Values and Types

The value associated with a particular location within a coverage, in short: its point value, can be obtained with probing function value(*c*,*p*) which is defined for every location *p* ∈*indexDomain*(*c*) and *p* inside domain(*c*).

All direct position values of a coverage share the range type which is a record structure containing an ordered, named list of atomic values.

Requirement 1
In a coverage addressed by a WCPS request the atomic range value types **shall** be taken from Table 1.

Example For an 8-bit RGB image the range type structure is given as the triple
< red: unsigned char; green: unsigned char; blue: unsigned char>.

NOTE 1 It is not required that all range fields within a coverage are of the same type.

NOTE 2 Range fields are also known as “bands” or “channels” or “variables”.

Table 1 – Coverage range field data types.

Range data type name	Meaning
boolean	Boolean
char	8-bit signed integer
unsigned char	8-bit unsigned integer
short	16-bit signed integer
unsigned short	16-bit unsigned integer
int	32-bit signed integer
unsigned int	32-bit unsigned integer
long	64-bit signed integer
unsigned long	64-bit unsigned integer
float	Single precision floating point number
double	Double precision floating point number
complex char	2*8-bit signed complex number
complex short	2*16-bit signed complex number
complex int	2*32-bit signed complex number
complex long	2*64-bit signed complex number
complex	Single precision complex number (identical to complex int)
complex2	Double precision complex number (identical to complex long)

A coverage's range type description can be obtained through function *rangeType*() which delivers a set of pairs of field names and field type:

$$rangeType(C) = \{ (f,t) \mid f \in rangeFieldNames(C),\ t \in T \}$$

6.8 Null Set

As per CIS, a coverage has a – possibly empty – set of null values associated which are ignored, e.g., in aggregations. The set of a coverage's values to be interpreted as null is obtained via probing function *nullSet*().

6.9 Metadata

The metadata component of a coverage can contain any information. Therefore, its syntax and semantics is not known to CIS nor WCPS. Consequently, in WCPS the metadata component is treated as a byte string container.

6.10 Coverage probing functions summary

Table 2 lists the probing functions provided.

For notational convenience in this document, on the list and set valued items the usual list and set functions are assumed for extraction and manipulation, such as union, inter-

section. Further, application of some function to a list or set which is defined on the elements denotes simultaneous application of this function to all list or set elements.

Example For a set of numbers {-1, 0, 1} the abs() function produces:
abs({-1, 0, 1}) = { abs(-1), abs(0), abs(1) } = { 0, 1 }
…while for a list (-1, 0, 1) the abs() function produces:
abs((-1, 0, 1)) = (abs(-1), abs(0), abs(1)) = (1, 0, 1)

Table 2 – Coverage probing functions.

Coverage characteristic	**Probing function for some coverage** C	**Comment**
Identifier of the coverage	*identifier*(C)	Identifier of the coverage ("coverage name" in OGC WCS terminology); can be the empty string
Coverage Native CRS	*crs*(C)	Domain set CRS/axis definition
Domain extent of coverage, in Native CRS	*domain*(C) := $D_1 \times \ldots \times D_d$ where D_i={ x_i | $x_i \in$ coordinate value set of axis a_i } for $1 \le i \le d$	Extent of the coverage in its Native CRS; coordinate values are taken from the values each axis provides
Coverage Index CRS	*indexCrs*(C)	Index CRS of the coverage, allowing for Cartesian grid coordinate addressing
Domain extent of coverage, in Index CRS	*indexDomain*(C) := $I_1 \times \ldots \times I_d$ where $lo_i \le hi_i \in \mathbf{Z}$ with $lo_i \le hi_i$ for $1 \le i \le d$	Extent of the coverage in Cartesian grid coordinates, relative to the coverage's Index CRS
Interpolation methods	*interpolationSet*(C) $\subseteq$ { nearest, linear, quadratic, cubic }	Unordered set, possibly empty, of all interpolation methods applicable to this coverage
Range type	*rangeType*(C) := ((f_1,t_1),…, (f_n,t_n)) where f_i are pairwise different names, t_i are admissible base types as per Table 1	The common data type of the range values, a record containing one or more atomic components. Technically, the data type of the coverage's range values is described by (i) an ordered sequence of (field name, field type) pairs and (ii) a set of null values of this type.
Range field name list	*rangeFieldNames*(C)	Ordered list all of the coverage's range fields names
Range field type	*rangeFieldType*(C, f) for some $f \in$ *rangeFieldNames*(C)	The data type of one coverage range field, given as some atomic type as per Table 1
Null value set	*nullSet*(C) for all $n \in$ *nullSet*(C): type of $n \in$ is *rangeType*(C)	Unordered set of all values that act as null when appearing as coverage range value

Range values	*value*(*C*, *p*) for all *p* ∈ *domain*(*C*) when based on the Native CRS or *value*(*C*, *p*) for all *p* ∈ *indexDomain*(*C*) when based on the grid CRS	The coverage's range value ("pixel", "voxel") at the direct position indicated; the direct position can be expressed either in the Native or the grid (Index) CRS.
metadata	*metadata*(*C*)	The metadata associated with the coverage; **may** be an empty string

NOTE Operations in WCPS rely solely on the structural information when performing semantic checks, i.e., on structural compatibility in operations. Ensuring semantic interoperability of coverage domains and ranges is not within the current scope of WCPS.

NOTE 2 For historical reasons, *imageCrs* etc. is kept in addition to the modern terminology, *indexCrs* etc.

Requirement 2

A coverage resulting from a WCPS query **shall** augment mandatory coverage parts coherent with the requirements specified in this standard and the requirements imposed by the CIS standard.

NOTE Parts not defined in this standard and not deducible are implementation dependent.

7 WCPS coverage processing language

The WCPS coverage processing language allows clients to request processing of one or more coverages available on a server supporting WCPS. Such a server evaluates an expression and returns an appropriate response to the client. The result returned to the client upon a successful request consists of an ordered sequence of one or more coverages or scalar values.

NOTE This standard does not specify the mode of returning responses, such as synchronous or asynchronous delivery. That level of behavior is defined in protocol bindings such as the WCS Processing extension [08-059rX].

A WCPS processing request consists of a **processCoveragesExpr** (see Subclause 7.1.1). Each server claiming to support WCPS **shall** implement the coverage processing operation as specified in the following subclauses.

NOTE WCPS has been designed so as to be "safe in evaluation" – i.e., implementations are possible where any valid WCPS request can be evaluated in a finite number of steps, based on the operation primitives. Hence, WCPS implementations can be constructed in a way that no single request can render the service permanently unavailable. Notwithstanding, it still is possible to send requests that will impose high workload on a server.

NOTE 2 Data items within a WCPS response list can be heterogeneous in size and structure. In particular, the coverages within a response list can have different dimensions, domains, range types, etc. However, a response always consists of either coverages or scalar values, no mix of both.

7.1 Expression syntax

The WCPS primitives plus the nesting capabilities form an expression language which is independent from any particular encoding and collectively is referred to as the **WCPS language**. In the following subsections the language elements are detailed. The complete syntax is listed in Appendix B.

A WCPS expression is called **admissible** if and only if it adheres to the syntax and semantics of the WCPS language definition.

Requirement 3
WCPS servers **shall** return an exception in response to a WCPS request that is not admissible.

Example The coverage expression fragment

```
$C * 2
```

is admissible as it adheres to WCPS syntax whereas

```
$C $C
```

seen as part of a coverage expression violates WCPS syntax and, hence, is not admissible.

The semantics of a WCPS expression is defined by indicating, for all admissible expressions, the value of each coverage constituent as defined in Subclause 6.9.

An expression is **valid** if and only if it is admissible and it complies with the conditions imposed by the WCPS language semantics.

Example The coverage expression following is valid if and only if the WCPS offers a coverage, in the processing expression bound to variable `$C`, that has a numeric field named `red`.

```
$C.red * 2.5
```

NOTE In the remainder of this section, tables are used to describe the effect of an operation on each coverage constituent. For the reader's convenience an extra column "Changed?" is provided containing an "X" character whenever the operation changes the resp. constituent, and left blank whenever the operation does not affect the resp. constituent.

7.1.1 processCoveragesExpr

The **processCoveragesExpr** element processes a list of coverages in turn.

Each coverage is optionally checked first for fulfilling some predicate, and gets selected – i.e., contributes to an element of the result list – only if the predicate evaluates to *true*. Each coverage selected will be processed, and the result will be appended to the result list. This result list, finally, is returned as the *ProcessCoverages* response unless no exception was generated.

Requirement 4
A **processCoveragesExpr shall** be defined as below.

Let

$v_1, \ldots v_n$ be n pairwise different **iteratorVar**s ($n \geq 1$),
$L_1, \ldots L_n$ be n **coverageList**s ($n \geq 1$),
b be a **booleanScalarExpr** possibly containing occurrences of one or more v_i ($1 \leq i \leq n$),
P be a **processingExpr** possibly containing occurrences of v_i ($1 \leq i \leq n$).

Then,

for any **processCoveragesExpr** E,
where

E = **for** v_1 **in** (L_1),
v_2 **in** (L_2),
…,
v_n **in** (L_n)
where b
return P

the result R of evaluating **processCoveragesExpr** E is constructed as follows:

```
Let R be the empty sequence;
while L1 is not empty:
{   assign the first element in L1 to iteration variable v1;
    while L2 is not empty:
    {   assign the first element in L2 to iteration variable v2;
        …
            while Ln is not empty:
            {   assign the first element in Ln to iteration variable vn;
                evaluate b and P, substituting any occurrence of iteration
                variable vi by the corresponding coverage;
                if (b)
                then
                    append evaluation result to R;
                    remove the first element from Ln;
            }
        …
        }
        remove the first element from L2;
    }
    remove the first element from L1;
}
```

The elements contained in the **coverageList** clause, constituting coverage identifiers, are taken from the coverage identifiers advertised by the server.

NOTE 1 In a WCS framework such information can be obtained via a *GetCapabilities* request.

NOTE 2 Coverage identifiers may occur more than once in a **coverageList**. In this case the coverage will be evaluated each time it appears, respecting the overall inspection sequence.

Example Assume a WCPS server offers coverages A, B, and C. Then, the server may execute the following WCPS request:

```
for $c in ( A, B, C )
return encode( $c, "image/tiff" )
```

to produce a result list containing three TIFF-encoded coverages.

Example Assume a WCPS server offers same-size satellite images A, B, and C and a coverage M acting as a mask (i.e., with range values between 0 and 1 and the same extent as A, B, and C). Then, masking each satellite image can be performed with a request like the following:

```
for $s in ( A, B, C ),
    $m in ( M )
return encode( $s * $m, "image/tiff" )
```

7.1.2 processingExpr

Requirement 5
A **processingExpr** element **shall** be either a **encodedCoverageExpr** (see Subclause 7.1.4), or a **storeCoverageExpr** (see Subclause 7.1.3), or a **scalarExpr** (see Subclause 7.1.5).

7.1.3 storeCoverageExpr

NOTE The **storeCoverageExpr** element, introduced with WCPS 1.0, is removed from the standard as more comprehensive functionality for side effects is provided with other OGC standards, such as PubSub.

7.1.4 encodedCoverageExpr

The **encodedCoverageExpr** element specifies encoding of a coverage-valued request result by means of a data format and possible extra encoding parameters.

Data format encodings are governed by the OGC coverage encoding standards; these are built to materialize, to the largest extent possible, the coverage's metadata.

Example For some georeferenced coverage, a GeoTIFF result file will contain the coverage's geo coordinate and resolution information as per the OGC GeoTIFF coverage encoding standard [12-100rX].

Requirement 6
An **encodedCoverageExpr shall** be defined as below.

Let

> *C* be a **coverageExpr**,
> *f* be a string,
> where
> *f* is the name of a data format allowed for *C*,
> the data format specified by *f* supports encoding of a coverage of *C*'s domain and range.

Then,

> for any **byteString** *S*
> where *S* is one of
> S_e = **encode (*C* , *f*)**
> S_{ee} = **encode (*C* , *f*, *extraParams*)**
> with *extraParams* being a string enclosed in double quotes (‘"’)
>
> *S* is defined as that byte string which encodes *C* into the data format specified by *formatName* and the optional *extraParams*. Syntax and semantics of the *extraParams* are not specified in this standard.

NOTE 1 In a WCS framework, the data encoding formats supported can be obtained from the *supportedFormats* list contained in the response to a *GetCapabilities* request.

NOTE 2 Some format encodings may lead to a loss of information, not allowing to reconstruct a complete coverage.

NOTE 3 The extraParams are data format and implementation dependent.

Example The following expression specifies retrieval of coverage `C` encoded in HDF:

```
encode( C, "application/x-hdf" )
```

Example A WCPS implementation may encode a JPEG quality factor of 50% as the string "`.50`".

7.1.5 scalarExpr

Requirement 7
A **scalarExpr shall** be either a **getComponentExpr** (see Subclause **Error! Reference source not found.**) or a **booleanScalarExpr** (see Subclause 7.1.6) or a **numericScalarExpr** (see Subclause 7.1.7) or a **stringScalarExpr** (see Subclause 7.1.7).

NOTE As such, such an expression returns a (simple or composite) result value, that is: not a coverage.

7.1.6 booleanScalarExpr

Requirement 8
A **booleanScalarExpr shall** be a **scalarExpr** (see Subclause 7.1.5) whose result type is Boolean. Operations **shall** be the well-known Boolean functions `and`, `or`, `xor`, and `not` , arithmetic comparison (`>`, `<`, `>=`, `<=`, `=`, `!=`) on strings and numbers, and parenthesing, all bearing the well-known standard semantics.

7.1.7 numericScalarExpr

Requirement 9
A **numericScalarExpr shall** be a **scalarExpr** (see Subclause 7.1.5) whose result type is numeric (i.e., an integer, float, or complex number).

Example Numeric constants and numerically-valued metadata retrieval functions deliver numbers.

Operations provided are the well-known arithmetic (`+`, `-`, `*`, `/`) operations bearing the standard mathematical semantics. The rounding function, `round()`, rounds a real (not complex) number to the next integer number towards zero. A **condenseExpr** (see Subclause 7.1.31) derives a summary value from a coverage expression.

7.1.8 stringScalarExpr

Requirement 10
A **stringScalarExpr shall** be a **scalarExpr** (see Subclause 7.1.5) whose result type is character string of length greater or equal to zero.

Example **IdentifierExpr**s deliver strings.

7.1.9 coverageExpr

Requirement 11
A **coverageExpr shall** be either a **coverageIdentifier** (see Subclause 7.1.12), or **setCom-**

ponentExpr (see Subclause **Error! Reference source not found.**), or an **inducedExpr** (see Subclause 7.1.13), or a **subsetExpr** (see Subclause 7.1.23), or a **crsTransformExpr** (see Subclause Requirement 42), or a **scaleExpr** (see Subclause 7.1.27), or a **coverageConstructorExpr** (see Subclause 7.1.29), or a **coverageConstantExpr** (see Subclause 0).

A **coverageExpr** always evaluates to a single coverage.

7.1.10 getComponentExpr

The **getComponentExpr** element extracts a coverage description element from a coverage.

NOTE The grid point value sets ("pixels", "voxels", …) can be extracted from a coverage using subsetting operations (see Subclause 7.1.22).

Requirement 12
A **getComponentExpr shall** be defined as below.

Let

C be a **coverageExpr**.

Then,

The following metadata extraction functions are defined;
the result **shall** be given by the probing functions defined in Table 2;
strings **shall** be interpreted case-sensitive;
quotes **shall** be single or double quotes, but no mix per quoted element;
arbitrary whitespace **may** occur in between any two syntactical elements:

WCPS function (for some coverage *C*)	**Result description**	**Examples**
`id(`*C***`)`** **`identifier(`***C***`)`** **`name(`***C***`)`**	*identifier*(*C*) Name of coverage as single- or double quoted string; may be of zero length, represented by a pair of quotes	'myLittleCove ' ""
`crs(`*C***`)`**	*crs*(*C*) Identifier of the coverage's CRS (e.g., CRS URL), in quotes	'EPSG:4326+OGC :date '
`domain(`*C***`)`**	*domain*(*C*)	(lower bound, upper bound) numer-

		ic / string pair
indexCrs(*C*)	*indexCRS*(*C*) Index CRS identifier, in quotes	'Index3D '
indexDomain(*C***)** **imageCrsDomain(***C***)**	*indexDomain*(*C*) List of named axes with lower and upper bound per axis, in proper axis order as per Index CRS, in brackets	[i(0:9),j(0:100), k(-10:+10)]
indexDomain(*C,a***)** **imageCrsDomain(***C,a***)**	from *indexDomain*(*C*) , lower and upper bound for the axis indicated, as per Index CRS	(0:9)
indexDomain(*C,a***).lo** **imageCrsDomain(***C,a***).lo** **indexDomain(***C,a***).hi** **imageCrsDomain(***C,a***).hi**	from *indexDomain*(*C*) , lower / upper bound, resp., for the axis indicated, as per Index CRS	9
nullSet(*C***)**	*nullSet*(*C*) curly-braced set of values, each structured according to *rangeType*(*C*); this set may be empty.	{ (255,255,255), (0,0,0) } { -9999 }
interpolationSet(*C***)**	*interpolationSet*(*C*) parenthesized, comma-separated list of interpolation identifiers; this list may be empty.	'linear,quadratic, cubic' 'nearest'

Example For some stored coverage named “iamacoverage” addressed by variable `$c`, the following expression evaluates to “iamacoverage”:

```
id( $c )
```

NOTE 1 The term “Image CRS” is becoming deprecated by the relevant coverage standards, being replaced by “Index CRS”. For backwards compatibility the old function names are kept, with an identical semantics.

Further, several concepts based on the deprecated WCS 1.x coverage model are not supported by CIS, including interpolation default and per-axis interpolation methods. These have been dropped.

NOTE 2 Not all information about a coverage can be retrieved this way. In a WCS framework, adding the information supplied in a *GetCapabilities* and *DescribeCoverage* response provides complete information about a coverage.

7.1.11 setComponentExpr

Some of a coverage's constituents can be changed in isolation (such as its name), while others would require the whole coverage to be redefined (such as the CRS or domain). The **setComponentExpr** element allows deriving a coverage with components changed where this is possible in isolation, leaving untouched all components not addressed.

Requirement 13
A **setComponentExpr shall** be defined as below.

Let

c_1 be a **coverageExpr**,
s be a **stringConstant**,
m, n, p be integers with $m \geq 0$ and $n \geq 0$ and $p \geq 0$,
$null$ be a **rangeExpr** with $null \in$ *nullSet*(c_1),
$null_1$, …, $null_m$ be **rangeExpr**s cast-compatible with type *rangeType*(c_1),
im_1, ..., im_n be **interpolationMethod**s
with $f_i \in$ *rangeFieldNames*(c_1) and $im \in$ *interpolationSet*(c_1) for $1 \leq i \leq n$,
crs be a **crsName**.

Then,

for any **coverageExpr** c_2
where c_2 is one of

c_{id} = **setIdentifier (** c_1, s **)**
| **setId (** c_1, s **)**
| **setName (** c_1, s **)**
c_{null} = **setNull (** c_1, **{** $null_1$, …, $null_m$ **})**
c_{int} = **setInterpolation (** c_1, **{** im_1,…, im_n **})**
c_{meta} = **setMetadata (** c_1, s **)**

c_2 is defined as follows:

Coverage constituent		Changed?
identifier(c_2) = s *identifier*(c_2) = *identifier*(c_1)	for $c_2 = c_{id}$, otherwise	**X**
crs(c_2) = *crs*(c_1)		
domain(c_2) = *domain*(c_1)		
indexCrs(c_2) = *indexCrs*(c_1)		
indexDomain(c_2) = *indexDomain*(c_1)		
interpolationSet(c_{int}) = { im_1, ..., im_n },		**X**

interpolationSet(C_2) = *interpolationSet*(C_1) otherwise	
rangeType(C_2) = *rangeType*(C_1)	
for all range fields $r \in$ *rangeFieldNames*(C_2): *rangeFieldType*(C_2,r) = *rangeFieldType*(C_1,r)	
nullSet(C_2) = { $null_1$, … , $null_m$ } for C_2=C_{meta}, *nullSet*(C_2) = *nullSet*(C_1) otherwise	**X**
for all $p \in$ *domain*(C_2): *value*(C_2,p) = *value*(C_1,p)	
metadata(C_{meta}) = s for C_2=C_{meta}, *metadata*(C_2) = *metadata*(C_1) otherwise	**X**

Example The following coverage expressions set various items in a coverage:

```
setId( $c, 'myLittleCoverage' )
setNull( $c, { -9999, -9998 } )
setInterpolation( $c, { linear, quadratic } )
setMetadata( $c, "<InspireMetadata>…</InspireMetadata>" )
```

7.1.12 coverageIdentifier

The **coverageIdentifier** element represents the name of a single coverage offered by the server addressed. It is represented by a coverage variable indicated in the **processCoveragesExpr** clause (see Subclause 7.1.1).

Requirement 14
A **coverageIdentifier shall** be defined as below.

Let

id be a **variableName** bound to a coverage C_1 offered by the server.

Then,

for any **coverageExpr** C_2,
where
$C_2 = id$

C_2 is defined as follows:

Coverage constituent	Changed?
identifier(C_2) = *identifier*(C_1)	

crs(C_2) = *crs*(C_1)	
domain(C_2) = *domain*(C_1)	
indexCrs(C_2) = *indexCrs*(C_1)	
indexDomain(C_2) = *indexDomain*(C_1)	
interpolationSet(C_2) = *interpolationSet*(C_1)	
rangeType(C_2) = *rangeType*(C_1)	
for all range fields $r \in$ *rangeFieldNames*(C_2): *rangeFieldType*(C_2,r) = *rangeFieldType*(C_1,r)	
nullSet(C_2) = *nullSet*(C_1)	
for all $p \in$ *domain*(C_2): *value*(C_2,p) = *value*(C_1,p)	
metadata(C_2) = *metadata*(C_1)	

Example The following coverage expression evaluates to the complete, unchanged coverage `C`, assuming that coverage iteration variable `$C` is bound to it at the time of evaluation:

```
$C
```

7.1.13 inducedExpr

Requirement 15
An **inducedExpr shall** be either a **unaryInducedExpr** (see Subclause 7.1.14) or a **binaryInducedExpr** (see Subclause 0) or a **rangeConstructorExpr** (see Subclause 7.1.22).

Induced operations allow to simultaneously apply a function originally working on a single value to all grid point values of a coverage.

Requirement 16
In an **inducedExpr**, in case the range type contains more than one range component, the function **shall** be applied to each point simultaneously.

Requirement 17
In an **inducedExpr**, whenever a numeric argument is expected (such as a coverage with numeric range fields), Boolean *false* and *true* **shall** be interpreted as 0 and 1, resp. Conversely, whenever a Boolean argument is expected (such as a coverage with numeric range fields), then 0 and 1 **shall** be interpreted as Boolean *false* and *true*, resp.

Requirement 18
In an **inducedExpr**, whenever one of the point values (“pixels”, etc.) participating in an induced operation is equal to one of the null values of its coverage then the result of the value

combination **shall** be one of the values in the participating coverage's null value set (for a unary induced operation) or one of the values in the null value set intersection of both participating coverages (for a binary induced operation) if said intersection is not empty. If no null value is available (for a unary induced operation) or the intersection of both input coverages' null values is empty (for a binary induced operation) then the server **shall** respond with a service exception.

Requirement 19
In an **inducedExpr** the result coverage **shall** have the same domain as the input coverage(s).

NOTE 1 In case of an n-ary induced operation, n>1, all input coverages need to share the same domain set as a precondition.

NOTE 2 The result may have a different range type, see Subclause 7.2.5. NOTE 2 The idea is that for each operation available on the range type, a corresponding coverage operation is provided ("induced from the range type operation") [1] [2].

Example Adding two RGB images will apply the "+" operation to each pixel, and within a pixel to each range field in turn.

7.1.14 unaryInducedExpr

The **unaryInducedExpr** element specifies a unary induced operation, i.e., an operation where only one coverage argument occurs.

NOTE The term "unary" refers only to coverage arguments; it is well possible that further non-coverage parameters occur, such as an integer number indicating the shift distance in a bit() operation.

Requirement 20
A **unaryInducedExpr shall** be either a **unaryArithmeticExpr**, or **trigonometricExpr**, or **exponentialExpr** (in which case it evaluates to a coverage with a numeric range type; see Subclauses 7.1.15, 7.1.16, 7.1.17), a **booleanExpr** (in which case it evaluates to a Boolean expression; see Subclause 7.1.18), a **castExpr** (in which case it evaluates to a coverage with unchanged values, but another range type; see Subclause 7.1.19), or a **fieldExpr** (in which case a range field selection is performed; see Subclause 7.1.20).

7.1.15 unaryArithmeticExpr

The **unaryArithmeticExpr** element specifies a unary induced arithmetic operation.

Requirement 21
A **unaryArithmeticExpr shall** be defined as below.

Let

c_1 be a **coverageExpr**

Then,

for any **coverageExpr** c_2
where c_2 is one of

C_{plus} = **+** C_1
C_{minus} = **-** C_1
C_{sqrt} = **sqrt(** C_1 **)**
C_{abs} = **abs(** C_1 **)**

C_2 is defined as follows:

Coverage constituent	Changed?
identifier(C_2) = "" (empty string)	**X**
crs(C_2) = *crs*(C_1)	
domain(C_2) = *domain*(C_1)	
indexCrs(C_2) = *indexCrs*(C_1)	
indexDomain(C_2) = *indexDomain*(C_1)	
interpolationSet(C_2) = *interpolationSet*(C_1)	
rangeFieldNames(C_2) = *rangeFieldNames*(C_1)	
for all range fields $r \in$ *rangeFieldNames*(C_2): *rangeFieldType*(C_{plus},r) = *rangeFieldType*(C_1,r), *rangeFieldType*(C_{minus},r) = *rangeFieldType*(C_1,r) if *rangeFieldType*(C_{minus},r) ∈ { char, short, int, long, float, double, complex, complex2 }, *rangeFieldType*(C_{minus},r) = char if *rangeFieldType*(C_1,r) = unsigned char, *rangeFieldType*(C_{minus},r) = short if *rangeFieldType*(C_1,r) = unsigned short, *rangeFieldType*(C_{minus},r) = int if *rangeFieldType*(C_1,r) = unsigned int, *rangeFieldType*(C_{minus},r) = long if *rangeFieldType*(C_1,r) = unsigned long, *rangeFieldType*(C_{sqrt},r) = double if *rangeFieldType*(C_1,r) ∉ { complex, complex2 } and $C_1.r \geq 0$, *rangeFieldType*(C_{sqrt},r) = complex2 otherwise, *rangeFieldType*(C_{abs},r) = *rangeFieldType*(C_1,r) if *rangeFieldType*(C_1,r) ∈ { boolean, unsigned char, unsigned short, unsigned int, unsigned long, float, double }, *rangeFieldType*(C_{abs},r) = unsigned char	**X**

if *rangeFieldType*(C_1,r) = char, *rangeFieldType*(C_{abs},r) = unsigned short if *rangeFieldType*(C_1,r) = short, *rangeFieldType*(C_{abs},r) = unsigned int if *rangeFieldType*(C_1,r) = int, *rangeFieldType*(C_{abs},r) = unsigned long if *rangeFieldType*(C_1,r) = long, *rangeFieldType*(C_{abs},r) = float if *rangeFieldType*(C_1,r) ∈ { float, complex }, *rangeFieldType*(C_{abs},r) = double if *rangeFieldType*(C_1,r) ∈ { double, complex2}	
nullSet(C_2) = {}	**X**
for all p ∈ *domain*(C_2): *value*(C_{plus}, p) = *value*(C_1, p), *value*(C_{minus}, p) = - *value*(C_1, p), *value*(C_{sqrt}, p) = sqrt(*value*(C_1, p)), *value*(C_{abs}, p) = abs(*value*(C_1, p))	**X**
metadata(C_2) = *metadata*(C_1)	

Example The following coverage expression evaluates to a float-type coverage where each range value contains the square root of the sum of the corresponding source coverages' values.

```
sqrt( $c + $d )
```

7.1.16 trigonometricExpr

The **trigonometricExpr** element specifies a unary induced trigonometric operation.

Requirement 22
A **trigonometricExpr shall** be defined as below.

Let

C_1 be a **coverageExpr**

Then,

for any **coverageExpr** C_2
where C_2 is one of
C_{sin} = **sin(** C_1 **)**
C_{cos} = **cos(** C_1 **)**
C_{tan} = **tan(** C_1 **)**
C_{sinh} = **sinh(** C_1 **)**
C_{cosh} = **cosh(** C_1 **)**
C_{arcsin} = **arcsin(** C_1 **)**

C_{arccos} = **arccos(** C_1 **)**
C_{arctan} = **arctan(** C_1 **)**

C_2 is defined as follows:

Coverage constituent	Changed?
identifier(C_2) = "" (empty string)	**X**
crs(C_2) = *crs*(C_1)	
domain(C_2) = *domain*(C_1)	
indexCrs(C_2) = *indexCrs*(C_1)	
indexDomain(C_2) = *indexDomain*(C_1)	
interpolationSet(C_2) = *interpolationSet*(C_1)	
rangeFieldNames (C_2) = *rangeFieldNames* (C_1)	
for all fields $r \in$ *rangeFieldNames*(C_2): *rangeFieldType*(C_2,r) = complex2 if *rangeFieldType*(C_1,r) $\in$ { complex, complex2 }, *rangeFieldType*(C_2,r) = double otherwise	**X**
nullSet(C_2) = {}	**X**
for all $p \in$ *domain*(C_2): *value*(C_{sin},p) = sin(*value*(C_1,p)) *value*(C_{cos},p) = cos(*value*(C_1,p)) *value*(C_{tan},p) = tan(*value*(C_1,p)) *value*(C_{sinh},p) = sinh(*value*(C_1,p)) *value*(C_{cosh},p) = cosh(*value*(C_1,p)) *value*(C_{arcsin},p) = arcsin(*value*(C_1,p)) *value*(C_{arccos},p) = arccos(*value*(C_1,p)) *value*(C_{arctan},p) = arctan(*value*(C_1,p))	**X**
metadata(C_2) = *metadata*(C_1)	

Example The following expression replaces all values of the coverage addressed by `$c` with their sine:

```
sin( $c )
```

Example To enforce a complex result for real-valued arguments the input coverage can be cast to complex:

```
arcsin( (complex) $c )
```

7.1.17 exponentialExpr

The **exponentialExpr** element specifies a unary induced exponential operation.

Requirement 23
An **exponentialExpr shall** be defined as below.

Let

c_1 be a **coverageExpr**,
p be a **floatConstant**

Then,

for any **coverageExpr** c_2
where c_2 is one of
c_{exp} = **exp(** c_1 **)**
c_{log} = **log(** c_1 **)**
c_{ln} = **ln(** c_1 **)**
c_{pow} = **pow(** c_1, p **)**

c_2 is defined as follows:

Coverage constituent	Changed?
identifier(c_2) = "" (empty string)	**X**
crs(c_2) = *crs*(c_1)	
domain(c_2) = *domain*(c_1)	
indexCrs(c_2) = *indexCrs*(c_1)	
indexDomain(c_2) = *indexDomain*(c_1)	
interpolationSet(c_2) = *interpolationSet*(c_1)	
for all fields $r \in$ *rangeFieldNames*(c_2): *rangeFieldType*(c_2,r) = complex2 if *rangeFieldType*(c_1,r) $\in$ { complex, complex2 }, *rangeFieldType*(c_2,r) = double otherwise	**X**
nullSet(c_2) = *nullSet*(c_1)	
for all $p \in$ *domain*(c_2): *value*(c_{exp}, p) = exp(*value*(c_1,p))	**X**

value(C_{log} , p) = log(*value*(C_1,p)) *value*(C_{ln} , p) = ln(*value*(C_1,p)) *value*(C_{pow}, p) = *value*(C_1,p)^p	
metadata(C_2) = *metadata*(C_1)	

Example The following expression replaces all (nonnegative numeric) values of coverage C with their natural logarithm:

```
ln( $c )
```

7.1.18 booleanExpr

The **booleanExpr** element specifies a unary induced Boolean operation.

Requirement 24
A **booleanExpr shall** be defined as below.

Let

C_1 be a **coverageExpr**,
n be a positive integer number**.**

Then,

for any **coverageExpr** C_2
where C_2 is one of
C_{not} = **not** C_1
C_{bit} = **bit(** C_1 , n **)**
where n is an expression evaluating to a nonnegative integer value

C_2 is defined as follows:

Coverage constituent	Changed?
identifier(C_2) = "" (empty string)	**X**
crs(C_2) = *crs*(C_1)	
domain(C_2) = *domain*(C_1)	
indexCrs(C_2) = *indexCrs*(C_1)	
indexDomain(C_2) = *indexDomain*(C_1)	
interpolationSet(C_2) = *interpolationSet*(C_1)	
rangeFieldNames (C_2) = *rangeFieldNames* (C_1)	

for all fields $r \in$ *rangeFieldNames*(c_2): *rangefieldType*(c_2,r) = boolean	**X**
nullSet(c_2) = {}	**X**
for all $p \in$ *domain*(c_2): *value*(c_{not} , p) = not(*value*(c_1,p)) *value*(c_{bit}, p) = (*value*(c_1,p) >> n) mod 2	
metadata(c_2) = *metadata*(c_1)	

Example The following expression inverts all (assumed: Boolean) range field values of coverage `$c`:

```
not $c
```

NOTE The operation `bit(a,b)` extracts bit position b (assuming a binary representation) from integer number a and shifts the resulting bit value to bit position 0. Hence, the resulting value is either 0 or 1.

7.1.19 castExpr

The **castExpr** element specifies a unary induced cast operation, that is: to change the range type of the coverage while leaving all other properties unchanged. All range components are converted to this same type.

NOTE Depending on the input and output types result possibly may suffer from a loss of accuracy through data type conversion.

Requirement 25
A **castExpr shall** be defined as below.

Let

c_1 be a **coverageExpr**,
t be a range field type name.

Then,

for any **coverageExpr** c_2
where
c_2 = **(** t **)** c_1

c_2 is defined as follows:

Coverage constituent	Changed?
identifier(c_2) = "" (empty string)	**X**
crs(c_2) = *crs*(c_1)	

domain(c_2) = *domain*(c_1)	
indexCrs(c_2) = *indexCrs*(c_1)	
indexDomain(c_2) = *indexDomain*(c_1)	
interpolationSet(c_2) = *interpolationSet*(c_1)	
rangeFieldNames (c_2) = *rangeFieldNames* (c_1)	
for all fields $r \in$ *rangeFieldNames*(c_2): *rangeFieldType*(C_2,r) = t	**X**
nullSet(c_2) = {}	**X**
for all $p \in$ *domain*(c_2): *value*(c_2 , p) = (t) *value*(c_1,p)	**X**
metadata(c_2) = *metadata*(c_1)	

Example the result range type of the following expression will be char, i.e., 8 bit:

```
(char) ( $c / 2 )
```

7.1.20 fieldExpr

The **fieldExpr** element specifies a unary induced field selection operation. Fields are selected by their name, in accordance with the WCS range field subsetting operation.

NOTE Due to the current restriction to atomic range fields, the result of a field selection has atomic values too.

Requirement 26
A **fieldExpr shall** be defined as below.

Let

c_1 be a **coverageExpr**,
f be a **fieldName** appearing in *rangeFieldNames*(c_1),
i be an **integer** with $0 \le i < |$*rangeFieldNames*(c_1)$|$.

Then,

for any **coverageExpr** c_2
where c_2 is one of:

$c_{2,f} = c_1 \ . \ f$
$c_{2,i} = c_1 \ . \ i$

c_2 is defined as follows:

Coverage constituent	Changed?
identifier(C_2) = "" (empty string)	**X**
crs(C_2) = *crs*(C_1)	
domain(C_2) = *domain*(C_1)	
indexCrs(C_2) = *indexCrs*(C_1)	
indexDomain(C_2) = *indexDomain*(C_1)	
interpolationSet(C_2) = *interpolationSet*(C_1)	
rangeFieldNames (C_2) = (f), the sequence containing only f	**X**
rangeFieldType(C_2,f) = *rangeFieldType*(C_1,f)	**X**
nullSet(C_2) = {}	**X**
for all $p \in$ *domain*(C_2): *value*($C_{2,f},p$) = *value*(C_1 . f,p) *value*($C_{2,i},p$) = *value*(C_1 . g,p) where g is the i^{th} field in *rangeFieldNames*(C_1)	**X**
metadata(C_2) = *metadata*(C_1)	

Example Let `$c` refer to a coverage with range type integer. Then the following request snippet describes a single-field, integer-type coverage where each grid point value contains the difference between red and green band:

```
$c.red - $c.green
```

Requirement 27
In a **fieldExpr** C . f where |*rangeFieldNames*(C)|=1 **shall** be identical to C.

Example Let `$c` refer to a coverage with range component `red`, `$d` a single-component range type (say, a panchromatic satellite scene). Assuming both are compatible the following expression is admissible:

```
$c.red - $d
```

7.1.21 binaryInducedExpr

The **binaryInducedExpr** element specifies a binary induced operation, i.e., an operation involving two coverage-valued arguments.

Requirement 28
Both participating coverages **shall** have the same number of range components; otherwise the server **shall** respond with a service exception.

Requirement 29
The operand coverage range types **shall** be numeric.

Requirement 30
A **binaryExpr shall** be defined as below.

Let

C_1, C_2 be **coverageExpr**s,
S_1, S_2 be **scalarExpr**s,
where
crs(C_1) = *crs*(C_2),
domain(C_1,a) = *domain*(C_2,a),
indexCrs(C_1) = *indexCrs*(C_2),
indexDomain(C_1) = *indexDomain*(C_2),
rangeFieldNames(C_1) = *rangeFieldNames*(C_2),
rangeType(C_1,f) is cast-compatible with *rangeType*(C_2,f) or
rangeType(C_2,f) is cast-compatible with *rangeType*(C_1,f)
for all $f \in$ *rangeFieldNames*(C_1).

Then,

for any **coverageExpr** C_3
where C_3 is one of

C_{plusCC} = C_1 **+** C_2
C_{minCC} = C_1 **-** C_2
C_{multCC} = C_1 ***** C_2
C_{divCC} = C_1 **/** C_2
C_{andCC} = C_1 **and** C_2
C_{orCC} = C_1 **or** C_2
C_{xorCC} = C_1 **xor** C_2
C_{eqCC} = C_1 **=** C_2
C_{ltCC} = C_1 **<** C_2
C_{gtCC} = C_1 **>** C_2
C_{leCC} = C_1 **<=** C_2
C_{geCC} = C_1 **>=** C_2
C_{neCC} = C_1 **!=** C_2
C_{ovlCC} = C_1 **overlay** C_2

C_{plusSC} = S_1 **+** C_2
C_{minSC} = S_1 **-** C_2
C_{multSC} = S_1 ***** C_2
C_{divSC} = S_1 **/** C_2
C_{andSC} = S_1 **and** C_2
C_{orSC} = S_1 **or** C_2
C_{xorSC} = S_1 **xor** C_2
C_{eqSC} = S_1 **=** C_2

C_{ltSC} = S_1 **<** C_2
C_{gtSC} = S_1 **>** C_2
C_{leSC} = S_1 **<=** C_2
C_{geSC} = S_1 **>=** C_2
C_{neSC} = S_1 **!=** C_2
C_{ovlSC} = S_1 **overlay** C_2

C_{plusCS} = C_1 **+** S_2
C_{mincS} = C_1 **-** S_2
C_{multCS} = C_1 ***** S_2
C_{divCS} = C_1 **/** S_2
C_{andCS} = C_1 **and** S_2
C_{orCS} = C_1 **or** S_2
C_{xorCS} = C_1 **xor** S_2
C_{eqCS} = C_1 **=** S_2
C_{ltCS} = C_1 **<** S_2
C_{gtCS} = C_1 **>** S_2
C_{leCS} = C_1 **<=** S_2
C_{geCS} = C_1 **>=** S_2
C_{neCS} = C_1 **!=** S_2
C_{ovlCS} = C_1 **overlay** S_2

C_3 is defined as follows:

Coverage constituent	**Changed?**
identifier(C_3) = "" (empty string)	**X**
crs(C_3) = *crs*(C_1)	
domain(C_3) = *domain*(C_1)	
indexCrs(C_3) = *indexCrs*(C_1)	
indexDomain(C_3) = *indexDomain*(C_1)	
interpolationSet(C_3) = *interpolationSet*(C_1)∩*interpolationSet*(C_2)	**X**
rangeFieldNames (C_3) = *rangeFieldNames* (C_1)	
for all $r \in$ *rangeFieldNames*(C_3): *rangeFieldType*(C_{plusCC}, r) is given by Section 7.2.5 *rangeFieldType*(C_{minCC}, r) is given by Section 7.2.5 *rangeFieldType*(C_{multCC}, r) is given by Section 7.2.5 *rangeFieldType*(C_{divCC}, r) is given by Section 7.2.5 *rangeFieldType*(C_{andCC}, r) = boolean *rangeFieldType*(C_{orCC}, r) = boolean	**X**

rangeFieldType(C_{xorCC}, r) = boolean *rangeFieldType*(C_{eqCC}, r) = boolean *rangeFieldType*(C_{ltCC}, r) = boolean *rangeFieldType*(C_{gtCC}, r) = boolean *rangeFieldType*(C_{leCC}, r) = boolean *rangeFieldType*(C_{geCC}, r) = boolean *rangeFieldType*(C_{neCC}, r) = boolean *rangeFieldType*(C_{ovlCC}, r) = *rangeFieldType*(C_1, r) *rangeFieldType*(C_{plusSC}, r) is given by Section 7.2.5 *rangeFieldType*(C_{minSC}, r) is given by Section 7.2.5 *rangeFieldType*(C_{multSC}, r) is given by Section 7.2.5 *rangeFieldType*(C_{divSC}, r) is given by Section 7.2.5 *rangeFieldType*(C_{andSC}, r) = boolean *rangeFieldType*(C_{orSC}, r) = boolean *rangeFieldType*(C_{xorSC}, r) = boolean *rangeFieldType*(C_{eqSC}, r) = boolean *rangeFieldType*(C_{ltSC}, r) = boolean *rangeFieldType*(C_{gtSC}, r) = boolean *rangeFieldType*(C_{leSC}, r) = boolean *rangeFieldType*(C_{geSC}, r) = boolean *rangeFieldType*(C_{neSC}, r) = boolean *rangeFieldType*(C_{ovlSC}, r) = *rangeFieldType*(C_2) *rangeFieldType*(C_{plusCS}, r) is determined by Table 4 *rangeFieldType*(C_{minCS}, r) is determined by Table 4 *rangeFieldType*(C_{multCS}, r) is determined by Table 4 *rangeFieldType*(C_{divCS}, r) is determined by Table 4 *rangeFieldType*(C_{andCS}, r) = boolean *rangeFieldType*(C_{orCS}, r) = boolean *rangeFieldType*(C_{xorCS}, r) = boolean *rangeFieldType*(C_{eqCS}, r) = boolean *rangeFieldType*(C_{ltCS}, r) = boolean *rangeFieldType*(C_{gtCS}, r) = boolean *rangeFieldType*(C_{leCS}, r) = boolean *rangeFieldType*(C_{geCS}, r) = boolean *rangeFieldType*(C_{neCS}, r) = boolean *rangeFieldType*(C_{ovlCS}, r) = boolean	
nullSet(C_3) = {}	**X**
for all $p \in$ *domain*(C_3): *value*(C_{plusCC}, p) = *value*(C_1, p) + *value*(C_2, p) *value*(C_{minCC}, p) = *value*(C_1, p) - *value*(C_2, p) *value*(C_{multCC}, p)= *value*(C_1, p) * *value*(C_2, p) *value*(C_{divCC}, p) = *value*(C_1, p) / *value*(C_2, p) *value*(C_{andCC}, p) = *value*(C_1, p) and *value*(C_2, p)	**X**

value(c_{orCC}, p) = *value*(c_1, p) or *value*(c_2, p) *value*(c_{xorCC}, p) = *value*(c_1, p) xor *value*(c_2, p) *value*(c_{eqCC}, p) = *value*(c_1, p) = *value*(c_2, p) *value*(c_{ltCC}, p) = *value*(c_1, p) < *value*(c_2, p) *value*(c_{gtCC}, p) = *value*(c_1, p) > *value*(c_2, p) *value*(c_{leCC}, p) = *value*(c_1, p) <= *value*(c_2, p) *value*(c_{geCC}, p) = *value*(c_1, p) >= *value*(c_2, p) *value*(c_{neCC}, p) = *value*(c_1, p) != *value*(c_2, p) *value*(c_{ovlCC}, p) = *value*(c_2, p) if *value*(c_1, p)=0 *value*(c_1, p) otherwise *value*(c_{plusSC}, p) = s_1 + *value*(c_2, p) *value*(c_{minSC}, p) = s_1 - *value*(c_2, p) *value*(c_{multSC}, p) = s_1 * *value*(c_2, p) *value*(c_{divSC}, p) = s_1 / *value*(c_2, p) *value*(c_{andSC}, p) = s_1 and *value*(c_2, p) *value*(c_{orSC}, p) = s_1 or *value*(c_2, p) *value*(c_{xorSC}, p) = s_1 xor *value*(c_2, p) *value*(c_{eqSC}, p) = s_1 = *value*(c_2, p) *value*(c_{ltSC}, p) = s_1 < *value*(c_2, p) *value*(c_{gtSC}, p) = s_1 > *value*(c_2, p) *value*(c_{leSC}, p) = s_1 <= *value*(c_2, p) *value*(c_{geSC}, p) = s_1 >= *value*(c_2, p) *value*(c_{neSC}, p) = s_1 != *value*(c_2, p) *value*(c_{ovlSC}, p) = *value*(c_2, p) if s_1=0 s_1 otherwise *value*(c_{plusCS}, p) = *value*(c_1, p) + s_2 value(c_{minCS}, p) = *value*(c_1, p) - s_2 *value*(c_{multCS}, p) = *value*(c_1, p) * s_2 *value*(c_{divCS}, p) = *value*(c_1, p) / s_2 *value*(c_{andCS}, p) = *value*(c_1, p) and s_2 *value*(c_{orCS}, p) = *value*(c_1, p) or s_2 *value*(c_{xorCS}, p) = *value*(c_1, p) xor s_2 *value*(c_{eqCS}, p) = *value*(c_1, p) = s_2 *value*(c_{ltCS}, p) = *value*(c_1, p) < s_2 *value*(c_{gtCS}, p) = *value*(c_1, p) > s_2 *value*(c_{leCS}, p) = *value*(c_1, p) <= s_2 *value*(c_{geCS}, p) = *value*(c_1, p) >= s_2 *value*(c_{neCS}, p) = *value*(c_1, p) != s_2 *value*(c_{ovlCS}, p) = s_2 if *value*(c_1, p)=0 *value*(c_1, p) otherwise Whenever necessary, appropriate cast operations are performed on the values prior to performing the binary value operation (cf. Subclause 7.2.5).	
metadata(c_3) = *metadata*(c_1)	

Example The following expression describes a coverage composed of the sum of the red, green, and blue fields of the coverage referred to by `$c`:

```
$c.red + $c.green + $c.blue
```

7.1.22 rangeConstructorExpr

The **rangeConstructorExpr**, an n-ary induced operation, allows to build coverages with compound range structures. To this end, coverage range field expressions enumerated are combined into one coverage.

All input coverages must match wrt. domains and CRSs. An input coverage range field may be listed more than once.

Requirement 31
The names of the range fields generated by the operation **shall** be given by the names pre-fixed to each component expression.

Requirement 32
A **rangeConstructorExpr shall** be defined as below.

Let

n be an **integer** with $n \geq 1$,
$C_1, \ldots, C_n$ be **coverageExpr**s with $|rangeFieldNames(C_i)|=1$,
$f_1, \ldots, f_n$ be **fieldName**s
where, for $1 \leq i,j \leq n$,
$crs(C_i) = crs(C_j)$,
$domain(C_i) = domain(C_j)$
$indexCrs(C_i) = indexCrs(C_j)$,
$indexDomain(C_i) = indexDomain(C_j)$.

Then,

for any **coverageExpr** C'
where C' is one of
$C'_a = \{ f_1 : C_1 ; \ldots ; f_n : C_n \}$
$C'_b =$ **struct** $\{ f_1 : C_1 ; \ldots ; f_n : C_n \}$

C' is defined as follows:

Coverage constituent	Changed?
identifier(C') = "" (empty string)	**X**
$crs(C_2) = crs(C_1)$	

domain(c_2) = *domain*(c_1)	
indexCrs(c_2) = *indexCrs*(c_1)	
indexDomain(c_2) = *indexDomain*(c_1)	
interpolationSet(c_2) = *interpolationSet*(c_1)	
rangeFieldNames(c_2) = (f_1, …, f_n)	**X**
for all range fields f_i: *rangeFieldType*(c_2,f_i) = *rangeFieldType*(c_i)	**X**
nullSet(c') = *nullSet*(c_1) × … × *nullSet*(c_n)	**X**
for all $p \in$ *domain*(c'): *value*($c'.f_i$,p) = *value*(c_i,p)	**X**
metadata(c_2) = *metadata*(c_1)	

Example 1: The expression below does a false color encoding by combining near-infrared, red, and green bands into a 3-band image of 8-bit channels each, which can be visually interpreted as RGB:

```
struct
{ red:   (char) $1.nir;
  green: (char) $1.red;
  blue:  (char) $1.green
}
```

Example 2: The following expression transforms a greyscale image referred to by variable `$g` containing a range field `panchromatic` into an RGB-structured image:

```
struct
{ red:   $g.panchromatic;
  green: $g.panchromatic;
  blue:  $g.panchromatic
}
```

7.1.23 subsetExpr

The **subsetExpr** element specifies spatial and temporal domain subsetting. It encompasses spatial and temporal trimming (i.e., constraining the result coverage domain to a subinterval, Subclause 7.1.24), slicing (i.e., cutting out a hyperplane from a coverage, Subclause 7.1.26), extending (Subclause 7.1.25), and scaling (Subclause 7.1.27) of a coverage expression.

Requirement 33
A **subsetExpr shall** be either a **trimmingExpr** (Subclause 7.1.24) or a **slicingExpr** (Subclause 7.1.26) or an **extendExpr** (Subclause 7.1.25) or a **scalingExpr** (Subclause 7.1.27).

All of the **subsetExpr** elements allow to make use of coordinate reference systems other than a coverage's image CRS. A coverage's individual mapping from some supported CRS coordinates to its Index CRS coordinates does not need to be disclosed by the server, hence coordinate transformation **should** be considered a "black box" by the client.

NOTE 1 The special case that subsetting leads to a single point remaining still resembles a coverage by definition; this coverage is viewed as being of dimension 0.

NOTE 2 Range subsetting is accomplished via the unary induced **fieldExpr** (cf. Subclause 7.1.20).

7.1.24 trimExpr

The **trimExpr** element extracts a subset from a given coverage expression along the dimension indicated, specified by a lower and upper bound for each dimension affected. Interval limits can be expressed in the coverage CRS or any other CRS explicitly indicated, as long as a transformation to the coverage CRS exists.

Requirement 34
In a **trimExpr** lower as well as upper limits **shall** lie inside the coverage's domain.

For syntactic convenience, both array-style addressing using brackets and function-style syntax are provided; both are equivalent in semantics.

Requirement 35
A **trimExpr shall** be defined as below.

Let

c_1 be a **coverageExpr**,
n be an **integer** with $0 \le n$,
$(lo_1:hi_1),\ldots,(lo_n:hi_n)$ be **dimensionIntervalExpr**s with $lo_i \le hi_i$ for $1 \le i \le n$.

Then,

for any **coverageExpr** c_2
where c_2 is one of

c_{bracket} = c_1 **[** p_1, …, p_n **]**

~~c_{func} = **trim (c_1, { p_1, …, p_n })**~~ (deprecated)

with

p_i is one of

$p_{\text{img},i}$ = a_i **(** lo_i **:** hi_i **)**

$p_{\text{crs},i}$ = a_i : crs_i **(** lo_i **:** hi_i **)**

where each interval is within the coverage's bounds, as expressed by interval and axis (possibly reprojected from a CRS indicated).

c_2 is defined as follows:

Coverage constituent	Changed?

identifier(C_2) = "" (empty string)	**X**
crs(C_2) = *crs*(C_1)	
domain(C_2) = *domain*(C_1) reduced to extent (lo_i:hi_i) for any domain axis a_i (reprojected from crs_i into the coverage Native CRS if crs_i is present), and with domain extent properly adjusted for any index axis a_i present in the trim list	
indexCrs(C_2) = *indexCrs*(C_1)	
indexDomain(C_2) = *indexDomain*(C_1) reduced to extent (lo_i:hi_i) for any index axis a_i (no crs_i allowed in this context), and with index extent properly adjusted for any domain axis a_i present in the trim list	**X**
interpolationSet(C_2) = *interpolationSet*(C_1)	
rangeFieldNames(C_2) = *rangeFieldNames*(C_1)	
for all $r \in$ *rangeFieldNames*(C_2): *rangeFieldType*(C_2,r) = *rangeFieldType*(C_i,r)	
nullSet(C_2) = *nullSet*(C_1)	
for all $p \in$ *domain*(C_2): *value*(C_2, p) = *value*(C_1, p)	**X**
metadata(C_2) = *metadata*(C_1)	

Example The following are syntactically valid, equivalent trim expressions:

```
$C[ Long (-120: -80), Lat (-10: +10) ]
```

7.1.25 extendExpr

The **extendExpr** element extends a coverage to the bounding box indicated. The new grid points are filled with one of the coverage's null values.

Requirement 36
In an **extendExpr** if the coverage's null value set is empty then the server **shall** throw an exception.

There is no restriction on the position and size of the new bounding box; in particular, it does not need to lie outside the coverage; it may intersect with the coverage; it may lie completely inside the coverage; it may not intersect the coverage at all (in which case a coverage completely filled with null values will be generated).

NOTE In this sense the **extendExpr** is a generalization of the **trimExpr**; still the **trimExpr** should be used whenever the application needs to be sure that a proper subsetting has to take place.

Requirement 37
An **extendExpr shall** be defined as below.

Let

C_1 be a **coverageExpr**,
n be an **integer** with $0 \le n$,
$a_1, \ldots, a_n$ be pairwise distinct **axisName**s with $a_i \in \text{axisNameSet}(C_1)$ for $1 \le i \le n$,
$crs_1, \ldots, crs_n$ be **crsName**s with $crs_i \in \text{crsList}(C_1)$ for $1 \le i \le n$,
$(lo_1:hi_1), \ldots, (lo_n:hi_n)$ be **dimensionIntervalExpr**s with $lo_i \le hi_i$ for $1 \le i \le n$.

Then,

for any **coverageExpr** C_2
where
C_2 = **extend (C_1, { p_1, …, p_n })**
with
p_i is one of
$p_{img,i}$ = a_i **(lo_i : hi_i)**
$p_{crs,i}$ = a_i : crs_i **(lo_i : hi_i)**

C_2 is defined as follows:

Coverage constituent	Changed?
identifier(C_2) = "" (empty string)	**X**
crs(C_2) = *crs*(C_1)	
domain(C_2) = *domain*(C_1) adjusted to extent (lo_i:hi_i) for any domain axis a_i (reprojected from crs_i into the coverage Native CRS if crs_i is present), and with domain extent properly adjusted for any index axis a_i present in the trim list	**X**
indexCrs(C_2) = *indexCrs*(C_1)	
indexDomain(C_2) = *indexDomain*(C_1) adjusted to extent (lo_i:hi_i) for any index axis a_i (no crs_i allowed in this context), and with index extent properly adjusted for any domain axis a_i present in the trim list	**X**
interpolationSet(C_2) = *interpolationSet*(C_1)	
rangeFieldNames(C_2) = *rangeFieldNames*(C_1)	
for all range fields $r \in$ *rangeFieldNames*(C_2): *rangeFieldType*(C_2,r) = *rangeFieldType*(C_1,r)	

nullSet(C_2) = *nullSet*(C_1)	
for all $p \in$ *domain*(C_2): *value*(C_2, p) = *value*(C_1,p) for $p \in$ *domain*(C_1) *value*(C_2, p) = n otherwise, for some $n \in$ *nullSet*(C_1)$\neq\varnothing$	**X**
metadata(C_2) = *metadata*(C_1)	

NOTE A server **may** decide to restrict the CRSs available on the result, as not all CRSs may be technically appropriate any more.

Example The following is a syntactically valid extend expression:

```
extend( $c, { x ( -200 : +200 ) } )
```

7.1.26 sliceExpr

The **sliceExpr** element extracts a spatial slice (i.e., a hyperplane) from a given coverage expression along one of its dimensions, specified by one or more slicing dimensions and a slicing position thereon. For each slicing dimension indicated, the resulting coverage has a dimension reduced by 1; its dimensions are the dimensions of the original coverage, in the same sequence, with the section dimension being removed from the list. CRSs / axes not used by any of the remaining dimensions are removed from the coverage's CRS set.

Requirement 38
In a **sliceExpr** the slicing coordinates **shall** lie inside the coverage's domain.

For syntactic convenience, both array-style addressing using brackets and function-style syntax are provided; both are equivalent in semantics.

Requirement 39
A **sliceExpr shall** be defined as below.

Let

C_1 be a **coverageExpr**,
n be an **integer** with $0 \le n$,
$a_1,\ldots,a_n$ be pairwise distinct **axisName**s with $a_i \in$ axisNameSet(C_1) for $1 \le i \le n$,
$s_1,\ldots,s_n$ be **axisPoint**s for $1 \le i \le n$.

Then,

for any **coverageExpr** C_2
where C_2 is one of
C_{bracket} = C_1 **[** S_1, …, S_n **]**
~~C_{func} = **slice(** C_1, **{** S_1, …, S_n **})**~~ (deprecated)
with

S_i is one of
$S_{img,i}$ = a_i (s_i)
$S_{crs,i}$ = a_i : crs_i (s_i)

C_2 is defined as follows:

Coverage constituent	Changed?
identifier(C_2) = "" (empty string)	**X**
crs(C_2) = *crs*(C_1) projected to the axes remaining	**X**
domain(C_2) = *domain*(C_1) projected to *crs*(C_2)	**X**
indexCrs(C_2) = m-D Index CRS where m=*dimension*(C_1)-n	
indexDomain(C_2) = *indexDomain*(C_1) projected to the axes remaining in *indexCrs*(C_2)	
interpolationSet(C_2) = *interpolationSet*(C_1)	
rangeFieldNames(C_2) = *rangeFieldNames*(C_1)	
for all $r \in$ *rangeFieldNames*(C_2): *rangeFieldType*(C_2,r) = *rangeFieldType*(C_1,r)	
nullSet(C_2) = *nullSet*(C_1)	
for all $p \in$ *domain*(C_1) : *value*(C_2, p) = *value*(C_1,p') where p' is the projection of p to *crs*(C_2)	
metadata(C_2) = *metadata*(C_1)	

Example The following are syntactically valid, equivalent slice expressions:

```
$c[ Lat ( 120 ) ]
```

The slicing axes can be taken from both the coverage's CRS or grid CRS (its underlying Index CRS), also in a mixed fashion, however every axis may be used only once in a slicing expression, and the "twin" axis cannot appear simultaneously in the same slicing operation.

Example Let a 3D datacube be given with axes *Lat*, *Long*, and *date* together with the corresponding grid index axes *i*, *j*, and *k*. Then, slicing in *Lat*, *Long*, and *k* is possible in one operation.

Using a grid index axis in a slicing operation is only applicable for coverages where domain and grid axes names are disjoint so that no ambiguity occurs.

Requirement 40
In a **sliceExpr** axes from the coverage's index CRS **shall** be used only in case the same name is not used for any axis in the CRS of the coverage.

Example A datacube C with axes (*k*,*Lat*,*Long*) having grid index axes (*i*,*j*,*k*) cannot be sliced in *k*.

7.1.27 scaleExpr

The **scaleExpr** element reduces resolution while leaving the geographic extent unchanged. There are three different variants:

- Scaling to a target extent (i.e., number of grid points along each axis selected)
- Scaling by factors applied individually to axes selected
- Scaling by a factor applied to all axes.

NOTE Scaling regularly involves range interpolation, hence numerical effects have to be expected.

Requirement 41
A **scaleExpr1 shall** be defined as below.

Let

C_1 be a **coverageExpr** with only index and regular grid axes,
m, n be **integer**s with $0 \le m$ and $0 \le n$,
$a_1, \ldots, a_m$ be pairwise distinct **axisName**s with $a_i \in$ *indexCrs*(C_1) for $1 \le i \le m$,
I_i be **intervalExpr**s for $1 \le i \le m$ which evaluate to pairs lo_i, hi_i with $lo_i \le hi_i$,
$im \in$ *interpolationSet*(C_1) be an **interpolationMethod**.

Then,

For any **coverageExpr** C_2,
where

C_2 = **scale (C_1, { a_1 (I_1), …, a_m (I_m) } [, im])**

C_2 is defined as follows:

Coverage constituent	Changed?
identifier(C_2) = "" (empty string)	**X**
crs(C_2) = *crs*(C_1)	
domain(C_2) = *domain*(C_1)	
indexCrs(C_2) = *indexCrs*(C_1)	

indexDomain(C_2) = X_1 ×…× X_d where X_i = I_i for axis a_i	**X**
interpolationSet(C_2) = *interpolationSet*(C_1)	
rangeFieldNames(C_2) = *rangeFieldNames*(C_1)	
for all r ∈ *rangeFieldNames*(C_2): *rangeFieldType*(C_2,r) = *rangeFieldType*(C_1,r)	
nullSet(C_2) = *nullSet*(C_1)	
for all p ∈ *domain*(C_2): *value*(C_2, p) is obtained by rescaling the coverage grid along dimensions a_i such that the coverage's extent along dimension a_i is set to (lo_i : hi_i), expressed in the coverage's Index CRS; all other dimensions remain unaffected. If im is specified then this method is used for interpolation, otherwise it is system-dependent.	**X**
metadata(C_2) = *metadata*(C_1)	

Example The following expression performs x/y scaling of some coverage referenced by variable *$C* using interpolation type `cubic` in both `x` and `y` dimension. Note that *$C* might have other axes, such as time, which would remain unaffected.

```
scale( $C, { x ( 100 : 200 ) , y ( 300 : 400 )  }, cubic )
```

Requirement 42
A **scaleExpr2 shall** be defined as below.

Let

C_1 be a **coverageExpr** with only index and regular grid axes,
m, n be **integer**s with $0 \le m$ and $0 \le n$,
$a_1, \ldots, a_m$ be pairwise distinct **axisName**s with $a_i \in$ *indexCrs*(C_1) for $1 \le i \le m$,
I_i be **indexExpr**s for $1 \le i \le m$,
$im \in$ *interpolationSet*(C_1) be an **interpolationMethod**.

Then,

For any **coverageExpr** C_2,
where

C_2 = **scale (C_1, { a_1 (I_1), …, a_m (I_m) } [, im])**

C_2 is defined as follows:

Coverage constituent	Changed?
identifier(C_2) = "" (empty string)	**X**
crs(C_2) = *crs*(C_1)	
domain(C_2) = *domain*(C_1)	
indexCrs(C_2) = *indexCrs*(C_1)	
indexDomain(C_2) = $X'_1 \times \ldots \times X'_d$ where $X'_i = (lo_i/I_i, hi_i/I_i)$ for axis a_i where $X_i = (lo_i, hi_i)$ is the extent in *indexDomain*(C_1) and is unchanged over *indexDomain*(C_1) otherwise	**X**
interpolationSet(C_2) = *interpolationSet*(C_1)	
rangeFieldNames(C_2) = *rangeFieldNames*(C_1)	
for all $r \in$ *rangeFieldNames*(C_2): *rangeFieldType*(C_2,r) = *rangeFieldType*(C_1,r)	
nullSet(C_2) = *nullSet*(C_1)	
for all $p \in$ *domain*(C_2): *value*(C_2, p) is obtained by rescaling the coverage grid along dimensions a_i such that the coverage's extent along dimension a_i is set to (lo_i : hi_i), expressed in the coverage's Index CRS; all other dimensions remain unaffected. If im is specified then this method is used for interpolation, otherwise it is system-dependent.	**X**
metadata(C_2) = *metadata*(C_1)	

Requirement 43
A **scaleExpr2 shall** be defined as below.

Let

C_1 be a **coverageExpr** with only index and regular grid axes,
s be a **scalarExpr**,
$im \in$ *interpolationSet*(C_1) be an **interpolationMethod**.

Then,

For any **coverageExpr** C_2,
where

C_2 = **scale (** C_1, s [, im] **)**

C_2 is defined as follows:

Coverage constituent	Changed?
identifier(C_2) = "" (empty string)	**X**
crs(C_2) = *crs*(C_1)	
domain(C_2) = *domain*(C_1)	
indexCrs(C_2) = *indexCrs*(C_1)	
indexDomain(C_2) = $X'_1 \times \ldots \times X'_d$ where $X'_i = (lo_i/s, hi_i/s)$ for all axes	**X**
interpolationSet(C_2) = *interpolationSet*(C_1)	
rangeFieldNames(C_2) = *rangeFieldNames*(C_1)	
for all $r \in$ *rangeFieldNames*(C_2): *rangeFieldType*(C_2,r) = *rangeFieldType*(C_1,r)	
nullSet(C_2) = *nullSet*(C_1)	
for all $p \in$ *domain*(C_2): *value*(C_2, p) is obtained by rescaling the coverage grid along dimensions a_i such that the coverage's extent along dimension a_i is set to (lo_i : hi_i), expressed in the coverage's Index CRS; all other dimensions remain unaffected. If im is specified then this method is used for interpolation, otherwise it is system-dependent.	**X**
metadata(C_2) = *metadata*(C_1)	

7.1.28 crsTransformExpr

The **crsTransformExpr** element performs reprojection of a coverage from its Native CRS into another one; the dimension of the coverage as well as the axis types (such as regular vs. irregular) remains unchanged whereas axes and range values generally change. For the interpolation and resampling which usually is incurred the interpolation method to be applied can be indicated optionally.

NOTE 1 This changes the range values (e.g., pixel radiometry).

NOTE 2 A service may refuse to accept some CRS combinations.

Requirement 44
A **crsTransformExpr shall** be defined as below.

Let

C_1 be a **coverageExpr**,
`crs` be a **crsName**,
`it` ∈ *interpolationSet*(C_1) be an **interpolationMethod**.

Then,

for any **coverageExpr** C_2
where
C_2 is one of:
C_{2a} = **`crsTransform( `** C_1 **`, crs )`**
C_{2b} = **`crsTransform( `** C_1 **`, crs , it )`**

C_2 is defined as follows:

Coverage constituent	Changed?
identifier(C_2) = "" (empty string)	**X**
crs(C_2) = `crs`	**X**
domain(C_2) = *domain*(C_1)	**X**
indexCrs(C_2) = *indexCrs*(C_1)	
indexDomain(C_2) is determined by the reprojection of C_1 to `crs`	**X**
interpolationSet(C_2) = *interpolationSet*(C_1)	
rangeFieldNames(C_2) = *rangeFieldNames*(C_1)	
for all range fields $r \in$ *rangeFieldNames*(C_2): *rangeFieldType*(C_2,r) = *rangeFieldType*(C_1,r)	
nullSet(C_2) = *nullSet*(C_1)	
for all $p \in$ *domain*(C_2): *value*(C_2,p) is obtained by reprojecting coverage C_1 from its Native CRS into CRS `crs`. If some interpolation is prescribed this will be applied whenever interpolation and resampling occurs; otherwise, choice of the interpolation method is implementation dependent.	**X**
metadata(C_2) = *metadata*(C_1)	

Example The following expression transforms coverage `$c` (which is assumed to be 2D) into the CRS identified by EPSG:3035.

```
crsTransform( $c, "EPSG:3035" )
```

7.1.29 coverageConstructorExpr

The **coverageConstructorExpr** element allows creating a d-dimensional coverage for some $d \geq 1$.

The coverage is Cartesian (without any georeference), its Native CRS is identical to its grid CRS. The domain definition consists, for each axis, of a unique axis name plus lower and upper bound of the coverage, expressed in the Index CRS of matching dimension and using integer coordinates.

NOTE Index CRS axes, as per OGC CRS definitions, are named *i*, *j*, *k*, etc.

The coverage's content is defined by an expression which gets evaluated for each direct position of the coverage's domain. Such expressions may contain occurrences of the current direct position coordinates, allowing to define position-dependent expressions (see examples below). The expression's type determines the overall coverage range type.

This coverage has no null values and interpolation methods associated. Finally, metadata are empty.

NOTE This constructor is useful

- whenever the coverage is too large to be described as a constant or
- when the coverage's range values are derived from some other source (such as in the course of a histogram computation, see example below).

Requirement 45
A **coverageConstructorExpr shall** be defined as below.

Let

id be a **stringConstant**,
d be an **integer** with $d>0$,
$axis_i$ be pairwise distinct **axisName**s for $1 \leq i \leq d$[2],
$name_i$ be pairwise distinct **variableName**s for $1 \leq i \leq d$, which additionally, in the request on hand, are not used already as a variable in this expression's scope,
I_i be **intervalExpr**s for $1 \leq i \leq d$ which evaluate to pairs lo_i, hi_i with $lo_i \leq hi_i$,
V be a **scalarExpr** possibly containing occurrences of $name_i$**.**

Then,

For any **coverageExpr** C
where

C = **coverage** id
over $name_1$ $axis_1$ **(** I_1 **),**

[2] In the future, introduction of a GeneralDomain concept is planned for WCS which, among others, will allow an arbitrary number of so-called "abstract axes", i.e., axes without spatio-temporal semantics. More than one dimension of this type will be allowed.

```
                …,
                name_d axis_d ( I_d )
        values  V
```

C is defined as follows:

Coverage constituent	Changed?
identifier(*C*) = *id*	**X**
crs(*C*) = Index*N*D where *N*=*d*	**X**
domain(*C*) = $I_1 \times \ldots \times I_d$	**X**
indexCrs(*C*) = Index*N*D where *N*=*d*	**X**
indexDomain(*C*,*a*) = $I_1 \times \ldots \times I_d$	**X**
interpolationSet(*C*) = {}	**X**
rangeFieldNames(*C*) = { *"band0"* }	**X**
rangeFieldType(*C*,*f*) = type of *V* i.e., the range field's type, which must be one of the atomic types supported by this standard, is the result type of expression *V*	**X**
nullSet(C) = {}	**X**
for all $p = (p_1,\ldots,p_d) \in$ *domain*(*C*): *value*(*C*,*p*) = *V'* where expression *V'* is obtained from expression *V* by substituting all occurrences of $name_i$ by p_i	**X**
metadata(C_2) = "" (empty string)	

Example The expression below represents a 2-D greyscale image with a diagonal shade from white to black (the cast operator forces the floating point division result into an integer):

```
coverage greyshade
over     $pi i ( 0 : 255 ),
         $pj j ( 0 : 255 )
values   (unsigned char) ( $pi + $pj ) / 2
```

Example The expression below computes a 256-bucket histogram over band `b` of some coverage `$C` of unknown domain and dimension:

```
Let $X : =
coverage histogram
over     $bucket i ( 0 : 255 )
values   count( $C.b = $bucket )
```

NOTE 1 To fully exploit the opportunities of CIS 1.1 it is recommended to instead use the modern xWCPS coverage constructor (see Subclause 8.2.5).

NOTE 2 As the axis names of Native and Grid CRS combined are not unambiguous it is not possible to use subsetting and scaling on such a coverage.

7.1.30 coverageConstantExpr

The **coverageConstantExpr** element allows creating a d-dimensional coverage, for some $d \geq 1$, having one range field component with point values immediately given in the expression.

The coverage is Cartesian (without any georeference), its Native CRS is identical to its grid CRS. The domain definition consists, for each axis, of a unique axis name plus lower and upper bound of the coverage, expressed in the Index CRS of matching dimension and using integer coordinates.

NOTE Index CRS axes, as per OGC CRS definitions, are named *i*, *j*, *k*, etc.

The coverage's content is defined by the sequence of values provided. The expression's type determines the overall coverage range type.

This coverage has no null values and interpolation methods associated. Finally, metadata are empty.

Requirement 46
In a **coverageConstantExpr** the range type **shall** be given by the narrowest range type encompassing all (atomic or record) values encountered. If the number of range components or their type turns out incompatible for any two points, or if the number of point values provided does not match with the domain extent specified, then the server **shall** respond with an exception.

This coverage has no other CRS associated beyond the abovementioned image CRS; further, it has no null values and interpolation methods associated. Finally, all other metadata are undefined. To set specific metadata for this new coverage the **setComponentExpr** (Subclause **Error! Reference source not found.**) is available.

NOTE This constructor is useful for supplying a small-sized constant coverage, such as a filter kernel.

Requirement 47
A **coverageConstantExpr shall** be defined as below.

Let

> id be a **stringConstant**,
> d be an **integer** with $d > 0$,
> I_i be **intervalExpr**s for $1 \leq i \leq d$ which evaluate to pairs lo_i, hi_i with $lo_i \leq hi_i$,
> $c_1, \ldots, c_m$ be **constant**s where $m = |indexDomain(C)|$ is the product of all d axis extents and all c_i have a common type supported by this standard**.**

Then,

For any **coverageConstantExpr** C
where

C = **coverage** *id*
over $axis_1$ **(** I_1 **),**
…,
$axis_d$ **(** I_d **)**
values < $c_1,\ldots, c_m$ **>**

where

I_i = lo_i **:** hi_i

C is defined as follows:

Coverage constituent	**Changed?**
identifier(C) = id	**X**
crs(C) = Index*N*D where *N*=d	**X**
domain(C) = $I_1 \times \ldots \times I_d$	**X**
indexCrs(C) = Index*N*D where *N*=d	**X**
indexDomain(C) = $I_1 \times \ldots \times I_d$	**X**
interpolationSet(C) = {}	**X**
rangeFieldNames(C) = *("band0")*	**X**
for $r \in$ *rangeFieldNames*(C): *rangeFieldType*(C,r) = type of c_i i.e., the range field's type is equal to the type of the values provided	**X**
nullSet(C) = {}	**X**
for all $p \in$ *domain*(C): *value*(C, p) is determined by assigning each value c_i in turn to a grid point location, whereby assignment proceeds in row-major order (per dimension from the lowest to the highest coordinate, and loops over the grid points with the first axis listed as outermost loop, the next axis listed as next-to-outermost loop, etc., and the last axis listed as innermost loop).	**X**
metadata(c_2) = "" (empty string)	**X**

Example For a Sobel filter, a 3x3 filter kernel can be provided by the expression below. The range value of matrix element (-1/-1) is 1, the value at position (0/-1) is 2, etc.

```
coverage Sobel3x3
over     i ( -1 : +1 ),
         j ( -1 : +1 )
values   <  1;  2;  1;
            0;  0;  0;
           -1; -2; -1
         >
```

7.1.31 condenseExpr

Requirement 48
A **condenseExprshall** be either a **reduceExpr** (see Subclause7.1.33) or a **generalCondense-Expr** (see Subclause 7.1.32).

It takes a coverage and summarizes its values using some summarization function. The value returned is scalar, i.e.: a single scalar value or a record of values, reflecting the number of the input coverage's range type components.

Requirement 49
In a **condenseExpr** whenever one of the point values ("pixels", etc.) participating in a condense operation is equal to one of the null values of its coverage then the result of the operation **shall** be one of the values in the coverage's null value set.

7.1.32 generalCondenseExpr

The general **generalCondenseExpr** consolidates the grid point values of a coverage along selected dimensions to a scalar value based on the condensing operation indicated. It iterates over a given domain while combining the result values of the **scalarExpr**s through the **condenseOpType** indicated. Admissible **condenseOpType**s are the binary operations `+`, `*`, `max`, `min`, `and`, and `or`.

Requirement 50
A **generalCondenseExpr shall** be defined as below.

Let

op be a **condenseOpType**,
n be some **integer** with $n \geq 0$,
d be some **integer** with $d > 0$,
$axis_i$ be **axisName**s for $1 \leq i \leq d$,
$name_i$ be pairwise distinct **variableName**s for $1 \leq i \leq d$ which, in the request on hand, are not used already as a variable in this expression's scope,
I_i be **intervalExpr**s for $1 \leq i \leq d$ which evaluate to pairs lo_i, hi_i with $lo_i \leq hi_i$,
C_j be **coverageExpr**s for $1 \leq j \leq n$,
P be a **booleanExpr** possibly containing occurrences of $name_i$ and C_j,
V be a **scalarExpr** or **coverageExpr** possibly containing occurrences of $name_i$

and C_j,
N be a neutral element of type(V)
where
$1 \le i \le d$.

Then,

For any **scalarExpr** S
where S is one of

```
S′   =  condense op
        over name1 axis1 ( I1 ),
             …,
             named axisd ( Id )
        [ where P ]
        using V

S″   =  condense op
        over axis1 ( I1 ),
             …,
             axisd ( Id )
        [ where P ]
        using V
```

S is constructed as follows (for S'', substitute $name_i$ by $axis_i$):

```
S := N;
for all name1 ∈ {lo1,… ,hi1}
    for all name2 ∈ {lo2,… ,hi2}
        …
            for all named ∈ {lod,… ,hid}
                if (filtering expression P is present)
                then
                    let predicate P′ be obtained from evaluating expression
                    P by substituting all occurrences of namei by its current
                    value where namei occurring in a coordinate position
                    of Cj are coordinates in the index CRS of Cj
                else
                    P' = true;
                fi
                if (P')
                then
                    let V′ be obtained from evaluating expression V
                    by substituting all occurrences of namei by its current
                    value where namei occurring in a coordinate position
                    of Cj are as coordinates in the image CRS of Cj where
                    possible excess dimensions in a coverageExpr are
                    treated as in induced operations;
```

```
                    S := S op value(V')
                fi
            endfor
        …
        endfor
    endfor
return S
```

NOTE 1 Condensers are heavily used, among others, in these two situations:

- To collapse Boolean-valued coverage expressions into scalar Boolean values so that they can be used in predicates.
- In conjunction with the **coverageConstructorExpr** (see Subclause 7.1.29) to phrase high-level imaging, signal processing, and statistical operations.

NOTE 2 The additional expressive power of **condenseExpr** over **reduceExpr** is twofold:

- A WCPS implementation may offer further summarisation functions, as long as these form a monoid, i.e.: they are commutative and associative and have a neutral element.
- The **condenseExpr** gives explicit access to the coordinate values; this makes summarisation considerably more powerful (see example below).

Example For a filter kernel *k*, the condenser must summarise not only over the grid point under inspection, but also some neighbourhood. The following applies a 3x3 filter kernel to band `b` of some coverage *$c* with extent x0…x1/y0…y1; note that the result image is defined to have an *x* and *y* dimension.

```
coverage filteredImage
over     $pi i ( x0 : x1 ),
         $pj j ( y0 : y1 )
values   condense +
         over    $ki i ( -1 : +1 ),
                 $kj j ( -1 : +1 )
         using   $C[ $ki+$pi, $kj+$pj ] * k[ $ki, $kj ]
```

where *k* is a 3x3 matrix like

1	2	1
0	0	0
-1	-2	-1

Starting WCPS 1.1 this example can be also written as:

```
coverage filteredImage
over     x ( x0 : x1 ),
         y ( y0 : y1 )
values   condense +
         over  i ( -1 : +1 ),
               j ( -1 : +1 )
         using $C[ x+i , y+j ] * k[ i, j ]
```

NOTE See **coverageConstantExpr** for a way to specify the *k* matrix in the query.

7.1.33 reduceExpr

A **reduceExpr** element derives a summary value from the coverage passed; in this sense it "reduces" a coverage to a scalar value.

Requirement 51
A **reduceExpr** shall be either an `add`, `avg`, `min`, `max`, `count`, `some`, or `all` operation as per Table 3.

Table 3 – reduceExpr definition via generalCondenseExpr

reduceExpr definition[3]	Meaning

`add($a) =`
`  condense +`
`  over  $p1 D1(indexDomain($a,D1)),`
`  …,`
`  $pd Dd(indexDomain($a,D1)),`
`  using $a[ $p1 , …, $pd ]`
	sum over all points in *$a*

`avg($a) =` `  add($a) / | indexDomain($a) |`	Average of all points in *$a*

`min($a) =`
`  condense min`
`  over  $p1 D1(indexDomain($a,D1)),`
`  …,`
`  $pd Dd(indexDomain($a,D1))`
`  using $a[ $p1 , …, $pd ]`
	Minimum of all points in *$a*

`max($a) =`
`  condense max`
`  over  $p1 D1(indexDomain($a,D1)),`
`  …,`
`  $pd Dd(indexDomain($a,D1))`
`  using $a[ $p1 , …, $pd ]`
	Maximum of all points in *$a*

`count($b) =`
`  condense +`
`  over  $p1 D1(indexDomain($b,D1)),`
`  …,`
`  $pd Dd(indexDomain($b,D1))`
`  where $b[ $p1 , …, $pd ]`
`  using 1`
	Number of points in *$b*

`some($b) =` `  condense or`	is there any point in *$b* bwith

[3] *$a* is assumed to evaluate to a coverage with a single numeric range field, *$b* to a coverage with a single Boolean r**ange field.**

over p1 D1(indexDomain($b,D1)), …, $pd Dd(indexDomain($b,D1)) **using** $b[$p1 , …, $pd]	value true?
all($b) = **condense** and **over** $p1 D1(indexDomain($b,D1)), …, $pd Dd(indexDomain($b,D1)) **using** $b[$p1 , …, $pd]	do all points of *$b* have value true?

7.2 Expression evaluation

This Sublause defines additional rules for *ProcessCoverages* expression evaluation.

7.2.1 Evaluation sequence

Requirement 52
A **processingExpr shall** evaluate coverage expressions from left to right.

7.2.2 Nesting

Requirement 53
A **processingExpr shall** allow nesting all operators, constructors, and functions arbitrarily, provided that each sub-expression's result type matches the required type at the position where the sub-expression occurs. This holds without limitation for all arithmetic, Boolean, String, and coverage-valued expressions.

7.2.3 Parentheses

A **processingExpr** may contain parentheses to enforce a particular evaluation sequence.

Requirement 54
Parentheses enforcing evaluation sequence in a **processingExpr shall** be defined as below.

Let

C_1 and C_2 be **coverageExpr**s

Then,

For any **coverageExpr** C_2
where
C_2 = **(** C_1 **)**

C_2 is defined as yielding the same result as C_1.

Example `$c * ( $c > 0 )`

7.2.4 Operator precedence rules

Requirement 55
In case of ambiguities in the syntactical analysis of a request, operators **shall** have the following precedence (listed in descending strength of binding):

- Range field selection, trimming, slicing
- unary –
- unary arithmetic, trigonometric, and exponential functions
- `*`, `/`
- `+`, `-`
- `<`, `<=`, `>`, `>=`, `!=`, `=`
- `and`
- `or`, `xor`
- `:` (interval constructor), condense, marray
- overlay

In all remaining cases evaluation **shall** be done left to right.

7.2.5 Range type compatibility and extension

A range type t_1 is said to be **cast-compatible** with a range type t_2 iff the following conditions hold:

- Both range types, t_1 and t_2, have the same number of field elements, say d;
- For each range field element position i with $1 \le i \le d$, the ith range field type $f_{1,i}$ of t_1 is **cast-compatible** with the ith range field type $f_{2,i}$ of t_2.

A range field type f_1 is said to be **cast-compatible** with a range field type f_2 iff f_2 can be cast to f_1, whereby **casting** of f_2 to f_1 is defined as looking up f_2 in Table 4 and replacing it by its right-hand neighbour type or, if it is the last type in line, by the first type of the next line. This is repeated until either f_1 is matched, or the end of the Table 4 is reached. Type f_1 can be cast to type f_2 if the casting procedure terminates with finding t_2, otherwise the cast is not possible.

Requirement 56
Extending `boolean` **shall** map `false` to 0 and `true` to 1.

Requirement 57
On both arguments to binary operations type extension **shall** be applied as per Table 4 until both argument types are equal; if such a common type can be found, then this **shall** be the binary operation result type; otherwise, an exception **shall** be thrown.

Example For three single-field coverages `$F`, `$I`, and `$B` with range types `float`, `integer`, and `boolean`, resp., the result type of the following expression is `float`:

```
$F + $I + $B
```

Table 4 – Type extension sequence.

Type extension rules
boolean > char
char > boolean
boolean > unsigned char
unsigned char > boolean
char > short
char > unsigned short
unsigned char > short
unsigned char > unsigned short
short > int
short > unsigned int
unsigned short > int
unsigned short > unsigned int
int > long
int > unsigned long
unsigned int > long
unsigned int > unsigned long
long > float
float > double
float > complex
double > complex2
complex > complex2
complex char > complex short
complex short > complex int
complex int > complex long
complex long > complex2
complex2 > complex long

Requirement 58 The type of each of the operands of a multiplicative operator (+, -, *, /) **shall** be a type that can be extended to a numeric numeric type, or an exception otherwise. The type of a multiplicative expression is the extended type of its operands. If this promoted type is an integer type, then integer arithmetic **shall** be performed; if this promoted type is a floating-point type, then floating-point arithmetic **shall** be performed; if this promoted type is a complex type, then complex arithmetic shall be performed.

NOTE Explicit and implicit casts should be used with caution, as unintended consequences can arise. Data can be lost when floating-point representations are converted to integral representations as the frac-

tional components of the floating-point values will be truncated (rounded down). Conversely, converting from an integral representation to a floating-point one can also loose precision, since the floating-point type may be unable to represent the integer exactly (for example, float might be an IEEE 754 single precision type, which cannot represent the integer 16777217 exactly, while a 32-bit integer type can). This can lead to situations such as storing the same integer value into two variables of type int and type float which return false if compared for equality.

Requirement 59
Whenever rounding from floating-point to integer numbers is required, rounding towards zero **shall** be applied.

Example For a Boolean single-field coverage `$b`, and an integer single-field coverage `$i`, the following expression will evaluate to some integer value:

```
count( $i * $b )
```

Requirement 60
Before executing any binary operation where the two operands are of different type a cast operation **shall** be attempted to achieve equal types. The result type of each of the binary induced operations (see Section 0) addition, subtraction, multiplication, and division **shall** be equal to the common type of the input operands.

Requirement 61
If a cast is attempted or implicitly needed, but not possible according to the rules of this standard then an exception **shall** be reported.

NOTE The cast operation is similar to that found in many programming languages, such as C/C++.

7.3 Evaluation exceptions

Requirement 62
Whenever a coverage expression cannot be evaluated according to the rules specified in Clauses 7.1 and 7.2, evaluation **shall** respond with an exception.

Example The following expressions will lead to an exception when used in a *ProcessCoverages* request (reasons: division by zero; illegal trigonometric argument):

```
$C / 0

arcsin( 2 )
```

7.4 processCoveragesExpr response

The response to a request sending a **processCoveragesExpr** is one of the following:

Requirement 63
Depending on its result type, the normal result of evaluating a valid WCPS query **shall** consist of one of the following alternatives:

- A (possibly empty) list of coverages.

- A (possibly empty) list of scalars (where scalar summarizes all non-coverage type data, such as numbers, strings, URLs) or of records of scalars.
- An exception.

Encoding of single coverages is governed by the **encodeCoverageExpr** (see Section 7.1.4). Encoding of scalar structures and exceptions as well as the representation of the overall response is be governed by a separate, additional protocol specification, WCS Processing [OGC 08-059r3], which specifies a response protocol suitable for WCPS results.

8 xWCPS

8.1 Overview

This Clause defines further WCPS functionality for OGC CIS 1.1 coverages, informally named xWCPS. Due to the wide range of coverage types CIS 1.1 supports (GeneralGridCoverage, RectifiedGridCoverage, ReferenceableGridCoverage, and subtypes thereof) a generic mechanism is established for extracting components from such coverages, as well as composing such coverages. This mechanism is based on XPath expressions operating on the structure of a coverage as defined in its XML schema.

NOTE Due to the schema agnostic nature of XPath it is not tied to any particular XML schema, and not even to XML itself. Based on these capabilities it is planned in future to support XPath operations also on the JSON schema, and on other suitable encodings of coverages.

An xWCPS expression may coverage, metadata, or combined expressions by integrating XPath into WCPS. This allows serving and querying data assets combined from data and metadata, even "reverse lookups" where the server evaluates predicates on coverage data and returns only pertaining metadata requested.

8.2 xWCPS language elements

An xWCPS query consists of an **xWcpsExpr**.

Requirement 64
Syntax and semantics of an ***xWcpsExpr*** **shall** be given by a WCPS **processCoveragesExpr** potentially in addition containing **letExprs**, **xpathExprs**, **xWcpsCoverageConstructorExprs**, **decodeCoverageExprs**, **switchExprs**.

NOTE Requirement counting starts with 1 because at the time of WCPS 1.0 writing the requirements scheme was not yet established. In this WCPS 1.1 document, all normative language is considered a formal requirement.

8.2.1 letExpr

The xWCPS **`let`** clause, adopted from W3C XQuery, declares a named constant (in XPath, not quite correctly, named *variable*) and gives it a value. In most cases, named constants are used purely for convenience, to simplify the expressions and make the code more readable.

Example The following statement defines a constant of name `$timeAxis` with value "date".

```
let $timeAxis := "date"
```

In a **`let`** clause the named constant only takes one value. This can be a single item or a sequence (there is no real distinction — an item is just a sequence of length one), and the sequence can contain nodes, or atomic values, or (beware!) a mixture of the two.

NOTE Named constants cannot be updated – something like `let $x:=$x+1` is not allowed. More specifically, it will not lead to an evaluation error, but the result will not be as expected (cf. XPath latera-ture). This rule might seem very strange if you are expecting WCPS to behave in the same way as procedural languages such as JavaScript or python. But WCPS isn't that kind of language, it's a declarative language and works at a higher level. This constraint is essential to give optimizers the chance to find execution strategies that can search vast databases in fractions of a second. SQL, XSLT, and XQuery users have found that this declarative style of programming grows on you. You start to realize that it enables you to code at a higher level: you tell the system what results you want, rather than telling it how to go about constructing those results.

A xWCPS expression can have any number of let clauses, and they can be in any order except that variables have to be defined before used. If a let clause contains multiple variables, it is semantically equivalent to multiple let clauses, each containing a single variable.

Example The following statement:

```
let $timeAxis := "date"
let $latAxis := "Lat"
```

is equivalent to:

```
let $timeAxis := "date", $latAxis := "Lat"
```

Requirement 65
Syntax and semantics of a **letExpr shall** be given as follows.

Let

$m,n \geq 1$ be natural numbers,
$c_1, \ldots c_m$, be n pairwise different **variableName**s,
$e_1, \ldots e_m$, be $n+m$ optional **coverageExpr**s or **scalarExpr**s or bracket-enclosed **intervalExpr**s,
c be a **coverageExpr** or **scalarExpr**,
where every c_i must be defined before used in an expression.

Then,

for any **xWcpsExpr** X
where

X = **for** v_1 **in** (L_1),
for v_2 **in** (L_2),
…,
for v_n **in** (L_n)
[**let** $c_1 := e_1, \ldots, c_m := e_m$]
[**where** b]
return P

the result of evaluating X is defined by substituting all occurrences of c_i in X by e_i and then evaluating the expanded X expression.

Example 1 The following xWCPS query:

```
for $C in ( myCoverage )
let $factor := 10
return encode( scale( $C, $factor ), "image/tiff" )
```

…is equivalent to:

```
for $C in ( myCoverage )
return encode( scale( $C, 10 ), "image/tiff" )
```

Example 2 Regions (such as for subsetting) can be extracted conveniently:

```
for $C in ( myCoverage )
let $subset := [ Lat(10:20), Long (30:40) ]
return max( $C [ $subset ] )
```

8.2.2 xpathExpr

W3C XPath expressions are syntactically allowed at any position. Semantically, the result of evaluating the expression must match with the context requirements.

Requirement 66
An xWCPS query consists of WCPS express **may** contain occurrences of variables defined in the scope of the XPath expression.

Requirement 67
An XPath expression in an xWCPS query **may** contain occurrences of variables defined in the scope of the XPath expression.

Example The following interleaving of WCPS and XPath is valid on some GeneralGridCoverage `$C`:

```
$C/domainSet/GeneralGrid/RegularAxis[@axisLabel=$a]
```

Requirement 68
In a context requiring coverages or constituents thereof the result of evaluating an XPath expression in an xWCPS query **shall** be interpreted as given by the CIS XML schema pertaining to the particular coverage type on hand.

Example For composing some CIS 1.1 GeneralGridCoverage through an xWCPS coverage constructor an XPath expression may contribute the range set; in this case, the result of the XPath evaluation must be a valid CIS 1.1 GeneralGridCoverage range set as per the XML schema.

A WCPS server may refuse to evaluate some expressions, in particular if the volume of intermediate or result data would exceed some internal limits. A typical example is retrieval of the complete range set.

Example For some coverage `$C`, the XPath expression `$C/GeneralGrid/@srsName` evaluates to the contents of the `srsName` attribute in the coverage's domain set.

NOTE XPath can be used to extract any part of a coverage. For example, the following expressions evaluate to components of coverage `$C`:

```
$c/domainSet
$c/rangeType
$c/rangeSet
$c/metadata
```

8.2.3 xWcpsCoverageConstructorExpr

The **xWcpsCoverageConstructorExpr** element creates a d-dimensional coverage of some given type for some $d \geq 1$ by defining the coverage's domain set, range type, range set, and metadata through expressions. This allows creating entirely new shapes, dimensions, and values – see the examples below.

NOTE This constructor is adjusted to CIS coverages, as opposed to the original coverage constructor (Subclause 7.1.29), and substantially more powerful than this older constructor, therefore using the new constructor is recommended.

The CIS coverage types supported by this standard are *RectifiedGridCoverage, ReferenceableGridCoverage*, and *GeneralGridCoverage*, or some subtype thereof. *GeneralGridCoverage* can express any grid type and axis type combination, for the other types some constraints apply.

Requirement 69
In a **xWcpsoverageConstructorExpr** a *RectifiedGridCoverage* **shall** contain only axes of type index and regular; a *ReferenceableGridCoverage* **shall** contain only axes of type index, regular and irregular.

The coverage domain set is built from a Coordinate Reference System (CRS) defining the multi-dimensional axes and the meaning of coordinates, including units of measure; indicating the coordinates of the direct positions, i.e., the points where values sit.

Axis names can be chosen according to the rules of CIS 1.1; it is recommended to keep Native CRS and grid CRS axis names disjoint.

A range type expression optionally creates the coverage range type. In the scope of the embedding WCPS condensers this expression defines the range component names as known (immutable) variables. Values derived for some such range component will automatically be cast to the target type (in SWE Common terminology: "quantity") of that range component.

A range set expression creates the coverage range set. A **scalarExpr** is evaluated at every direct position of the coverage's domain set.

An optional metadata expression creates the coverage metadata component. As such metadata are not interpreted by the coverage they are represented as a string which may contain any character, depending on the character set supported (which is out of scope of this standard).

8.2.3.1 Coverage Constructor

Requirement 70
A **xWcpsCoverageConstructorExpr shall** be defined as follows.

Let

t be an **identifier** which is either a coverage type defined in CIS or a coverage type validly derived from those,
id be an **identifier**,
D be a **domainSetExpr**,
T be a **rangeTypeExpr**,
R be a **rangeSetExpr**,
M be a **metadataExpr.**

Where

C is a **xWcpsCoverageConstructorExpr**
with

C = **coverage** *t id* [*D*] [*T*] *R* [*M*]

8.2.3.2 Domain Set

Let further

d be an **integer** with $d>0$,
c be a **crsName** representing a *d* -dimensional CRS,
a_i be pairwise distinct **variableName**s for $1 \le i \le d$,
$axis_i$ be pairwise distinct **axisName**s for $1 \le i \le d$,
$ie_{i,1}$, $ie_{i,2}$ be integer-valued **indexExpr**s for $1 \le i \le d$ with $ie_{i,1} \le ie_{i,2}$,
$ce_{i,1}$, $ce_{i,2}$ be **indexExpr**s for $1 \le i \le d$, which are valid coordinates for axis *i* as per CRS *c* with $ce_{i,1} \le ce_{i,2}$,
res_i be **indexExpr**s with $res_1 < \ldots < res_d$ for $1 \le i \le d$ valid for the i^{th} axis as per *c*,
$xe_{i,1}, \ldots$ be **indexExpr**s for $1 \le i \le d$, which are valid coordinates for axis $axis_i$ as per CRS *c* with $xe_{i,1} < xe_{i,2} < \ldots$,
$im_1, \ldots, im_m$ be **interpolationMethod**s for $1 \le i \le m$ with $m>0$.

Where

D is a **domainSetExpr**
with

D = **domain set**
crs *c* **with**
$axis_1$ $axisdef_1$,
…,
$axis_d$ $axisdef_d$

[, **interpolation** im_1,..., im_m]

and $axisdef_i$ is one of

$axisdef_{i,index}$ = **index (** $ie_{i,1}$ **:** $ie_{i,2}$ **)**

$axisdef_{i,regular}$ = **regular (** $ce_{i,1}$ **:** $ce_{i,2}$ **) resolution** res_i

$axisdef_{i,irregular}$ = **irregular (** $xe_{i,1}$ **,** ... **,** $xe_{i,n}$ **)**

D is defined as a domain set of CRS c with axis types as indicated and axis extents per CRS axis as indicated. Axis names used in the **domainSetExpr** shall be matched pairwise against the CRS axes based on their order of occurrence in the expression.

The axis names $axis_i$ are made available in the current context for use as iteration variables in the range set computation where coordinate values get bound to each direct position in turn allowing to inspect each direct position of the coverage. However, this is only admissible for index axes (see below).

8.2.3.3 Range Type

Let further

n be an **integer** with d>0,
f_1,..., f_n be **fieldNames**,
t_1,..., t_n be **rangeType**s,
$nil_{1,1}$,..., $nil_{i,1}$,..., $nil_{n,1}$,..., be **constant**s compatible with type q_i.

Where

T is a **rangeTypeExpr**
with

T = **range type**
f_1 **:** t_1 [**nil** $nil_{1,1}$, ...] **,**
…
f_n **:** t_n [**nil** $nil_{n,1}$, ...]

NOTE DataRecords in SWE Common, on which the CIS range type relies, define several more components which, however, have turned out to be unused in WCPS practice. Hence, there is a restriction to the practically relevant parts leaving other parts implementation dependent. A future version of this standard may add definitions for further SWE DataRecord components.

8.2.3.4 Range Set

Let further

r be a **scalarExpr** possibly containing occurrences of direct position indicators as defined in D and range component identifiers defined in T.

Where

R is a **rangeSetExpr**
with
R = **range set** r

8.2.3.5 Metadata

Let further

m be a **stringExpr**

Where

M is a $metadataExpr$
with
M = **metadata** m

NOTE If the metadata value resembles valid XML or JSON then XPath can address into these data.

8.2.3.6 Coverage Constructor Semantics

Then,

C is defined as a coverage of type t as follows:

Coverage constituent	Changed?
identifier(C) = id	**X**
crs(C) = c if D is present, or the Native CRS resulting from evaluating r otherwise	**X**
domain(C) = domain extent resulting from evaluating D, or the domain extent resulting from evaluating r otherwise	**X**
indexCrs(C) = d-dimensional Index CRS if D is present, or the Index CRS resulting from evaluating r otherwise	**X**
indexDomain(C) = $I_1 \times \ldots \times I_d$ if D is present, or the index domain extent resulting from evaluating r otherwise. Where the I_i are defined as: if I_i is index axis then $I_1 = [0:ie_{i,2}-ie_{i,1}]$ if I_i is regular axis then $I_i = [0:(ce_{i,2}-ce_{i,1})/res_i]$ if I_i is irregular axis then $I_i = [0:\lvert\{xe_{i,1},\ldots,xe_{i,n}\}\rvert-1]$	**X**
interpolationSet(C) = { $im_1,\ldots, im_m$ } if D is present, or the empty set {} otherwise	**X**
rangeFieldNames(C) = ($f_1,\ldots, f_n$) if T is present,	**X**

or the range field names resulting from evaluating r otherwise	
for all range fields f_i: *rangeFieldType*(C,f_i)=t_i if T is present, or the range type resulting from evaluating r otherwise	**X**
nullSet(C) = {$nil_{1,1}$, …} ×…× {$nil_{n,1}$, … } if T is present, or the empty set {} otherwise	**X**
for all p =(p_1,…,p_d)∈ *domain*(C): *value*(C,p) = range value resulting from evaluating r with possible occurrences of a_i substituted by the corresponding p_i coordinate value.	***x***
metadata(C) = m if M is present, or the empty string "" otherwise	**X**

If some range type components have null values but only one has not then the cross product is empty and the coverage cannot have any null value altogether. To avoid erroneous constructs, therefore, if at least one field has nulls then all fields must have nulls.

Requirement 71
In the range type definition of a **xWcpsCoverageConstructorExpr**, if there is at least one field with at least one null value defined then every field **shall** have at least one null value defined.

Addressing a neighbourhood of a pixel in a query is restricted to inspection on Index CRS level, so as to avoid addressing into positions which are not direct positions.

Requirement 72
In the range set definition of a **xWcpsCoverageConstructorExpr**, axis iterator variables **may** only be used if they are of type **index**.

NOTE One way of doing so is to use the Grid Index axes instead of the domain coordinates, such as *i*, *j*, *k* instead of *Lat, Long, time*. However, if the domain CRS has index axes these can be used as well (see below for an example). The definition of the 4D Index CRS can be obtained from here: http://www.opengis.net/def/crs/OGC/0/Index4D

8.2.3.7 Examples

The following domain set establishes a 2D WGS84 grid with several allowed interpolation methods.

```
domain set
  crs "EPSG:4326" with
  Lat  regular (10:30) resolution 0.01,
  Long regular (10:30) resolution 0.01,
  interpolation nearest, linear, quadratic, cubic
```

EPSG:4326 establishes Lat and Long axes, therefore in the following domain set expression the first axis will be associated with *Lat* and the second with *Long*, regardless of the axis naming in the domain set expression:

```
domain set
  crs "EPSG:4326" with
  a regular (10:30) resolution 0.5,
  b regular (10:30) resolution 0.5
```

The next domain set establishes a 4D georeferenced timeseries datacube with a spectral dimension, regular in Lat/Long and irregular in time (given the varying number of days a month has and based on the daily resolution specified).

```
domain set
  crs "EPSG:4326+OGC:unixTtime" with
  Lat  regular (10:30) resolution 0.5,
  Long regular (10:30) resolution 0.5,
  date irregular ( "2017-01-01", "2017-02-01",
                   "2017-03-01", "2017-04-01",
                   "2017-05-01", "2017-06-01",
                   "2017-07-01", "2017-08-01",
                   "2017-09-01", "2017-10-01",
                   "2017-11-01", "2017-12-01"
                 ),
  band index (1:32)
```

The expression below represents a single-band range type of data type unsigned short with null values 254 and 255; allowed interpolations are linear and quadratic.

```
range type
  panchromatic : unsigned char nil 254, 255
```

The following range type defines RGB pixels.

```
range type
  red   : unsigned char,
  green : unsigned char,
  blue  : unsigned char
```

The coverage constructor below resembles an induced operation, reducing intensity in all range fields by ½. Domain set and range type are adopted from the input coverage.

```
coverage GeneralGridCoverage Half
range set (unsigned char) $C / 2
```

Below follows a complete coverage constructor representing a 3-D georeferenced image timeseries whose range set gets loaded from some input file provided, represented by the positional parameter `$1`. Further, some sketchy INSPIRE XML metadata record is associated:

```
coverage GeneralGridCoverage MySatelliteDatacube
domain set
  crs "EPSG:4326+OGC:unixTime" with
  Lat  regular (10:30) resolution 0.5,
  Long regular (10:30) resolution 0.5,
  date regular ("2017-01":"2019-12") resolution "P1M"
range type panchromatic : short,
range set decode( $1 )
metadata "<inspireMetadata>...</inspireMetadata>"
```

The expression below computes a 256-bucket histogram over band `blue` of some coverage `$C` of unknown domain extent and dimension:

```
coverage GeneralGridCoverage histogram
domain set
  crs OGC:Index1D with
  bucket index (0:255)
range type
  b : unsigned long
range set
  count( $C.blue = bucket )
```

If constituents can be determined then they do not need to be indicated; in this case input coverage `$C` is copied; assuming it has range type unsigned short then the *log*() operation suggests a float result, so this will be adopted as range type. Along the same line, the domain set is adopted from `$C`:

```
coverage GeneralGridCoverage LogOfCube
range set log( $C )
```

The earlier mentioned filter kernel operation is expressed in xWCPS like this:

```
coverage GeneralGridCoverage FilteredImage
domain set
  crs "OGC:Index2d" with
  x index ( 0 : 5000 ),
  y index ( 0 : 5000 )
range set
  condense +
  over  $pi i index ( -1 : +1 ),
        $pj j index ( -1 : +1 )
  using $C.blue[ x(x+i), y(y+j) ]
```

8.2.4 decodeCoverageExpr

A *`decodeCoverageExpr`* evaluates a byte stream passed as parameter to a coverage by decoding the byte stream. This byte stream must represent a coverage encoding following CIS 1.1 [09-146r6] and its coverage encoding profiles.

NOTE Implementations will be able to recognize the encoding format used from analyzing the input byte stream, hence no format indication parameter is required.

Requirement 73
Syntax and semantics of a *decodeCoverageExpr* **shall** be given as follows.

Let

b be a *byteString*

where

b is a valid (binary or ASCII) representation of a complete coverage or a domain set, range type, range set, or metadata component of a coverage,
extraParams is a **stringConstant** containing decoding directives, such as defined in the CIS encoding profile for the format on hand,

Then,

for any *decodeCoverageExpr C*
where *C* is one of
C_e = decode (*b*)
C_{ee} = decode (*b* , *extraParams*)

C is defined as the decoded coverage or coverage component equivalent to *b* while applying the directives in *extraParams*.

In practice, this function can be used in several ways:

- To provide inline constants, such as XML documents or fragments
- To provide input files, accompaniying the query, through positional parameters

Example Assume a NetCDF file is passed as positional parameter $1 in the WCS Processing Extension [OGC 08-059]). This decodes the byte stream and establishes the corresponding coverage:

```
decode( $1 )
```

Note The *extraParams* syntax and semantics is data format dependent, specified in the CIS encoding formats.

8.2.5 switchExpr

The *switchExpr* provides a case distinction for choosing among a set of coverages that all share domain and range type. Conditions provided are evaluated sequentially, and the first *true* alternative is chosen if any; otherwise, the default alternative is chosen.

- If the result expressions return scalar values, the returned scalar value on a branch is used in places where the condition expression on that branch evaluates to *true*.
- If the result expressions return coverages, the values of the returned coverage on a branch are copied in the result coverage in all places where the condition coverage on that branch contains pixels with value *true*.

The conditions of the statement are evaluated in a manner similar to the *if-then-else* statement in programming languages such as Java or C++.

NOTE This implies that the conditions must be specified by order of generality, starting with the least general and ending with the default result, which is the most general one. A less general condition specified after a more general condition will be ignored, as the expression meeting the less general expression will have had already met the more general condition.

Requirement 74
Syntax and semantics of a `switchExpr` **shall** be given as follows.

Let

n be an `integer` with $n \geq 1$,
$b_1, \ldots, b_n$ be `booleanExpr`s with a single Boolean range component,
$C_1, \ldots, C_n$ be `coverageExpr`s with a single Boolean range component,
$R, R_1, \ldots, R_{n+1}$ be `coverageExpr`s,

where, for $1 \leq i \leq n$,

$crs(C_1) = \ldots = crs(C_n) = crs(R_1) = \ldots = crs(R_{n+1})$,
$domain(C_1) = \ldots = domain(C_n) = domain(R_1) = \ldots = domain(R_{n+1})$,
$indexCrs(C_1) = \ldots = indexCrs(C_n) = indexCrs(R_1) = \ldots = indexCrs(R_{n+1})$,
$indexDomain(C_1) = \ldots = indexDomain(C_n) = indexDomain(R_1) = \ldots = indexDomain(R_{n+1})$,
$rangeType(R_1) = \ldots = rangeType(R_{n+1})$.

Then,

for any `coverageExpr C'`
where

```
C' =  switch
          case C1   return R1
          …
          case Cn   return Rn
          default return Rn+1
```

coverage `C'` is defined as follows:

Coverage constituent	Changed?
identifier(`C'`) = "" (empty string)	**X**
crs(`C'`) = *crs*(R_1)	
domain(`C'`) = *domain*(R_1)	
indexCrs(`C'`) = *indexCrs*(R_1)	
indexDomain(`C'`) = *indexDomain*(R_1)	

interpolationSet(C') = *interpolationSet*(R_1) ∩ … ∩ *interpolationSet*(R_{n+1})	**X**
rangeFieldNames(C') = *rangeFieldNames*(R_1)	
rangeType(C') = rangeType(R_1)	
nullSet(C') = *nullSet*(R_1) ∪ … ∪ *nullSet*(R_{n+1})	**X**
for all $p \in$ *domain*(C'): *value*(C', p) = if *values*(C',p) = *true* then *values*(R_1,p) else if *values*(C',p) ∈{*false*,*null*} then *values*(R_2,p) … else *values*(R_{n+1},p)	**X**
metadata(C_2) = {}	**X**

Example 1 The expression below performs a traffic light classification on some single-band coverage `$c`.

```
switch
  case $c < 10 return $c * {red: 0;   green: 0;   blue: 255}
  case $c < 20 return $c * {red: 0;   green: 255; blue: 0}
  case $c < 30 return $c * {red: 255; green: 0;   blue: 0}
  default      return      {red: 0;   green: 0;   blue: 0}
```

Example 2 The example below computes log of all positive values in `$c`, and assigns 0 to the remaining ones. This way it avoids an exception that would otherwise be thrown should any cell not be above zero.

```
switch
  case $c>0 return log($c)
  default   return 0
```

8.3 Evaluation response

The response to a valid xWCPS query shall be a (possibly empty) sequence of strings or a (possibly empty) sequence of coverages.

Requirement 75
A coverage-valued response to a valid WCPS request **shall** validate against OGC CIS.

8.4 Metadata

An implementation may choose to not return the complete metadata component of a coverage.

NOTE A possible reason is to differentiate between global metadata (which are valid for the whole coverage and get delivered always) and local metadata (which are valid for a part of a coverage and get only

delivered in a subsetting when the area of their validity is retrieved). In this version of the standard this behavior is implementation dependent; in a future version, conventions may be established.

8.4.1 Operator precedence rules

Operator precedence of WCPS remains unchanged, but new operators are phased in.

Requirement 76
xWCPS Operator precedence **shall** be as follows:

- Range field selection, trimming, slicing
- unary –
- unary arithmetic, trigonometric, and exponential functions
- `*`, `/`
- `+`, `-`
- `<`, `<=`, `>`, `>=`, `!=`, `=`
- `and`
- `or`, `xor`
- `:` (interval constructor), condense, coverage, xWCPS coverage constructor
- Overlay, switch

In all remaining cases evaluation is done left to right.

8.5 Character encoding

The character set available in an xWCPS query is depending on the embedding protocol.

Requirement 77
The character set of an xWCPS request **shall** follow the rules of the embedding protocol binding context; default is US ASCII.

Example An extra parameter in a WCS Processing request may specify a Unicode character set.

A quoted string may contain any character, even national special characters and non-printable characters, as long as these are properly encoded following http URL rules.

Requirement 78
Non-printable characters in an xWCPS request **shall** be represented according to http rules.

8.6 Evaluation exceptions

Not all syntactically valid xWCPS expressions are semantically admissible. Possible errors include:

- An XPath expression result is used as input for a function which expects a different structure;
- The data volume construed by the XPath expression is exceeding the server's capabilities;
- An XPath expression attempts to reach into the metadata, but these are not valid XML.

Requirement 79
Whenever a coverage expression cannot be evaluated an xWCPS service **shall** respond with an exception.

NOTE No specific exception handling is defined here as this will be defined by the concrete protocol binding the WCPS service provides, such as GET/KVP, XML/POST, SOAP, or OpenAPI.

Annex A
(normative)

Abstract Test Suite

The Abstract Test Suite for WCPS is provided in [OGC 08-069], defining conformance class WCPS which is the core conformance class. The additional rules for xWCPS, which needs to be passed in addition to WCPS, are provided below.

Requirement 1 – Syntax and semantics of a *xWcpsExpr* **shall** be given by a WCPS **processCoverageExpr** potentially in addition containing **letExprs**, **xpathExprs**, **xWcpsCoverageConstructorExprs**, **decodeCoverageExprs**, **switchExprs**.

Test:

- Send query to system under test containing valid versions of each of the expressions
- Check for proper result.

Requirement 2 – Syntax and semantics of a *letExpr* **shall** be given as follows.

Test:

- Send query to system under test containing valid versions of let expressions
- Check for proper result.

Requirement 3 – An xWCPS query consists of WCPS express **may** contain occurrences of variables defined in the scope of the XPath expression.

Test:

- Send query to system under test containing let expressions with valid variables used in other places of the query
- Check for proper result.

Requirement 4 – An XPath expression in an xWCPS query **may** contain occurrences of variables defined in the scope of the XPath expression.

Test:

- Send query to system under test containing valid XPath expressions utilizing let variables
- Check for proper result.

Requirement 5 – In a context requiring coverages or constituents thereof the result of evaluating an XPath expression in an xWCPS query **shall** be interpreted as given by the CIS XML schema pertaining to the particular coverage type on hand.

Test:

- Send query to system under test containing valid XPath expressions
- Check for result matching CIS XML schema.

Requirement 6 – A **domainSetExpr shall** describe a valid coverage domain set following the rules defined in OGC CIS whereby **domainSetExpr** establishes the following domain set constituents: (…)

Test:

- Send query to system under test containing valid domainSetExprs (as part of coverage constructor)
- Check for proper domain set result.

Requirement 7 – Syntax and semantics of a **rangeTypeExpr shall** be given as follows.

Test:

- Send query to system under test containing valid rangeTypeExprs (as part of coverage constructor)
- Check for proper domain set result.

Requirement 8 – Syntax and semantics of a **rangeSetExpr shall** be given as follows.

Test:

- Send query to system under test containing valid rangeSetExprs (as part of coverage constructor)
- Check for proper result.

Requirement 9 – Syntax of a **metadataExpr shall** be given as follows.

Test:

- Send query to system under test containing valid metadataExprs (as part of coverage constructor)
- Check for proper result.

Requirement 10 – Syntax and semantics of an **xWcpsCoverageConstructorExpr shall** be given as follows.

Test:

- Send query to system under test containing valid xWcpsCoverageConstructo-rExprs
- Check for proper result.

Requirement 11 – In a **xWcpsoverageConstructorExpr** a *RectifiedGridCoverage* **shall** contain only axes of type index and regular; a ReferenceableGridCoverage **shall** contain only axes of type irregular.

Test:

- Send query to system under test containing valid xWcpsCoverageConstructo-rExprs
- Check for proper result.

Requirement 12 – Syntax and semantics of a ***decodeCoverageExpr*** **shall** be given as follows.

Test:

- Send query to system under test containing valid decodeCoverageExprs
- Check for proper domain set result.

Requirement 13 – Syntax and semantics of a **switchExpr shall** be given as follows.

Test:

- Send query to system under test containing valid switchExprs
- Check for proper domain set result.

Requirement 14 – A coverage-valued response to a valid WCPS request **shall** validate against OGC CIS.

Test:

- Send query to system under test returning OGC CIS coverages
- Check for proper domain set result.

Requirement 15 xWCPS Operator precedence **shall** be as follows: (…)

Test:

- Send query to system under test containing unparenthesized expressions, send same queries adequately parenthesized to determine same result
- Check for proper domain set result.

Requirement 16 – The character set of an xWCPS request **shall** follow the rules of the embedding protocol binding context; default is US ASCII.

Test:

- Send query to system under test containing valid characters for protocol bindings supported by the system under test; test at least US ASCII.
- Check for proper domain set result.

Requirement 17 – Non-printable characters in an xWCPS request **shall** be represented according to http rules.

Test:

- Send query to system under test containing non-printable characters in input and output
- Check for proper domain set result.

Requirement 18 – Whenever a coverage expression cannot be evaluated an xWCPS service **shall** respond with an exception.

Test:

- Send query to system under test which must fail, expecting proper exceptions
- Check for proper domain set result.

Annex B
(normative)

WCPS Expression Syntax

B.1 Overview

This Annex summarizes the WCPS expression syntax. It is described in EBNF grammar syntax according to [IETF RFC 2616].

Underlined tokens represent literals which appear "as is" in a valid WCPS expression ("terminal symbols"), tokens in italics represent either sub-expressions to be substituted according to the grammar production rules ("non-terminals") or terminal symbol classes like identifiers, strings, and numbers as listed at the end of this Annex. The *processCoveragesExpr* nonterminal is the start of the production system.

Any number of whitespace characters (blank, tabulator, newline) **may** appear between tokens as long as parsing is unambiguous.

Example Between language tokens (such as "for") and names there must be at least one whitespace character, whereas between names and non-alphanumeric tokens (such as opening parenthesis, "("), no whitespace is required.

Meta symbols used are as follows:

- brackets ("[...]") denote optional elements which **may** occur or be left out;
- an asterisk ("*") denotes that an arbitrary number of repetitions of the following element **can** be chosen, including none at all;
- a vertical bar ("|") denotes alternatives from which exactly one **must** be chosen;
- Double slashes ("//") begin comments which continue until the end of the line. Comments are normative.

NOTE The syntax as is remains ambiguous; the semantic rules listed in this document disambiguate the grammar.

B.2 WCPS terminal symbols

The following are terminal symbols, in addition to the underlined terminal literals: variableName; *name; stringConstant; booleanConstant; integerConstant; floatConstant.*

A variableName **shall** be a consecutive sequence of characters where the first character **shall** be either an alphabetical character or the "$" character and the remaining characters consist of decimal digits, upper case alphabetical characters, lower case alphabetical characters, underscore ("_"), and nothing else. The length of an identifier **shall** be at least 1.

NOTE The regular expression describing a variable is: $[a-zA-Z_][0-9a-zA-Z_]*.

A name **shall** either be a consecutive sequence of digits and/or letters where the first character is a letter, or a non-empty string constant.

NOTE 1 The regular expression describing a name identifier is: ([a-zA-Z_][0-9a-zA-Z_]*)|(".+").

NOTE 2 WCS [OGC 07-067r5] allows significant freedom in the choice of names; to combine syntactical simplicity with this generality in a syntax-controlled environment both a non-quoted and a quoted variant are provided.

A *booleanConstant* **shall** represent a logical truth value expressed as one of the literals "true" and "false" resp., whereby upper and lower case characters **shall** not be distinguished.

An *integerConstant* **shall** represent an integer number expressed in either decimal, octal (with a "0" prefix), or hexadecimal notation (with a "0x" or "0X" prefix).

A *floatConstant* **shall** represent a floating point number following the syntax of the Java programming language.

A *stringConstant* **shall** represent a character sequence expressed by enclosing it into single or double quotes, with no mix of both in a single constant.

B.3 WCPS syntax

```
processCoveragesExpr:
        for variableName in ( coverageList )
            *( , variableName in ( coverageList ) )
        [ where booleanScalarExpr ]
        return processingExpr
      | xWcpsExpr

coverageList:
        coverageName *( , coverageName )

processingExpr:
        encodedCoverageExpr
      | scalarExpr

encodedCoverageExpr:
        encode ( coverageExpr , formatName
                 [ , extraParams ] )

formatName:
        stringConstant

extraParams:
        stringConstant

scalarExpr:
        getComponentExpr
```

```
        | booleanScalarExpr
        | numericScalarExpr
        | ( scalarExpr )

booleanScalarExpr:
          booleanScalarExpr or booleanScalarTerm
        | booleanScalarExpr xor booleanScalarTerm
        | booleanScalarTerm

booleanScalarTerm:
          booleanScalarTerm and booleanScalarFactor
        | booleanScalarFactor

booleanScalarFactor:
          numericScalarExpr compOp numericScalarExpr
        | stringScalarExpr compOp stringScalarExpr
        | not booleanScalarExpr
        | ( booleanScalarExpr )
        | booleanConstant

numericScalarExpr:
          numericScalarExpr + numericScalarTerm
        | numericScalarExpr - numericScalarTerm
        | numericScalarTerm

numericScalarTerm:
          numericScalarTerm * numericScalarFactor
        | numericScalarTerm / numericScalarFactor
        | numericScalarFactor

numericScalarFactor:
          ( numericScalarExpr )
        | - numericScalarFactor
        | round ( numericScalarExpr )
        | integerConstant
        | floatConstant
        | complexConstant
        | condenseExpr

compOp:
          =
        | !=
        | >
        | >=
        | <
        | <=

stringScalarExpr:
          identifierExpr
        | stringConstant

getComponentExpr:
          identifierExpr
```

```
        | crs ( coverageExpr )
        | domainExpr
        | imageCrs ( coverageExpr )
        | indexCrs ( coverageExpr )
        | indexDomain ( coverageExpr )
        | imageCrsDomain ( coverageExpr )
        | indexDomain ( coverageExpr , axisName )
        | imageCrsDomain ( coverageExpr , axisName )
        | indexDomain ( coverageExpr , axisName ) . lo
        | imageCrsDomain ( coverageExpr , axisName ) . lo
        | indexDomain ( coverageExpr , axisName ) . hi
        | imageCrsDomain ( coverageExpr , axisName ) . hi
        | nullSet ( coverageExpr )
        | interpolationSet ( coverageExpr )

identifierExpr:
          identifier ( coverageExpr )
        | id ( coverageExpr )
        | name ( coverageExpr )

domainExpr:
          domain ( coverageExpr )

setComponentExpr:
          setIdentifier ( coverageExpr , stringConstant )
        | setId ( coverageExpr , stringConstant )
        | setName ( coverageExpr , stringConstant )
        | setNull ( coverageExpr ,
            { [ rangeExpr *( , rangeExpr ) ] } )
        | setInterpolation ( coverageExpr ,
            { interpolationMethod
                *( , interpolationMethod ) } )
        | setMetadata ( coverageExpr , stringConstant )

rangeExpr:
          struct { fieldName : scalarExpr
                   *( , fieldName : scalarExpr ) }
        | { scalarExpr *( , scalarExpr ) }

coverageExpr:
          coverageLogicExpr or coverageLogicTerm
        | coverageLogicExpr xor coverageLogicTerm
        | coverageLogicTerm

coverageLogicTerm:
          coverageLogicTerm and coverageLogicFactor
        | coverageLogicFactor

 coverageLogicFactor:
          coverageArithmExpr compOp coverageArithmExpr
        | coverageArithmExpr
```

```
coverageArithmExpr:
        coverageArithmExpr + coverageArithmTerm
      | coverageArithmExpr - coverageArithmTerm
      | coverageArithmTerm

coverageArithmTerm:
        coverageArithmTerm * coverageArithmFactor
      | coverageArithmTerm / coverageArithmFactor
      | coverageArithmFactor

coverageArithmFactor:
        coverageArithmFactor overlay coverageValue
      | coverageValue

coverageValue:
        coverageAtom
      | subsetExpr
      | unaryInducedExpr
      | crsTransformExpr
      | scaleExpr

coverageAtom:
        variableName
      | setComponentExpr
      | scalarExpr
      | coverageConstantExpr
      | coverageConstructorExpr
      | ( coverageExpr )

unaryInducedExpr:
        unaryArithmeticExpr
      | exponentialExpr
      | trigonometricExpr
      | booleanExpr
      | castExpr
      | fieldExpr
      | rangeConstructorExpr

unaryArithmeticExpr:
        + coverageAtom
      | - coverageAtom
      | sqrt ( coverageExpr )
      | abs ( coverageExpr )
      | re ( coverageExpr )
      | im ( coverageExpr )

exponentialExpr:
        exp ( coverageExpr )
      | log ( coverageExpr )
      | ln ( coverageExpr )

trigonometricExpr:
        sin ( coverageExpr )
```

```
        | cos ( coverageExpr )
        | tan ( coverageExpr )
        | sinh ( coverageExpr )
        | cosh ( coverageExpr )
        | tanh ( coverageExpr )
        | arcsin ( coverageExpr )
        | arccos ( coverageExpr )
        | arctan ( coverageExpr )

booleanExpr:
          not coverageExpr
        | bit ( coverageExpr , indexExpr )

indexExpr:
          indexExpr + indexTerm
        | indexExpr - indexTerm
        | indexTerm

indexTerm:
          indexTerm * indexFactor
        | indexTerm / indexFactor
                        // integer division, rounding to next integer towards 0
        | indexFactor

indexFactor:
          indexConstant
        | round ( numericScalarExpr )
        | ( indexExpr )

castExpr:
          ( rangeType ) coverageExpr

rangeType:
          boolean
        | char
        | unsigned char
        | short
        | unsigned short
        | long
        | unsigned long
        | float
        | double
        | complex
        | complex2
        | complex char
        | complex short
        | complex int
        | complex long

fieldExpr:
          coverageExpr . fieldName
```

```
rangeConstructorExpr:
        struct { fieldName : coverageExpr
                    *( , fieldName : coverageExpr ) }

subsetExpr:
        trimExpr
      | sliceExpr
      | extendExpr

trimExpr:
        coverageExpr [ dimensionIntervalList ]

dimensionIntervalList:
        dimensionIntervalElement
        *( , dimensionIntervalElement )

dimensionIntervalElement:
        axisName ( axisPointExpr : axisPointExpr )
      | axisName ( domainExpr )

dimensionCrsList:
        { dimensionCrsElement *( , dimensionCrsElement ) }

dimensionCrsElement:
        axisName : crsName

sliceExpr:
        coverageExpr [ axisPointList ]

axisPointList:
        axisPointElement *( , axisPointElement )

axisPointElement:
        axisName ( axisPointExpr )

axisPointExpr:
        stringConstant
      | booleanConstant
      | integerConstant
      | floatConstant

extendExpr:
        extend ( coverageExpr , dimensionIntervalList )

scaleExpr:
        scale ( coverageExpr , dimensionIntervalList ,
                [ , InterpolationMethod ] )
      | scale ( coverageExpr , scalarExprList
                [ , InterpolationMethod ] )
      | scale ( coverageExpr , scalarExpr
                [ , InterpolationMethod ] )
```

```
scalarExprList:
        { scalarExpr [ , scalarExpr ] }

crsTransformExpr:
        crsTransform ( coverageExpr ,
                       dimensionCrsList ,
                       fieldInterpolationList )

interpolationMethod:
        nearest
      | linear
      | quadratic
      | cubic

coverageConstructorExpr:
        coverage coverageName
        over axisIterator *( , axisIterator )
        values scalarExpr

axisIterator:
         variableName axisName ( intervalExpr )

intervalExpr:
        indexExpr : indexExpr     // left value ≤ right value
      | imageCrsDomain ( variableName , axisName )
      | indexDomain ( variableName , axisName )


coverageConstantExpr:
        coverage coverageName
        over axisSpec *( , axisSpec )
        values < constant *( ; constant ) >

axisSpec:
        axisName ( intervalExpr )

condenseExpr:
        reduceExpr
      | generalCondenseExpr

reduceExpr:
        all ( coverageExpr )
      | some ( coverageExpr )
      | count ( coverageExpr )
      | add ( coverageExpr )
      | avg ( coverageExpr )
      | min ( coverageExpr )
      | max ( coverageExpr )

generalCondenseExpr:
        condense condenseOpType
        over axisIterator *( , axisIterator )
```

```
        [ where booleanScalarExpr ]
        using scalarExpr

condenseOpType:
        +
      | *
      | max
      | min
      | and
      | or

coverageName:
        nameOrString

crsName:
       stringConstant    // containing a valid CRS identifier

axisName:
        name

fieldName:
        name

constant:
        stringConstant
      | booleanConstant
      | integerConstant
      | floatConstant
      | complexConstant

complexConstant:
        ( floatConstant , floatConstant )
        ( integerConstant , integerConstant )
```

B.3 xWCPS syntax

For xWCPS definition this new start symbol is used. It relies on the WCPS syntax, i.e., its nonterminals and terminals declared in Subclauses B.1 and B.2.

```
xWcpsExpr:
        for variableName in ( coverageList )
            *( , variableName in ( coverageList ) )
        [ letExpr ]
        [ where booleanScalarExpr ]
        return processingExpr

letExpr:
         let letBinding *( , letBinding )
```

```
letBinding:
          variableName := ( coverageExpr | scalarExpr
                          | [ intervalExpr ] )

xWcpsCoverageConstructorExpr:
          coverage coverageTypeName nameOrIdentifier
          [ domainSetExpr ] [ rangeTypeExpr ] rangeSetExpr
          [ metadataExpr ]

coverageTypeName:
            RectifiedGridCoverage
          | ReferenceableGridCoverage
          | GeneralGridCoverage
          | nameOrString // name of a coverage type derived from the above

domainSetExpr:
          domain set
          crs nameOrString with
          nameOrString axisdefExpr
          *( , nameOrString axisdefExpr )
          [ interpolationExpr ]

interpolationExpr:
          interpolation
          interpolationMethod *( , interpolationMethod )

axisdefExpr:
            index ( indexExpr : indexExpr )
          | regular (axisPointExpr : axisPointExpr )
            resolution axisPointExpr
          | irregular (axisPointExpr *( , axisPointExpr ) )

rangeTypeExpr:
          range type rangeComponent ( , rangeComponent )*

rangeComponent:
          nameOrString : name [ nilExpr ]

nilExpr:
          nil rangeExpr *( , rangeExpr )

rangeSetExpr:
          range set scalarExpr

metadataExpr:
          metadata stringConstant

nameOrString:
          name | stringConstant

scalarExpr:
            …          // B.2 definitions
          | xpathExpr  // as per W3C XPath 3.1
```

Coordinates can be more general, and even non-numeric (cf. date and time):

```
intervalExpr:
                ...        // WCPS definitions
        | nameOrString
```

Bibliography

[1] Ritter, G., Wilson, J., Davidson, J.: Image Algebra: An Overview. Computer Vision, Graphics, and Image Processing, 49(3)1990, pp. 297-331

[2] Baumann, P.: A Database Array Algebra for Spatio-Temporal Data and Beyond. The Fourth International Workshop on Next Generation Information Technologies and Systems (NGITS '99), July 5-7, 1999, Zikhron Yaakov, Israel, Lecture Notes on Computer Science 1649, Springer Verlag, pp. 76 – 93

[3] Baumann, P.: The OGC Web Coverage Processing Service (WCPS) Standard. Geoinformatica, 14(4)2010, pp 447-479

[4] P. Baumann, D. Misev, V. Merticariu, B. Pham Huu: Datacubes: Towards Space/Time Analysis-Ready Data.. In: J. Doellner, M. Jobst, P. Schmitz (eds.): Service Oriented Mapping - Changing Paradigm in Map Production and Geoinformation Management, Springer Lecture Notes in Geoinformation and Cartography, 2018

[5] P. Baumann, A.P. Rossi et al: Fostering Cross-Disciplinary Earth Science Through Datacube Analytics. In. P.P. Mathieu, C. Aubrecht (eds.): Earth Observation Open Science and Innovation - Changing the World One Pixel at a Time, International Space Science Institute (ISSI), 2017, pp. 91 - 119