JACOBS UNIVERSITY

Peter Baumann, Georgi Chulkov

Web Coverage Processing Service (WCPS) Implementation Specification

Technical Report No. 9
July 2007

School of Engineering and Science

Web Coverage Processing Service (WCPS) Implementation Specification

Peter Baumann, Georgi Chulkov
School of Engineering and Science
Jacobs University Bremen gGmbH
Campus Ring 1
28759 Bremen
Germany
E-Mail: p.baumann@jacobs-university.de, g.chulkov@jacobs-university.de
http://www.jacobs-university.de/

Summary

The Open GeoSpatial Consortium (OGC) is the main driving force for open, interoperable service interfaces for geospatial information. To serve this mission, OGC develops and maintains a family of modular standards. The historically main standard for multi-dimensional raster data is the Web Coverage Service (WCS) Implementation Specification which defines basic access functionality on server-stored raster data (aka "coverages").

During the development of WCS several requests have came up to add this or that processing functionality. It was decided, however, to keep WCS basic in its functionality and rather have a separate, WCS-related standard which allows users to compose their own functions without complexity restrictions – in short: to develop a standard offering a raster processing language.

The pertaining specification is the Web Coverage Processing Service (WCPS) Implementation Specification which is currently under development within OGC, by a working group initiated and led by Jacobs University Bremen. WCPS provides access to original or derived sets of geospatial coverage information, in forms that are useful for client-side rendering, input into scientific models, and other client applications. As such, WCPS includes WCS functionality and extends it with an expression language to form requests of arbitrary complexity allowing, e.g., multi-valued coverage results.

The official status of WCPS within OGC is that of a Best Practice document (document no. 06-035r1), ie., an official recommendation by OGC. The report on hand presents an advanced WCPS specification version, which is considered complete and consistent enough to present it for public discussion and, finally, decision on becoming an accepted OGC standard.

Contents

1 Introduction

Increasingly raster data are becoming integral component of geo services, since today's hardware and software technology is powerful enough to allow online access even to objects of Terabyte to Petabyte size. Following commercial service offerings such as GlobeXplorer (www.globexplorer.com) and scientific portals like NASA's WorldWind (worldwind.arc.nasa.gov), GoogleEarth has brought the final public breakthrough for navigational services on large-scale earth observation imagery.

Actually, 2-D imagery is but the tip of the iceberg - the general concept of multi-dimensional spatio-temporal raster data covers 1-D sensor time series, 2-D imagery, 3-D image time series (*x*/*y*/*t*) and exploration data (*x*/*y*/*z*), 4-D climate models (*x*/*y*/*z*/*t*), and many more. Data sizes frequently are extremely high, amounting to multi-Terabyte volumes for single objects. Hence, truly comprehensive services should include multi-dimensional objects.

Another logical next step from standalone geo Web services are interoperable, community-sharable archive networks, and this raises the question of standardised access interfaces.

The Open GeoSpatial Consortium (OGC, www.opengeospatial.org) is the main driving force in standardizing interoperable geo services. It does so in close collaboration with relevant players in the field, such as ISO, OASIS, W3C, and geo domain bodies like EPSG and IUGS. OGC's approach is to establish a modular family of geo service standards, each one addressing a particular technological or geo-scientific application scenario. Aside from metadata and vector data, raster data obviously play an important role.

The term "*coverage*", in OGC [7] and ISO [6] definition, denotes "space-varying phenomena" – a definition which is rather general, ranging from regular grids to triangulated irregular networks (TINs) and beyond. In practice, however, all coverage-related specifications exclusively address raster data. We follow this focus for the purpose of this paper, defining raster data as any (usually, but not exclusively) spatio-temporal phenomenon which consists of sampled values aligned on a regularly spaced grid.

Access to spatio-temporal coverage data in OGC is being addressed in a dedicated manner by the Web Coverage Service (WCS) Implementation Specification [4]. Basically, WCS supports a fixed set of functions which can be invoked in a request: spatial, temporal, and band subsetting, scaling, reprojection, and final result packaging, including data format encoding.

The OGC WCS Revision Working Group (WCS.RWG) during its work on improving, refining, and extending WCS has been confronted with several requests for adding some domain-specific functionality. For example, derivation of the Normalized Difference Vegetation Index (NDVI) is mathematically well defined on some hyperspectral earth observation image `a` containing a red (`red`) and a near-infrared (`nir`) channel:

```
NDVI(a) = ( a. nir – a.red ) / ( a.nir + a.red )
```

The result is an image showing, per pixel, a vegetation indicator ranging from -1 to +1; the closer the value is to +1, the higher the vegetation indication is. Obviously it is possible to dynamically derive the NDVI from a given suitable coverage.

Still, the WCS.RWG refused to include such operations with two reasons: adding further functions beyond the fixed basic set would complicate the WCS interface specification for both client and server implementation significantly. Further, the list of feasible functions is unlimited, and there is no coherent concept for structuring them in a meaningful way for applications – in other words, the result would be an unmanageable, hard to understand, arbitrary conglomerate of functions.

Instead, the approach has been taken to implement a coverage processing language which allows to express algorithmically well-defined operations through one coherent, extensible concept. This language is defined over the coverage model and, hence, extends WCS in a compatible manner. The corresponding standard in spe is the Web Coverage Processing (WCPS) Implementation Specification. WCPS supports retrieval of multi-dimensional coverage data with spatial, temporal, and non-spatiotemporal ("abstract") semantics. WCPS expressiveness is suitable for server-side navigation, portrayal, analysis, and processing.

This report incorporates the current state of the WCPS specification document. The author of the WCPS specification is active member of the OGC WCS Revision Working Group, Chair of the WCPS Working Group, and Co-Chair of the Coverages Working Group.

As an official OGC document, a specification must adhere to a given structure which is largely followed also in this report. What has been left out here is the OGC document headers, most of the appendices, and the XML schema which is contained in separate files available, e.g., from the EarthLook website (www.earthlook.org). On the other hand, a wrapper has been added to make this report fully self-contained.

The resulting report structure is as follows. In the subsequent Section 2 we briefly characterize some relevant OGC standards. In Section 3 we summarise the requirements on WCPS. In Section 4 we briefly describe the WCPS service stack of the reference implementation under development by Jacobs University Bremen. Section 5 concludes the report wrapper. Subsequently, the WCPS specification text follows in its entirety, with numbering restarted to keep consistent with the twin document posted at the OGC portal. The reader is assumed to be familiar with WCS [4] on whose concepts and terminology WCPS build.

2 Coverage Relevant OGC Services

Traditionally, services providing access to spatio-temporal coverages in OGC have been addressed by the Web Coverage Service (Whiteside, Evans 2006). More recently, the Web Processing Service and the Sensor Web Enablement have joined. We briefly relate these three specifications.

2.1 WCS

The WCS 1.1 document [4] specifies that "A Web Coverage Service (WCS) describes and delivers multidimensional coverage data over the World Wide Web. This version of the Web Coverage Service is limited to describing and requesting grid (or "simple") coverages." The WCS coverage model defines coverages as bearing a locally unique name, the multi-dimensional data array itself, plus some technical metadata needed for coverage description and evaluation. The data array can be of 2, 3, or 4 dimensions, containing mandatory *x* and *y* axes and optional *z* and *t* axes. The array's spatio-temporal extent, its *domain*, consists of lower and upper limits per axis, expressed in some coordinate system. Each coverage has a list of coordinate reference systems associated in which it can be queried; requesting values in another geographic coordinate system than the one in which it is stored (or in the image coordiante system, directly using pixel coordinates) obviously will involve reprojection.

A coverage may use *null values* to denote pixel values that are unknown, undefined, or similar. To this end, a coverage knows one or more null values.

When scaling or reprojection are involved, usually resampling and interpolation have to be applied in the course of request evaluation. As several different interpolation methods are in common use among the perceived user communities, the WCS coverage model contains a list of interpolation method (one of them being distinguished as the default) from which a request can choose one to be applied. The current list of possible interpolation methods is *none*, *nearest* [*neighbor*], *linear*, *quadratic*, *cubic*. The effect of null values on interpolation can be controlled via the so-called *null resistance* parameter.

The common request structure for OGC standards, which WCS follows, is that a client first issues a GetCapabilities request to learn about a service's offerings and capabilities. Subsequently, the client performs retrieval based on this information. As a particularity of WCS, the GetCoverage request allows for such retrieval. In addition, a DescribeCoverage request is foreseen which delivers details information about coverages, as the WCS GetCapabilities request essentially only lists the coverages offered. Requests can be phrased as HTTP GET using key-value pair notation or as HTTP POST requests using XML syntax based on XML Schema definitions being part of the standard.

GetCoverage offers a fixed set of operations which can be combined in a request. These operators allow for spatial, temporal, and band subsetting, scaling, reprojection, and final result packaging, including data format encoding.

The following example shows a GetCoverage request against some satellite image time series coverage `ModisCube`, expressed in key-value pair notation:

```
http://myServer/wcsServlet?
   VERSION=1.1.0 & SERVICE=WCS & REQUEST=GetCoverage &
   COVERAGE=ModisCube &
   RANGESUBSET=nir;red
   SRS=EPSG:31464 &
   BBOX=4636000.0,5717000.0,4687000.0,5768000.0 & TIME=max &
   WIDTH=246 & HEIGHT=300 & DEPTH=1 &
   FORMAT=HDF-EOS &
   EXCEPTIONS=application/vnd.ogc.se_xml
```

The request extracts according to the given bounding box expressed in spatial reference system (SRS) EPSG:31464 and taking the most recent time slice of the cube. The result is scaled to 246x300 pixel and delivered in the HDF-EOS format. Any eventual error is to be reported back in XML.

2.2 WPS

The Web Processing Service (WPS) [4] allows wrapping any program interface into a SOAP-based XML structure; as such, WPS might be characterized as "geo SOAP". This approach is general enough to allow modeling of any kind of geo service. On the downside, its specification covers only the function signature (name and parameters) whereas the semantics is laid down in the fulltext description of the title and abstract field, hence it can only be interpreted by humans and is not machine-readable. This lack of built-in explicit semantics makes WPS difficult to employ for Semantic Web services and to introduce automatic service distribution and composition. WCPS, on the other hand, has a well-defined machine-understandable semantics which allows for such techniques.

2.3 SWE

The recently standardized Sensor Web Enablement (SWE) [4] standards group is emerging, with its sub-standards for Observation and Measurements, SensorML, TransducerML, Sensor Observation Service (SOS), Sensor Alert Service (SAS), Sensor Processing Service (SPS), and Web Notification Services (WNS). The term „sensor" is understood in a rather generic way encompassing, among others, cameras on board a satellite. With this perception WCS and SWE begin to overlap, and consequently harmonization of the standards is on the agenda of the resp. working groups now.

2.4 Comparative Synopsis

While traditionally the distinction between raster data (such as satellite images and elevation data) and sensors (delivering 1-D time series like water temperature or pollution indicators at one particular point of measurement) was felt to be clear, this distinction is breaking up recently. On the one hand, WCS, by extending its range of dimensionalities covered, tends to not exclude the 1-D case; on the other hand, sensor services more and more are understood to also include, for example, remote sensing instruments, which naturally leads to spatio-temporal data beyond 1-D.

Simultaneously with the growing outreach of OGC standards new requirements emerge from further communities which tend to be interested in Web-based geo services. An obvious extension is from traditional mapping (i.e, geodesy) into the complete range of earth sciences, incorporating geophysics, geochemsitry, and climate modeling, among many others. Not so obvious, but foreseeable already is the inclusion of domains, which per se have no connection with geo sciences, but use georeferenced data and, hence, need to integrate geographic data with their domain-specific information. As an example, OLAP / data warehousing dealing with business data analysis using multi-dimensional statistics may include geographid and demographic data in their analysis data set.

3 WCPS Requirements

Design of WCPS has been accomplished with several goals in mind, integrating best-practice knowledge from the fields of GIS technology, databases, programming languages, Web services, and imaging, among others. This has led to the following set of core requirements.

From a software engineering viewpoint, a standard defining the interface between clients and servers coming from independent developers needs to be concise, unambiguous, and understandable. This calls for a formal specification of the language's syntax and semantics. Notably this collides with the requirement of understandability of the service by implementers and service providers/users which do not have a high working knowledge of formal specifications – actually, the majority of the target group. The WCPS specification attempts a reasonable compromise between the rigidity of a formal specification and the understandability of a loose textual presentation. Examples have been added with each construct to aid understanding, and tutorial material is under development in parallel (to be made available via the EarthLook website, www.earthlook.org).

From a database viewpoint, the language in particular should offer declarativeness, data independence, optimizability, and it should be safe in evaluation [8]. A query language is said to be declarative if queries phrased in it describe the result structure, rather than specifying the algorithms to be executed for obtaining such results (*what* do I want vs. *how* is it computed). The maybe best known delcarative language is SQL [8]. Data independence decouples processing functionality from data and storage structures; this not only has proven advantageous for human readability of the languages, but moreover is generally recognized as being an indispensable prerequisite for the next requirement: Optimization of requests means replacing the evaluation algorithm for a given query by a semantically equivalent, but less costly variant. Optimization has a long and successful history in databases; for multi-dimensional raster data, it has been shown that optimization can improve query response time by orders of magnitude [11]. A database language which is safe in evaluation allows only those requests whose evaluation is guaranteed to terminate after a finit number of steps; this rules out one source of Denial of Service attacks.

From a GIS, imaging, and statistics viewpoint, the expressive power should be sufficient to allow formulation of current algorithms to a large extent. Notably this collides with the above requirements for declarativeness (many algorithms are formulated only in a procedural manner) and evaluation safety (with a language powerful enough for those algorithms termination cannot be guaranteed any longer). In the end, a tradeoff had to be made which, following established database tradition, ultimately favoured declarativeness and safety over computational power. This tradeoff is also influenced by the fact that the WPS standard indeed allows Turing-complete requests (consequently, without being safe in evaluation). Hence, with WCPS and WPS there are two complementary approaches available to implementers and service providers.

While raster processing languages exist in commercially available desktop products since long, such as MatLab (www.mathworks.com) and IDL On The Net (www.ittvis.com/ion), there is no such language available which fulfils all of the above requirements. The language which, to the best of our knowledge, comes closest to the envisaged goal is the rasdaman Array Algebra [2] and query language [10];

actually, rasdaman has very much influenced the design of WCPS and is being used for the reference implementation.

4 Implementation

In parallel to the specification of WCPS a reference implementation is pursued to ensure feasibility of the WCPS concepts and also to have some showcasing tools for promoting the standard *in spe* [5]. At the same time, the outcome is intended to become the reference implementation, continuously maintained and developed further by Jacobs University.

The service stack (Figure 2) consists of the WCPS interface implemented by a Java servlet, the rasdaman array server middleware, and a relational DBMS holding the raster data. The Java servlet accepts WCPS requests adhering to the WCPS XML schema specification, and returns the responses. Coverage results consist of an XML document accompanied by binary coverage data in the requested encoding format, encapsulated within a multipart/mixed HTTP response. This format allows on principle even a web browser to be used as a client, by uploading an XML request directly to the web service. Non-coverage results can be shipped back directly within the XML response.

The WCPS component translates a request into rasdaman's query language, rasql [10], and hands this to rasdaman for processing. The results obtained from rasdaman are MIME-encoded and shipped back to the client, together with the XML-encoded manifest describing them.

The rasdaman ("raster data manager") raster server extends standard relational databases with multidimensional raster data of unlimited size [1], [3]. Its query language, rasql, extends standard SQL with a raster language. Rasql expressions are optimized on server side and executed against the database where the raster objects are stored partitioned in sets of blobs (binary large objects). The relational database, then, for a Terabyte-size raster objects holds a million blobs of about a Megabyte size each – a value which has proven suitable for high request throughput.

Rasdaman on principle can plug into virtually any database system and is commercially operational on top of Oracle, Informix, DB2, and PostgreSQL. For the purpose of the WCPS development the open-source system PostGreSQL is used which has proven competitive indeed against its commercial counterparts.

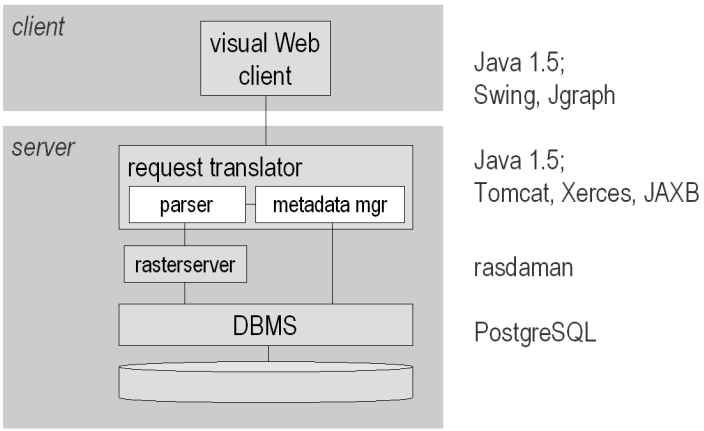

Figure 2
WCPS reference implementation architecture and technology used

5 Conclusion and Future Work

Web-based analysis and processing of large raster archives is an emerging technology, and developing an open, unifying standard in this early phase is expected to comprise guidelines to vendors and ultimately lead to high-value, interoperable products.

WCPS attempts to provide an open-ended framework for submitting requests of unlimited complexity for processing on server-side and returning only the results, by doing so in a formally defined behavior. The design of WCPS actually embodies the long-standing experience with the rasdaman array DBMS in its applications as geo raster server, hence we are confident about feasibility of WCPS. Still, there is a need for in-depth evaluation and assessment unde real-life conditions in as many application domains (and combinations thereof) as possible.

Future work is characterized by the evolution of the standards, and by gaining more experience by setting up real-life services. At the time of this writing, WCPS has reached a first stable state. WCS is in the transition from version 1.1 to version 1.2; as WCPS relies on the concepts of WCS (in aprticular: its conceptual coverage model), this will require continuous adaptation of WCPS to the latest WCS version to maintain lock-step synchronization. In the course of WCS restructuring into a core document as the basis and add-on plus application profile specifications on top of it, WCPS will become an add-on to WCS in terms of the specification document structure.

Among the missing elements of the WCS specification is a comprehensive Conformance Clause, best based on a testbed implementation.

In the course of the EarthLook project (www.earthlook.org) a realistic WCPS demonstration scenario is being set up; exposing this to real-life users is expected to give further insights and will probably raise new questions.

On the conceptual side as well regarding implementation further studies are required. Issues in the domain of array databases, such as advanced optimization, parallelization, and intelligent service distribution are on the research agenda.

Report References

[1] P. Baumann: On the Management of Multidimensional Discrete Data. VLDB Journal 4(3)1994, Special Issue on Spatial Database Systems, pp. 401-444

[2] P. Baumann: A Database Array Algebra for Spatio-Temporal Data and Beyond. Proc. Next Generation IT and Systems (NGITS), Zikhron Yaakov, Israel, 1999, pp. 76 - 93

[3] P. Baumann: Large-Scale Raster Services: A Case for Databases. Invited keynote, 3rd Intl Workshop on Conceptual Modeling for Geographic Information Systems (CoMoGIS), Tucson, USA, 6 - 9 November 2006. In: John Roddick et al (eds): Advances in Conceptual Modeling - Theory and Practice, pp. 75 - 84

[4] Botts, M., Robin, A., Davidson, J., Simonis, I. (eds.): Sensor Web Enablement Architecture. OGC document 06-021r1, 2006

[5] G. Chulkov: Architecture and Implementation of a Web Coverage Processing Service Suing a Database Backend. Bachelor Thesis, Jacobs University Bremen, May 2006

[6] ISO TC211: Geographic information – Schema for coverage geometry and functions. ISO DIS 19123, TC 211/SC/WG 2, 2002

[7] C. Kottman: OGC Abstract Specification Topic 6: The Coverage Type and its Subtypes. OGC document 00-106, 2000

[8] H.G. Molina, J. Ullman, J. Widom: Database System Implementation. Prentice Hall 2000

[9] n.n.: Web Processing Service. OGC document 05-007r4, 2005

[10] n.n.: Rasql Query Language Guide. Rasdaman GmbH, 2001

[11] R. Ritsch: Optimization and Evaluation of Array Queries in Database Management Systems. PhD Thesis, Technische Universität München, 1999

OpenGIS Interface:
Web Coverage Processing Service (WCPS)

1 Scope

This document specifies how a Web Coverage Processing Service (WCPS) allows to describe, request, and deliver multi-dimensional grid coverage data over the World Wide Web.

Grid coverages have a domain comprised of regularly spaced locations along an arbitrary number of axes. Specific semantics is associated with spatio-temporal axes; A coverage can optionally have an x axis, a y axis (which, if both present and equipped with a coordinate reference system, **shall** bear a common coordinate reference system), a time axis, and an elevation axis. A coverage's grid point (i.e., cell) data types define, at each location in the domain, either a single (scalar) value (such as elevation), or an ordered series of values (such as brightness values in different parts of the electromagnetic spectrum).

Result coverages can be transmitted directly or made available for download by URLs communicated to the client.

The Web Coverage Processing Service provides three operations: *GetCapabilities*, *DescribeCoverage*, and *ProcessCoverage*. The *GetCapabilities* operation, like in WCS, returns an XML document describing the service and brief descriptions of the data collections from which clients may request coverages; additionally WCPS specific processing service capabilities are delivered. Clients would generally run the *GetCapabilities* operation when opening a session with some particular server and cache its result for use throughout the session.

Like in WCS, the *DescribeCoverage* operation lets clients request a full description of one or more coverages served by a particular WCPS server. The server responds with an XML document that fully describes the identified coverages.

The *ProcessCoverage* operation allows to process and analyse coverages and coverage sets stored on the server as well as to extract information – both grid data and metadata – from coverages. To this end, requests are phrased in a formally defined processing language which supports coverage expressions of unlimited complexity. Result coverages can be transmitted directly back to the client or made available for download by URLs communicated to the client.

Coverages advertised by a service can be stored on the corresponding server, but the service may well itself rely on external data sources to substantiate the portfolio. In any case, the appearance towards the service clients always is one homogeneously accessible coverage offering.

For future versions it is intended, to extend WCPS to incorporate further coverage types defined in the OpenGIS Abstract Specification (Topic 6, "The Coverage Type", OGC document 00-106), in synchronization with WCS.

2 Conformance

Conformance with this OGC Implementation Specification may be checked using all the relevant tests specified in Annex D.

3 Normative references

The following normative documents contain provisions that, through reference in this text, constitute provisions of this specification. 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 2045 (November 1996), Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies, Freed, N. and Borenstein N., eds., <http://www.ietf.org/rfc/rfc2045.txt>

IETF RFC 2616 (June 1999), Hypertext Transfer Protocol – HTTP/1.1, Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T., eds., <http://www.ietf.org/rfc/rfc2616.txt>

IETF RFC 2396 (August 1998), Uniform Resource Identifiers (URI): Generic Syntax, Berners-Lee, T., Fielding, N., and Masinter, L., eds., <http://www.ietf.org/rfc/rfc2396.txt>

ISO 19105: Geographic information — Conformance and Testing

ISO 19123, Geographic Information — Coverage Geometry and Functions

OGC 02-023r4, OpenGIS Geography Markup Language (GML) Implementation Specification, v3.00 <http://www.opengis.org/techno/documents/02-023r4.pdf>

OGC AS 0, The OpenGIS Abstract Specification Topic 0: Overview, OGC document 99-100r1 <http://www.opengis.org/techno/abstract/99-100r1.pdf>

OGC AS 12, The OpenGIS Abstract Specification Topic 12: OpenGIS Service Architecture (Version 4.2), Kottman, C. (ed.), <http://www.opengis.org/techno/specs.htm>

OGC 05-096r1, GML 3.1.1 grid CRSs profile, v1.0.0, <http://portal.opengeospatial.org/files/?artifact_id=13205>

XML 1.0, W3C Recommendation 6 October 2000, Extensible Markup Language (XML) 1.0 (2nd edition), World Wide Web Consortium Recommendation, Bray, T., Paoli, J., Sperberg-McQueen, C.M., and Maler, E., eds., <http://www.w3.org/TR/2000/REC-xml>

W3C Recommendation 2 May 2001: XML Schema Part 0: Primer,
<http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/>

W3C Recommendation 2 May 2001: XML Schema Part 1: Structures,
<http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/>

W3C Recommendation 2 May 2001: XML Schema Part 2: Datatypes,
<http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/>

4 Terms and definitions

For the purposes of this document, the terms and definitions given in the above references (in particular: WCS [4]) apply, and additionally the following terms.

4.1
axis
a totally ordered set of values which can be used for coordinate referencing in a coverage. A coverage may have any number of axes, of which a subset may carry spatio-temporal semantics as defined in WCS [4]. Axes without spatial or temporal semantics are called “abstract”.

4.2
bounding box
the extent of a coverage, consisting of the spatio-temporal bounding box as defined in WCS [4] plus optional abstract dimension extents.

4.3
cell
a data element of a coverage which is uniquely identified by its grid point coordinate, i.e., its spatio-temporal position; each cell within a coverage’s bounding box contains a value of the structure specified in the coverage’s range definition. Depending on the application, cells commonly are also known as “pixel”, “voxel”, etc.

4.4
domain
the extent of a coverage, defined by its lower and upper bound per axis. The domain concept basically is used as defined by WCS, with two modifications: firstly spatial and temporal axes are optional, and secondly any number of abstract axes can appear.

whereby axes with a geo semantics have geographic coordinates, a time axis has time coordinates. Both spatial and temporal axes are optional in a WCPS coverage.

4.5
general domain
the extent of a WCPS coverage, which can have both spatio-temporal and abstract axes. Spatial, temporal, and abstract axes all are optional in a WCPS coverage, as long as the coverage has at least one axis.

4.6
identifier
a string, acting as an identifier in some context, which **shall** adhere to the specification of the syntax element "identifier" defined in Annex A.1.

5 Conventions

5.1 Symbols (and abbreviated terms)

Most of the abbreviated terms listed in Subclause 5.1 of the OWS Common Implementation Specification [OGC 05-008] also apply to this document.

Further, this document assumes familiarity with the terms and concepts of the Web Coverage Service Implementation Specification [4].

5.2 UML notation

All the diagrams that appear in this specification are presented using the Unified Modeling Language (UML) static structure diagram, as described in Subclause 5.2 of the OGC Web Services Common Implementation Specification [OGC 05-008].

5.3 Platform-neutral and platform-specific specifications

As specified in Clause 10 of OGC Abstract Specification Topic 12 "OpenGIS Service Architecture" (which contains ISO 19119), this document includes both Distributed Computing Platform-neutral and platform-specific specifications. This document first specifies each operation request and response in platform-neutral fashion. This is done using a table for each data structure, which lists and defines the parameters and other data structures contained. These tables serve as data dictionaries for the UML model in Annex C, and thus specify the UML model data type and multiplicity of each listed item.

Example Platform-neutral specifications are contained, e.g., in Subclause 10.2.

The specified platform-neutral data could be encoded in many alternative ways, each appropriate to one or more specific DCPs. This document now specifies encoding appropriate for use of HTTP GET transfer of operations requests (using KVP encoding), and for use of HTTP POST transfer of operations requests (using KVP or XML or SOAP encoding). However, the same operation requests and responses (and other data) could be encoded for other specific computing platforms.

Example Platform-specific specifications for KVP encoding are contained, e.g., in Subclause 10.2.5.

Example Platform-specific specifications for XML encoding are contained in Subclause 10.2.6.

5.4 Data dictionary tables

The UML model data dictionary is specified herein in a series of tables. The contents of the columns in these tables are described in Table 1. The contents of these data dictionary tables are normative, including any table footnotes.

Table 1 — Contents of data dictionary tables

Column title	Column contents
Names (left column)	Two names for each included parameter or association (or data structure). The first name is the UML model attribute or association role name. The second name uses the XML encoding capitalization specified in Subclause 11.6.2 of [OGC 05-008]. Some names in the tables may appear to contain spaces, but no names contain spaces.
Definition (second column)	Specifies the definition of this parameter (omitting un-necessary words such as "a", "the", and "is"). When the parameter value is the identifier of something, not a description or definition, the definition of this parameter should read something like "Identifier of TBD".
Data type and value (third column) *or* Data type (when no second items are included in rows of table)	Normally contains two items: The mandatory first item is often the data type used for this parameter, using data types appropriate in a UML model, in which this parameter is a named attribute of a UML class. Alternately, the first item can identify the data structure (or class) referenced by this association, and reference a separate table used to specify the contents of that class (or data structure). The optional second item in the third column of each table should indicate the source of values for this parameter, the alternative values, or other value information, unless the values are quite clear from other listed information.
Multiplicity and use (right or fourth column) *or* Multiplicity (when no second items are included in rows of table)	Normally contains two items: The mandatory first item specifies the multiplicity and optionality of this parameter in this data structure, either "One (mandatory)", "One or more (mandatory)", "Zero or one (optional)", or "Zero or more (optional)". (Yes, these are redundant, but we think ISO wants this information.) The second item in the right column of each table should specify how any multiplicity other than "One (mandatory)" shall be used. When that parameter is optional, under what condition(s) shall that parameter be included or not included? When that parameter can be repeated, for what is that parameter repeated?

When the data type used for this parameter, in the third column of such a table, is an enumeration or code list, all the values specified by a specific OWS shall be listed, together with the meaning of each value. When this information is extensive, these values and meanings should be specified in a separate table that is referenced in the third column of this table row.

6 Basic service elements

6.1 Introduction

This clause describes aspects of Web Coverage Processing Server behavior (more generally, of OGC Web Service behavior) that are independent of particular operations, or that are common to several operations or interfaces.

6.2 Version numbering and negotiation

6.2.1 Version number form

The published specification version number contains three positive integers, separated by decimal points, in the form "x.y.z". The numbers "y" and "z" will never exceed 99. Each OWS specification is numbered independently.

6.2.2 Version changes

A particular specification's version number **shall** be changed with each revision. The number **shall** increase monotonically and **shall** comprise no more than three integers separated by decimal points, with the first integer being the most significant. There may be gaps in the numerical sequence. Some numbers may denote experimental or interim versions. Service instances and their clients need not support all defined versions, but **shall** obey the negotiation rules below.

6.2.3 Appearance in requests and in service metadata

The version number appears in at least two places: in the Capabilities XML describing a service, and in the parameter list of client requests to that service. The version number used in a client's request of a particular service instance **shall** be equal to a version number which that instance has declared it supports (except during negotiation as described below). A service instance may support several versions, whose values clients may discover according to the negotiation rules.

6.2.4 Version number negotiation

A Client may negotiate with a Service Instance to determine a mutually agreeable specification version. Negotiation is performed using the **GetCapabilities** operation [see Clause 8] according to the following rules.

All Capabilities XML **shall** include a protocol version number. In response to a **GetCapabilities** request containing a version number, an OGC Web Service **shall** either respond with output that conforms to that version of the specification, **or** negotiate a mutually agreeable version if the requested version is not implemented on the server. If no version number is specified in the request, the server **shall** respond with the highest version it understands and label the response accordingly.

Version number negotiation occurs as follows:

a) If the server implements the requested version number, the server **shall** send that version.

b) If a version unknown to the server is requested, the server **shall** send the highest version it knows that is less than the requested version.

c) If the client request is for a version lower than any of those known to the server, then the server **shall** send the lowest version it knows.

d) If the client does not understand the new version number sent by the server, it **may** either cease communicating with the server **or** send a new request with a new version number that the client does understand but which is less than that sent by the server (if the server had responded with a lower version).

e) If the server had responded with a higher version (because the request was for a version lower than any known to the server), and the client does not understand the proposed higher version, then the client **may** send a new request with a version number higher than that sent by the server.

The process is repeated until a mutually understood version is reached, or until the client determines that it will not or cannot communicate with that particular server.

Example 1 - Server understands versions 1, 2, 4, 5 and 8. Client understands versions 1, 3, 4, 6, and 7. Client requests version 7. Server responds with version 5. Client requests version 4. Server responds with version 4, which the client understands, and the negotiation ends successfully.

Example 2 - Server understands versions 4, 5 and 8. Client understands version 3. Client requests version 3. Server responds with version 4. Client does not understand that version or any higher version, so negotiation fails and client ceases communication with that server.

The negotiated version parameter **shall** be supplied with ProcessCoverage requests.

6.3 General HTTP request rules

6.3.1 Overview

At present, the only distributed computing platform (DCP) explicitly supported by OGC Web Services is the World Wide Web itself, or more specifically Internet hosts implementing the Hypertext Transfer Protocol (HTTP). Thus the Online Resource of each operation supported by a service instance is an HTTP Uniform Resource Locator (URL). The URL may be different for each operation, or the same, at the discretion of the service provider. Each URL **shall** conform to the description in [HTTP] but is otherwise implementation-dependent; only the parameters comprising the service request itself are mandated by the OGC Web Services specifications.

HTTP supports two request methods: GET and POST. One or both of these methods may be defined for a particular OGC Web Service type and offered by a service instance, and the use of the Online Resource URL differs in each case.

An Online Resource URL intended for HTTP GET requests is in fact only a URL prefix to which additional parameters must be appended in order to construct a valid Operation request. A URL prefix is defined as an opaque string including the protocol, hostname, optional port number, path, a question mark '?', and, **optionally**, one or more server-specific parameters ending in an ampersand '&'. The prefix uniquely

identifies the particular service instance. For HTTP GET, the URL prefix **shall** end in either a '?' (in the absence of additional server-specific parameters) or a '&'. In practice, however, Clients **should** be prepared to add a necessary trailing '?' or '&' before appending the Operation parameters defined in this specification in order to construct a valid request URL.

An Online Resource URL intended for HTTP POST requests is a complete and valid URL to which Clients transmit encoded requests in the body of the POST document. A WCPS server **shall not** require additional parameters to be appended to the URL in order to construct a valid target for the Operation request.

6.3.2 Key-value pair encoding (GET or POST)

6.3.2.1 Overview

Using Key-Value Pair encoding, a client composes the necessary request parameters as keyword/value pairs in the form "keyword=value", separated by ampersands ('&'), with appropriate encoding [6] to protect special characters. The resulting query string may be transmitted to the server via HTTP GET or HTTP POST, as prescribed in the HTTP Common Gateway Interface (CGI) standard [IETF RFC 2616].

Table 2 summarizes the request parameters for HTTP GET and POST.

Table 2 – Parts of a Key-Value Pair OGC Web Service Request

URL Component	Description
http://host[:port]/path	URL of service operation. The URL is entirely at the discretion of the service provider.
{name[=value]&}	The query string, consisting of one or more standard request parameter name/value pairs defined by an OGC Web Service. The actual list of required and optional parameters is mandated for each operation by the appropriate OWS specification.
Notes: [ ] denotes 0 or 1 occurrence of an optional part; {} denotes 0 or more occurrences.	

A request encoded using the HTTP GET method interposes a '?' character between the service operation URL and the query string, to form a valid URI which may be saved as a bookmark, embedded as a hyperlink, or referenced via Xlink in an XML document.

6.3.2.2 Parameter ordering and case

Parameter names **shall not** be case sensitive, but parameter values **shall** be case sensitive.

NOTE In this document, parameter names are typically shown in uppercase for typographical clarity, not as a requirement.

Parameters in a request **may** be specified in any order.

An OGC Web Service **shall** be prepared to encounter parameters that are not part of this specification. In terms of producing results per this specification, an OGC Web Service **shall** ignore such parameters.

6.3.2.3 Parameter lists

Parameters consisting of lists **shall** use the comma (",") as the delimiter between items in the list.

Example `parameter=item1,item2,item3`

Multiple lists **shall** be specified as the value of a parameter by enclosing each list in parentheses ("(", ")")

Example `parameter=(item1a,item1b,item1c),(item2a,item2b)`

If a parameter name or value includes a space or comma, it shall be escaped using the URL encoding rules [6].

6.3.3 XML encoding

Clients **may** also encode requests in XML for transmission to the server using HTTP GET or HTTP POST. The XML request **shall** conform to the schema corresponding to the chosen operation, and the client **shall** send it to the URL listed for that operation in the server's Getabilities response, in accordance with HTTP POST [7]).

NOTE To support SOAP messaging, clients need only enclose the XML document *ogcdoc* in a SOAP envelope as follows:

```
<env:Envelope xmlns:env="http://www.w3.org/2001/09/soap-envelope">
    <env:Body>
        ogcdoc
    </env:Body>
</env:Envelope>
```

6.4 General HTTP response rules

Upon receiving a valid request, the service **shall** send a response corresponding exactly to the request as detailed in the appropriate specification. Only in the case of Version Negotiation (described above) may the server offer a differing result.

Upon receiving an invalid request, the service **shall** issue a Service Exception as described in Subclause 6.5 below.

NOTE As a practical matter, in the WWW environment a client should be prepared to receive either a valid result, or nothing, or any other result. This is because the client may itself have formed a non-conforming request that inadvertently triggered a reply by something other than an OGC Web Service, because the service itself may be non-conforming, etc.

6.5 Service exceptions

Upon receiving an invalid request, the service **shall** issue a Service Exception XML message to describe to the client application or its human user the reason(s) that the request is invalid.

Service Exception XML **shall** be valid according to the Service Exception XML Schema in Subclause B.6. In an HTTP environment, the MIME type of the returned XML **shall** be "`application/vnd.ogc.se_xml`". Specific error messages can be included either as chunks of plain text or as XML-like text containing angle brackets ("<" and ">") if included in a character data (CDATA) section as shown in the example of Service Exception XML in Subclause B.6.

Service Exceptions **may** include exception codes as indicated in Subclause B.6. Servers **shall not** use these codes for meanings other than those specified. Clients **may** use these codes to automate responses to Service Exceptions.

7 Conceptual coverage model

7.1 Overview

The coverage model of WCPS (see Subclause 7.2) relies on the coverage model of WCS [4]. Some restrictions and extensions apply wrt. WCS which are listed below in Subclause 7.2 and 7.5, resp.; no further deviation to the WCS coverage model exists beyond those listed. Subclause 7.3 describes, based on this coverage model, the constituents a WCPS coverage has by defining a set of coverage probing functions. For the comprehensive description of the data structures, refer to Annex B (XML) and Annex C (UML).

7.2 Coverage model

7.2.1 Coverages

A coverage consists of a set of locations bearing some value. Following the mathematical notion of a function that maps elements of a domain (here: spatio-temporal and/or abstract coordinates) to a range (here: "pixel", "voxel", … values), the set of coverage locations bearing values is called the coverage domain while the set of possible values, i.e., the coverage value data type, is called the coverage range.

A coverage domain with its set of locations, also termed (grid) *cell positions* or *coordinates*, is aligned along some d-dimensional grid where $d>0$ is called the coverage's *dimensionality*. The coordinate space, i.e. the set of all possible coordinates, is spanned by d independent axes. An axis is identified by its name which is unique within the coverage. The set of all axis names of a coverage c is obtained via the function axisNameSet(c).

NOTE In its current version, the WCPS coverage model is constrained to equally spaced grids, meaning that the distance between any two adjacent grid points in a coverage is constant per axis. This notion of a coverage - which intrinsically resembles that of a multi-dimensional array in programming languages – may be extended in future versions of this document.

7.2.2 Axes

Each axis has an axis type associated, which is one of the elements listed in Table 3. A coverage can have at most one *x*, *y*, *z*, and *time* axis, but can have any number of axes of type *abstract*.

Table 3 – Coverage domain axis types.

Axis type	Meaning
X	East-West extent, expressed in the coverage's CRS
Y	North-South extent, expressed in the coverage's CRS
z	Geographical elevation, i.e., height or depth
time	Time; coordinates are expressed as time strings according to [10]
abstract	None of the above; no spatio-temporal semantics is associated with such an axis

Each axis has one or more coordinate reference systems (CRSs) associated. One ImageCRS [05-096r1] common to all axes of a given coverage is mandatory, given by imageCrs(C). Additionally, any number of further CRSs can be associated with a coverage axis, given by the set crsSet(C,a). Image CRS and further CRSs together determine the set of CRSs which can be used in coordinate-sensitive operations.

NOTE An image CRS always allows to address a coverage in all axes. Fur the other CRSs, however, several CRSs together may be necessary to fully address a coverage – for example, WGS84 only knows x and y and thus does not allow to specify z and t coordinates in a 4-D x/y/z/t climate model.

NOTE CRSs are specified in operations using URNs. Some standards use EPSG notation for geospatial coordinates; this approah hasno tbeen adopted to achieve uniformity across spatial, temporal, and abstract axes.

The WCPS service does **not need** to publish the mapping between coordinates of the different supported CRSs.

7.2.3 Locations

A cell location is unambiguously defined by listing its coordinate position for every axis. A location L is a set $L_C = \{ (a,c,p) \mid a \in \text{axisNameSet}(C),\ c \in \text{crsSet}(C),\ p \in \text{AxisPointValues} \}$ consisting of axis names, the coordinate system used, and a coordinate relative to this axis and this CRS; each of the coverage's axis name shall appear exactly once in this set. The set AxisPointValues is a generalization of numeric and string values that allows to express all kind of coordinates, including geographic floating-point coordinates and date/time strings.

Example For axis type *time*, encoding follows ISO 8601:2000 [10] as described in WCS [4] Table 16, 17 and owsTime (that is, the possible values are ASCII strings). For an image CRS, encoding will be integer for all axis types, and for x/y type geographic coordinates it will usually be float.

On each axis a total ordering relation "≤" **shall** be available under all CRSs used.

Example On a time axis, this ordering relation will yield true for the following comparison:

```
  "Sun Jan 1 23:59:59 CET 2006"
≤ "Tue Dec 5 22:17:48 CET 2006"
```

Along each axis a coverage is delimited by a lower and upper bound value, these border values being part of the coverage extent. Location addresses always are relative to a particular coverage.

7.2.4 Domain

The set of all locations contained in a coverage forms its domain. A domain's location set always is non-empty and is such that it can be described, for each axis, by a lower and upper bound (lo,hi) expressed in one of the coverage's CRSs applicable for this axis where $lo \le hi$. The domain of a coverage consists of exactly those cell locations where, for each of its axes, its location l is contained in the closed interval given by lo and hi: $lo \le l \le hi$.

To differentiate from the traditional 2-D domain, the multi-dimensional domain concept employed by this document is termed General Domain.

For a coverage C, function generalDomain() describes its domain structure, which is a set of axis descriptions consisting of axis name, axis type, CRS used, and the lower and upper bound of the coverage domain expressed in the CRS at hand: generalDomain(C) = { (a,t,c,lo,hi) | $a \in$ axisNameSet(C), $t \in$ {*x*,*y*,*z*,*time*,*abstract*}, $c \in$ crsSet(C), lo,$hi \in$ AxisPointValues, $lo \le hi$ }

A location L is inside the general domain of a coverage C if its coordinates are inside the domain extent under all CRSs supported:

Let

C be a coverage,
L_C be a location wrt. coverage C
with L_C = { (a,c,p) | $a \in$ axisNameSet(C), $c \in$ crsSet(C), $p \in$ AxisPointValues },
G_C be the general domain of coverage C
with G_C = { (a,t,c,lo,hi) | $a \in$ axisNameSet(C), $t \in$ {*x*,*y*,*z*,*time*,*abstract*}, $c \in$ crsSet(C), lo,$hi \in$ AxisPointValues, $lo \le hi$ }.

Then,

L_C inside G_C
if and only if
for all (a,c,p)$\in L_C$ there is some lo,$hi \in$ AxisPointValues such that:
(a,t,c,lo,hi)$\in G_C$ and $lo \le p \le hi$ relative to CRS c

NOTE The GeneralDomain of WCPS generalizes the Domain of WCS (see Subclause 7.5).

7.2.5 Range values and types

The value associated with a particular cell location within a coverage, in short: its cell value, can be obtained with function value(C,l_C) which is defined for every location $l_C \in$ imageCrsDomain(C) and l_C inside generalDomain(C).

For addressing in some GeneralDomain CRS, coordinates are server-internally translated to image CRS coordinates and rounded towards the nearest cell location if necessary.

All cell values of a coverage share the same type, the coverage's range type. Admissible types consist of named components called fields; each field is identified by a field name unique for the coverage on hand and bears one of the (atomic) numeric or Boolean types enumerated in the set RangeFieldTypes (see Table 4):

RangeFieldTypes = { boolean, char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, double, complex, complex2 }

Table 4 – Coverage range field data types.

Cell 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	Single precision complex number
complex2	Double precision complex number

NOTE This is a restriction over WCS [4], see Subclause 7.4.

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

NOTE Components of a range list are also known as "bands".

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

rangeType(C) = { (f,t) | $f \in$ rangeFieldNames(C), $t \in$ RangeFieldTypes }

7.2.6 Null and interpolation

Specific range values may serve as null values (WCS [4]). The set of a coverage's values to be interpreted as null is obtained via function nullSet().

Operations sometimes require interpolation. Available interpolation types are, adopted from WCS [4]: *linear*, *quadratic*, and *cubic*.

Behavior of interpolation methods in face of null values is described by the interpolation method's null resistance (WCS [4]).

Function interpolationSet(C) returns a set of pairs (im,nr) where im indicates the interpolation type and nr the null resistance employed.

For both null values and interpolation method defaults are associated with a coverage, which can be obtained through functions nullDefault() and interpolationDefault().

7.3 Coverage probing functions

A set of so-called *probing functions* allows to extract the constituents listed above from a given coverage. These functions are not part of the interface specification, but serve for the sole purpose of defining the semantics of **ProcessCoverage** requests in Clause 10.

Table 5 summarises the probing functions available.

Table 5 – Coverage probing functions.

Coverage characteristic	Probing function for some coverage C	Comment
Identifier	identifier(C)	For original coverages only, not for processed coverage results
Cell values	value(C,p) for all $p \in$ imageCrsDomain(C)	The coverage cell ("pixel"), "voxel", …) values themselves, of data type rangeType(C)
Domain axis set	axisSet(C)	Set of all of the coverage's axis names
Domain axis type	axisType(C, a)	Axis type
Image CRS	imageCRS(C)	Image CRS of the coverage, allowing direct grid cell point addressing
Domain extent of coverage, expressed in Image CRS	imageCrsDomain(C)	Extent of the coverage in (integer) grid cell coordinates, relative to the coverage's Image CRS[1]; essentially, the set of all point coordinates inside the coverage
Domain extent of coverage along axis, expressed in Image CRS	imageCrsDomain(C, a) for some $a \in$ axisSet(C)	Extent of the coverage in (integer) grid cell coordinates, relative to the coverage's Image CRS, for a given axis; essentially, the set of all values inside the extent interval
CRS set	crsSet(C, a) for some $a \in$ axisSet(C)	Set of all CRSs which can be used for addressing in the given axis; following WCS [4] it **shall** not include the Image CRS
General domain extent of coverage along axis, expressed in arbitrary CRS	generalDomain(C, a, c) for some $a \in$ axisSet(C) and some $c \in$ crsSet(C)	General domain of the coverage, expressed in one of its CRSs, for a given (spatial, temporal, or abstract) axis
Range data type	rangeType(C)	The data type of the coverage's cells
Range field type	rangeFieldType(C, f) for some $f \in$ rangeFieldNames(C)	The data type of one coverage range field
Range field name set	rangeFieldNames(C)	Set all of the coverage's range fields names
Default null value	nullDefault(C)	Optional default null value, to be used whenever an operation returns a null range value
Null value set	nullSet(C, r) for all $r \in$ rangeType(C)	The set of all values that represent null as coverage range field value
Default interpolation method	InterpolationDefault(C, r) for all $r \in$ rangeType(C)	Default interpolation method, per coverage field

[1] Note that the same image CRS is supported by all axes of a coverage.

Interpolation method set	interpolationSet(*C*, *r*) for all *r* ∈ rangeType(*C*)	All interpolation methods applicable to the particular coverage range field; must list at least the default interpolation method
Interpolation type	interpolationType(*im*) for all *im*∈ interpolationList(*C*)	Interpolation type of a particular interpolation method; possible values are listed in WCS [4] Table I.7
Null resistance	nullResistance(*im*) for all *im*∈ interpolationList(*C*)	Null resistance level of a particular interpolation methods; possible values are listed in WCS [4] Table I.8

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, intersection. 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)

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 scope of WCPS.

7.4 Restrictions relative to WCS coverage model

The following features of a WCPS coverage are restricted as compared to the WCS 1.1 coverage concept.

- Range base types in WCPS are constrained to the set indicated in Table 4 while WCS does not constraint the atomic) base types.

  NOTE This is necessary to fix the semantics of operations on the range types.

  NOTE In practice this should hardly pose a restriction, as at least all numeric types occurring in the applications perceived are provided.

- Atomic range list fields:

  Range list fields in WCPS are atomic.

  In WCS they alternatively can be composites forming multi-dimensional arrays.

  NOTE This restriction is intended to be lifted in a future version of WCPS.

- Mandatory image CRS:

  In WCPS an image CRS is is mandatory for a coverage; thus, a coverage always allows addressing its grid cell points by their array (“pixel”, “voxel”, …) coordinates.

  In WCS an image CRS is optional for a coverage (WCS [4] Table 14).

7.5 Extensions relative to WCS coverage model

The following features of a WCPS coverage extend the WCS coverage concept.

- Extended domain:

  WCS allows 2/3/4-D coverages with axes forming a subset of x, y, z, and t, whereby x and y are mandatory. WCPS adds an arbitrary number of abstract (i.e., non-spatiotemporal) axes and allows coverage domains of any number of dimensions greater or equal to one. Spatial, temporal, and abstract axes all are optional.

  NOTE In particular, WCPS does not require a coverage to have spatial axes while WCS does so, i.e., every coverage must have an x and a y axis in WCS. While non-spatial coverages are not the primary focus of WCPS, it does not exclude them.

  Example Possible 2-D coverages resulting from slicing a 4-D x/y/z/t climate model are slices having x/y, x/z, x/t, y/z, z/t axes, resp.

  NOTE From a conceptual viewpoint, single (scalar) values can be modelled as zero-dimensional coverages. However, this theoretical completion of the dimension numbers has no practical relevance.

  Consequently, WCPS extends the WCS Domain data type which is used by **DescribeCoverage** and **ProcessCoverage** requests (see Annex B). To this end, the WCS Domain element, which describes the spatio-temporal extent of a coverage, in WCPS is replaced by the element GeneralDomain. This way, GeneralDomain contains the WCS Domain as a special case.

  The extensions of GeneralDomain consist of the following items:

  - additional element AbstractDomain. This optional element allows to define an arbitrary number of axes without spatio-temporal semantics.
  - additional element ElevationDomain. This optional element allows to define an elevation axis.
  - SpatialDomain is optional in WCPS (as opposed to being mandatory in WCS).

    If present the SpatialDomain element does not need to have both x and y coordinates, but may have just one of x and y present.

  NOTE If a coverage has only axes as known to WCS (i.e., one of the axis combinations x/y, x/y/t, x/y/z, x/y/z/t) then the corresponding WcpsDomain is structurally identical to the WCS Domain element. In this sense, WCPS is downward compatible with WCS.

- Default null value:

  Each coverage has an optional metadata element containing the value to be used whenever some operation applied to coverage grid cells yields null. This value must be of the coverage's range type.

  Whenever request evaluation yields a null value for some grid cell, then this default value **shall** be used. If the default null value is not available then the

server shall assume 0 for numeric range field types and false for boolean range field types values as null values.

NOTE While it is not required that such a value is defined (for compatibility with the current WCS version [4]) it is recommended to use it.

NOTE While it is not required by this standard that the default null value is one of the null values listed in the known null values an implementation may want to ensure this for overall coherence.

- CRS per axis:

  In WCPS the CRSs are tied to the axes, respecting the fact that many CRSs allow only to express a subset of the axes a coverage may bear, so that consequently more than one CRSs must be used for a complete coverage location addressing. In WCS, the CRS set is associated with the coverage as a whole.

7.6 WCS compatibility statement

The WCPS standard is designed so as to be downward compatible to WCS. This means that any WCPS coverage that has only a spatio-temporal extent and no abstract axes has a structure, from a WCPS client view, which is identical to the structure conveyed by the same coverage to a WCS client by a WCS server. In particular, in such a case

- the result of a WCPS **DescribeCoverage** request is identical to the result of a WCS **DescribeCoverage** request, except that the name of the domain element in the response in WCPS is “GeneralDomain” and in WCS is “Domain”;

- the response to a WCPS **ProcessCoverage** request has the same structure as a WCS **GetCoverage** request;

- the WCPS metadata retrieval functions (see Subclause 10.3.8) deliver information compatible and coherent with the information delivered by a WCS/WCPS **DescribeCoverage** request.

8 GetCapabilities operation

The mandatory **GetCapabilities** operation allows WCS clients to retrieve service metadata from a WCS server. The response to a **GetCapabilities** request shall be an XML document containing service metadata about the server, usually including summary information about the data collections from which coverages may be requested. This clause specifies KVP and XML encoding of a GetCapabilities request and the XML document that a WCS server shall return to describe its capabilities.

The WCPS **GetCapabilities** operation is identical to the WCS GetCapabilities operation [4].

Each server **shall** implement the **GetCapabilities** operation.

9 DescribeCoverage operation

Once a client has obtained summary descriptions of a WCS server's available coverages, it may be able to make **ProcessCoverage** requests. However, in most cases the client will need to issue a **DescribeCoverage** request to obtain a full description of one or more coverages available. The server responds to such a request with an XML document describing one or more coverages served by that WCPS.

The WCPS **DescribeCoverage** operation is identical to the WCS DescribeCoverage operation [4].

Each server **shall** implement the **DescribeCoverage** operation.

NOTE In a future version the DescribeCoverage response most likely will be extended so as to contain further processing relevant information about the coverage on hand; in particular, concise summarizability information will be provided (which currently implicitly is contained in the interpolation methods).

10 ProcessCoverage operation

10.1 Introduction

A Web Coverage Processing Server evaluates a **ProcessCoverage** request and returns an appropriate response to the client.

Each server **shall** implement the **ProcessCoverage** operation.

While the WCS **GetCoverage** operation allows retrieval of a coverage from a coverage offering, possibly modified through operations like spatial, temporal, and band subsetting and coordinate transformation, the WCPS **ProcessCoverage** extends this functionality through more powerful processing capabilities. This includes, on the one hand, further coverage processing primitives and, on the other hand, nesting of function application, thereby allowing for arbitrarily complex requests.

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.

Clients **can** choose whether to phrase **ProcessCoverage** requests based on a coverage's cell coordinates (i.e., using its ImageCRS) or through spatio-temporal coordinates (i.e., using some other CRS listed in the coverage's **DescribeCoverage** description).

A WCPS response is an ordered sequence of data items. A data item can be a coverage or the result of any other processing expression. The **ProcessCoverage** operation returns a coverage as stored on the server, or a constituent thereof, or a derived coverage, or a constituent thereof.

NOTE 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.

NOTE As the functionality of WCPS centers around coverage processing, metadata are considered only to the extent necessary for a coherent service. This way WCPS keeps orthogonal to other OGC standards.

10.2 WCPS expression language specification

The WCPS primitives plus the nesting capabilities form an expression language; this abstract language collectively is referred to as **WCPS language**. In the following subsections the language elements are detailed. The complete syntax is listed in Appendix A.

A WCPS expression is called **admissible** if and only if it adheres to the WCPS language syntax. WCPS servers **shall** return an exception in response to a WCPS request that is not admissible.

Example The coverage expression

```
C * 2
```

is admissible as it adheres to WCPS syntax whereas

```
C C
```

seen as 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 7.3.

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 with identifier `C` that has a numeric field named `red`.

```
C.red * 2.5
```

10.3 ProcessCoverage abstract request syntax

10.3.1 Overview

A WCPS expression is a **coverageListExpr** (which evaluates to a list of encoded coverages; see Subclause 10.3.2). Each WCPS request shall contain exactly one **coverageListExpr**.

10.3.2 coverageListExpr

The **coverageListExpr** 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 **ProcessCoverage** response unless no exception was generated.

The elements in the **coverageList** clause are taken from the coverage identifiers advertised by the server in its **GetCapabilities** response document. The **coverageList** elements **shall** be inspected sequentially in the order given.

Coverage identifiers **may** occur more than once in a **coverageList**. In this case the coverage **shall** be inspected each time it is listed, respecting the overall inspection sequence.

Let

$v_1, \ldots v_n$ be n **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 **responseList** R,
where

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

R is constructed as follows:

Let R be the empty sequence;
while L_1 is not empty:
{ assign the first element in L_1 to v_1;
while L_2 is not empty:
{ assign the first element in L_2 to v_2;
…
while L_n is not empty:
{ assign the first element in L_n to v_n;
evaluate P, substituting any occurrence
of coverage identifier v_i by the coverage
this identifier refers to;
append the evaluation result to R;
remove the first element from L_n;
}
…
}
remove the first element from L_2;
}
remove the first element from L_1;
}

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 tiff( c )
```

to produce a result list containing three TIFF-encoded coverages

```
( tiff(A), tiff(B), tiff(C) )
```

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

```
for s in ( A, B, C ),
    m in ( M )
return tiff( s * m )
```

10.3.3 processingExpr

The **processingExpr** element is either a **encodedCoverageExpr** (which evaluates to an encoded coverage; see Subclause 10.3.5), or a **storeCoverageExpr** (see Subclause 10.3.4), or a **scalarExpr** (which evaluates to coverage description data or coverage summary data; see Subclause 10.3.7).

10.3.4 storeCoverageExpr

The **storeCoverageExpr** element specifies that an encoded coverage result as described by its *E* sub element is not to be delivered immediately as response to the request, but to be stored on server side for subsequent retrieval. The result of the **storeCoverageExpr** expression is the URL under which the result is provided by the server, and the server returns only the XML response part with the URL(s) being in place of the coverage(s) generated.

Let

E be an **encodedCoverageExpr**.

Then,

for any **URI** *U*
where
U = **store (*E*)**

U is defined as that URI at which the coverage result is made available by the server.

Example The following expression will deliver the URL under which the server stores the TIFF-encoded result coverage C:

```
store( encode( C, "TIFF" ) )
```

NOTE It is not specified in this standard for how long server-side stored coverages remain available; usually they will be deleted after some implementation dependent time to free server space. Future versions of this standard may offer means to address this.

10.3.5 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 **should**, to the largest extent possible, materialise the coverage's metadata. A service **may** store further information as part of the encoding.

Example For a georeferenced coverage, a GeoTIFF result file **should** contain the coverage's geo coordinate and resolution information.

NOTE: For materialization of the coverage grid cell values the coverage's image CRS **shall** be used by default. See **crsTransformExpr** (Subclause 10.3.27) for controlling coverage grid cell values via other CRSs.

Let

C be a **coverageExpr**,
f be a string,
where
f is the name of a data format listed under **supportedFormats** in the **GetCapabilities** response,
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 Some format encodings may lead to a loss of information.

NOTE The extraParams are data format and implementation dependent.

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

```
encode( C, "hdf-eos" )
```

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

Usage of formats **shall** adhere to the regulations set forth in WCS [4] Subclause 9.3.2.2.

10.3.6 booleanExpr

The **booleanExpr** element is a **scalarExpr** (see Subclause 10.3.7) whose result type is Boolean.

NOTE WCPS implementors may extend this to allow, e.g., the usual boolean, arithmetic, and further scalar functions.

10.3.7 scalarExpr

The **scalarExpr** element is either a **getMetaDataExpr** (see Subclause 10.3.8) or a **condenseExpr** (see Subclause10.3.30).

NOTE As such, it returns a result which is not a coverage.

10.3.8 getMetaDataExpr

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

NOTE The cell value sets can be extracted from a coverage using subsetting operations (see Subclause 10.3.21).

Let

C be a **coverageExpr**.

Then,

The following metadata extraction functions are defined, whereby the result is specified in terms of the coverage's probing functions (Table 5):

Metadata function (for some coverage *C*, axis *a*, range field *r*)	**Result** (in terms of probing functions)	**Result type**
identifier(*C***)**	identifier(*C*)	Name
imageCrs(*C*)	imageCRS(*C*)	URN
imageCrsDomain(*C*,*a***)**	imageCrsDomain(*C*,*a*)	(lower bound, upper bound) integer pair
crsSet(*C***)**	crsSet(*C*,*a*)	Set of URNs
generalDomain(*C*,*a*,*c***)**	generalDomain(*C*,*a*,*c*)	(lower bound, upper bound) numeric / string pair
nullDefault(*C***)**	nullDefault(*C*)	value, structured according to rangeType(*C*)
nullSet(*C***)**	nullSet(*C*)	List of values, each structured according to rangeType(*C*)
interpolationDefault(*C*,*r***)**	interpolationDefault(*C*)	Pair of enumeration values
interpolationSet(*C*,*r***)**	interpolationSet(*C*,*a*)	List of pairs of enumeration values

NOTE Not all information about a coverage can be retrieved this way. Adding the information supplied in a **GetCapabilities** and **DescribeCoverage** response provides complete information about a coverage.

Example For some stored coverage C, the following expression evaluates to "C":

```
identifier( C )
```

10.3.9 setMetaDataExpr

The **setMetaDataExpr** element allows to derive a coverage with modified metadata, leaving untouched the coverage cell values and all metadata not addressed.

NOTE As WCPS focuses on the processing of the coverage range values, advanced capabilities for manipulating a coverage's metadata are currently not foreseen.

Let

c_1 be a **coverageExpr**,
m, n, p be integers with $m \geq 0$ and $n \geq 0$ and $p \geq 0$,
$null$ be a **rangeValue** with $null \in$ nullSet(c_1),
$null_1$, …, $null_m$ be **rangeValue**s which are cast-compatible with type rangeType(c_1),
f be an **identifier**, it an **interpolationType**, nr a **nullResistance** with $f \in$ rangeFieldNames(c_1) and $(im,nr) \in$ interpolationSet(c_1,f),
it_1, ..., it_n be **interpolationType**s, and nr_1, ..., nr_n be **nullResistance**s with $f_i \in$ rangeFieldNames(c_1) for $1 \leq i \leq n$ and $im_i \in$ interpolationSet(c_1,f_i),
crs_1, …, crs_p be **crsName**s.

Then,

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

$C_{nullDef}$ = **setNullDefault(** c_1, $null$ **)**
C_{null} = **setNullSet(** c_1, **{** $null_1$, …, $null_m$ **}** **)**
C_{intDef} = **setInterpolationDefault(** c_1, f, **(**im**,**nr**)** **)**
C_{int} = **setInterpolationSet(** c_1, f, **{** **(**im_1**,**nr_1**)**,…,**(**im_n**,**nr_n**)** **}** **)**
C_{crs} = **setCrsSet(** c_1, **{** crs_1,…,crs_p **},** a **)**

c_2 is defined as follows:

Coverage constituent	Changed?
identifier(c_2) = "" (empty string)	**X**
for all $p \in$ imageCrsDomain(c_2): value(c_2, p) = value(c_1, p)	
imageCrs(c_2) = imageCrs(c_1)	
imageCrsDomain(c_2) = imageCrsDomain(c_1)	
axisSet(c_2) = axisSet(c_1)	
for all $a \in$ axisSet(c_2): crsSet($C_{nullDef}$, a) = crsSet(c_1, a) crsSet(C_{null}, a) = crsSet(c_1, a) crsSet(C_{intDef}, a) = crsSet(c_1, a) crsSet(C_{int}, a) = crsSet(c_1, a) crsSet(C_{crs}, a) = { crs_1,…,crs_p } axisType(c_2, a) = axisType(c_1, a)	**X**
for all $a \in$ axisSet(c_2), $c \in$ crsSet(c_2, a): generalDomain(c_2, a, c) = generalDomain(c_2, a, c)	**X**

for all fields $r \in$ rangeFieldNames(C_2): rangeFieldType(C_2, r) = rangeFieldType(C_1, r)	
nullDefault($C_{nullDef}$) = n nullDefault(C_{null}) = if nullDefault(C_1)$\in$ {$null_1$,..., $null_m$} then nullDefault(C_1) else undefined[2] nullDefault(C_{intDef}) = nullDefault(C_1) nullDefault(C_{int}) = nullDefault(C_1) nullDefault(C_{crs}) = nullDefault(C_1)	**X**
nullSet($C_{nullDef}$) = nullSet(C_1) nullSet(C_{null}) = { $null_1$, ... , $null_m$ } nullSet(C_{intDef}) = nullSet(C_1) nullSet(C_{int}) = nullSet(C_1) nullSet(C_{crs}) = nullSet(C_1)	**X**
for all $r \in$ rangeFieldNames(C_2): interpolationDefault($C_{nullDef}$, r) = interpolationDefault(C_1, r) interpolationDefault(C_{null} , r) = interpolationDefault(C_1, r) interpolationDefault(C_{intDef} , r) = (it,nr) interpolationDefault(C_{int} , r) = if interpolationDefault(C_1)$\in$ {(im_1, nr_1),...,(im_n, nr_n)} then interpolationDefault(C_1, r) else undefined[3] interpolationDefault(C_{crs} , r) = interpolationDefault(C_1, r)	**X**
for all $r \in$ rangeFieldNames(C_2): interpolationSet($C_{nullDef}$, r) = interpolationSet(C_1, r) interpolationSet(C_{null} , r) = interpolationSet(C_1, r) interpolationSet(C_{intDef} , r) = interpolationSet(C_1, r) interpolationSet(C_{int} , r) = interpolationSet(C_1, r) interpolationSet(C_{crs} , r) = if r=f then { (im_1, nr_1), ..., (im_n, nr_n) } else interpolationSet(C_1, r) interpolationSet(C_{crs} , r) = interpolationSet(C_1, r)	**X**

Example Assuming that coverage C has a single numeric range field, the following expression evaluates to a coverage that has -100 as its default null value:

```
setNullDefault( C, -100 )
```

[2] For an undefined null default, 0 **shall** be used for numeric null and false for Boolean null (see Clause 7).

[3] An undefined default interpolation method **shall** lead to a runtime exception whenever it needs to be applied (see Clause 7).

Example The following coverage expression evaluates to a coverage that, in its data, resembles *C* , but has no interpolation method available on its range field *landUse*, allows *linear* interpolation with full null resistance, and *quadratic* interpolation with half null resistance on *C*'s range field *panchromatic*:

```
setInterpolation( setInterpolation( C, landUse, { } ),
panchromatic, { linear:full, quadratic:half } )
```

The setNullDefault() and setNullSet() operations **shall** not change any preexisting value in the coverage in an attempt to adapt old null values to the new ones.

NOTE Obviously changing a coverage's null values can render its contents inconsistent.

A server may respond with an exception if it does not support a CRS specified in a setCrsSet() call.

10.3.10 coverageExpr

The **coverageExpr** element is either a **coverageIdentifier** (see Subclause 10.3.11), or **setMetaDataExpr** (see Subclause 10.3.9), or an **inducedExpr** (see Subclause 10.3.12), or a **subsetExpr** (see Subclause 10.3.21), or a **crsTransformExpr** (see Subclause 10.3.27), or a **scaleExpr** (see Subclause 10.3.26), or a **coverageConstructorExpr** (see Subclause 10.3.28), or a **coverageConstructorExpr** (see Subclause 10.3.27), or a **condenseExpr** (see Subclause 10.3.30).

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

10.3.11 coverageIdentifier

The **coverageIdentifier** element represents the name of a single coverage offered by the server addressed.

Let

id be the **identifier** of 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) = id	
for all $p \in$ imageCrsDomain(C_2): value(C_2,p) = value(C_1,p)	
imageCrs(C') = imageCrs(C_1)	

imageCrsDomain(c_2) = imageCrsDomain(c_1)	
axisSet(c_2) = axisSet(c_1)	
for all $a \in$ axisSet(c_2): crsSet(c_2, a) = crsSet(c_1, a) axisType(c_2, a) = axisType(c_1, a)	
for all $a \in$ axisSet(c_2), $c \in$ crsSet(c_2, a): generalDomain(c_2, a, c) = generalDomain(c_2, a, c)	
for all fields $r \in$ rangeFieldNames(C_2): rangeFieldType(c_2, r) = rangeFieldType(c_1, r)	
nullDefault(c_2) = nullDefault(c_1)	
nullSet(c_2) = nullSet(c_1)	
for all $r \in$ rangeFieldNames(c_2): interpolationDefault(c_2, r) = interpolationDefault(c_1, r)	
for all $r \in$ rangeFieldNames(c_2): interpolationSet(c_2, r) = interpolationSet(c_1, r)	

Example The following coverage expression evaluates to the complete, unchanged coverage C, assuming it is offered by the server:

```
C
```

10.3.12 inducedExpr

The inducedExpr element is either a **unaryInducedExpr** (see Subclause 10.3.13) or a **binaryInducedExpr** (see Subclause 10.3.20) or a **rangeConstructorExpr** (see Subclause 10.3.21).

Induced operations allow to simultaneously apply a function originally working on a single cell value to all cells of a coverage. In case the range type contains more than one component, the function is applied to each cell component simultaneously.

The result coverage has the same domain, but **may** change its base type.

NOTE The idea is that for each operation available on the range type, a corresponding coverage operation is provided ("induced from the range type operation"), a concept first introduced by Ritter et al. [8].

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

10.3.13 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.

A **unaryInducedExpr** is either a **unaryArithmeticExpr** or **exponentialExpr** or **trigonometricExpr** (in which case it evaluates to a coverage with a numeric range type; see Subclauses 10.3.14, 10.3.15,10.3.16), a **boolExpr** (in which case it evaluates to a Boolean expression; see Subclause 10.3.17), a **castExpr** (in which case it evaluates to a coverage with unchanged values, but another range type; see Subclause 10.3.18), or a **fieldExpr** (in which case a range field selection is performed; see Subclause 10.3.19).

10.3.14 unaryArithmeticExpr

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

Let

c_1 be a **coverageExpr**
where
for all range fields $r \in$ rangeFieldNames(c_1): r is numeric.

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**
for all p ∈ imageCrsDomain(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**
imageCrs(c_2) = imageCrs(c_1)	
imageCrsDomain(c_2) = imageCrsDomain(c_1)	
axisSet(c_2) = axisSet(c_1)	
for all $a \in$ axisSet(c_2): crsSet(c_2, a) = crsSet(c_1, a) axisType(c_2, a) = axisType(c_1, a)	

for all $a \in$ axisSet(C_2), $c \in$ crsSet(C_2, a): generalDomain(C_2, a, c) = generalDomain(C_2, a, c)	
for all fields $r \in$ rangeFieldNames(C_2): rangeFieldType(C_{plus},r) = rangeFieldType(C_1,r) rangeFieldType(C_{minus},r) = rangeFieldType(C_1,r) rangeFieldType(C_{sqrt},r) = if rangeFieldType(C_1,r)$\in$ {*complex,complex2*} then *complex2* else *double* rangeFieldType(C_{abs},r) = rangeFieldType(C_1,r)	**X**
nullDefault(C_{plus}) = nullDefault(C_1) nullDefault(C_{minus}) = - nullDefault(C_1)) nullDefault(C_{sqrt}) = sqrt(nullDefault(C_1)) nullDefault(C_{abs}) = abs(nullDefault(C_1))	**X**
nullSet(C_{plus}) = nullSet(C_1) nullSet(C_{minus}) = - nullSet(C_1) nullSet(C_{sqrt}) = sqrt(nullSet(C_1)) nullSet(C_{abs}) = abs(nullSet(C_1))	**X**
for all $r \in$ rangeFieldNames(C_2): interpolationDefault(C_2, r) = interpolationDefault(C_1, r)	
for all $r \in$ rangeFieldNames(C_2): interpolationSet(C_2, r) = interpolationSet(C_1, r)	

The server **shall** respond with an exception if one of the coverage's grid cell values or its null values is negative.

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

```
sqrt( C + D )
```

10.3.15 trigonometricExpr

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

Let

C_1 be a **coverageExpr**
where
for all fields $r \in$ rangeFieldNames(C_1): r is numeric.

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**
for all p ∈ imageCrsDomain(C_1): 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**
imageCrs(C_2) = imageCrs(C_1)	
imageCrsDomain(C_2) = imageCrsDomain(C_1)	
axisSet(C_2) = axisSet(C_1)	
for all a ∈ axisSet(C_2): crsSet(C_2, a) = crsSet(C_1, a) axisType(C_2, a) = axisType(C_1, a)	
for all a ∈ axisSet(C_2), c ∈ crsSet(C_2, a): generalDomain(C_2, a, c) = generalDomain(C_2, a, c)	
for all fields r ∈ rangeFieldNames(C_2): rangeFieldType(C_2,r) = if rangeFieldType(C_1,r)∈ {*complex,complex2*} then *complex2* else *double*	**X**
nullDefault(C_{sin}) = sin(nullDefault(C_1)) nullDefault(C_{cos}) = cos(nullDefault(C_1)) nullDefault(C_{tan}) = tan(nullDefault(C_1)) nullDefault(C_{sinh}) = sinh(nullDefault(C_1)) nullDefault(C_{cosh}) = cosh(nullDefault(C_1)) nullDefault(C_{arcsin}) = arcsin(nullDefault(C_1)) nullDefault(C_{arccos}) = arccos(nullDefault(C_1))	**X**

nullDefault(c_{arctan}) = arctan(nullDefault(c_1))	
nullSet(c_{sin}) = sin(nullSet(c_1)) nullSet(c_{cos}) = cos(nullSet(c_1)) nullSet(c_{tan}) = tan(nullSet(c_1)) nullSet(c_{sinh}) = sinh(nullSet(c_1)) nullSet(c_{cosh}) = cosh(nullSet(c_1)) nullSet(c_{arcsin}) = arcsin(nullSet(c_1)) nullSet(c_{arccos}) = arccos(nullSet(c_1)) nullSet(c_{arctan}) = arctan(nullSet(c_1))	**X**
for all $r \in$ rangeFieldNames(c_2): interpolationDefault(c_2, r) = interpolationDefault(c_1, r)	
for all $r \in$ rangeFieldNames(c_2): interpolationSet(c_2, r) = interpolationSet(c_1, r)	

The server **shall** respond with an exception if one of the coverage's grid cell values or its null values is not within the domain of the function to be applied to it.

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

```
sin( C )
```

10.3.16 exponentialExpr

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

Let

c_1 be a **coverageExpr**
where
for all fields $r \in$ rangeFieldNames(C_1): r is numeric.

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_2 is defined as follows:

Coverage constituent	Changed?
identifier(c_2) = "" (empty string)	**X**
for all p $\in$ imageCrsDomain(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))	
imageCrs(C_2) = imageCrs(C_1)	
imageCrsDomain(C_2) = imageCrsDomain(C_1)	
axisSet(C_2) = axisSet(C_1)	
for all $a \in$ axisSet(C_2): crsSet(C_2, a) = crsSet(C_1, a) axisType(C_2, a) = axisType(C_1, a)	
for all $a \in$ axisSet(C_2), $c \in$ crsSet(C_2, a): generalDomain(C_2, a, c) = generalDomain(C_2, a, c)	
for all fields $r \in$ rangeFieldNames(C_2): rangeFieldType(C_2,r) = if rangeFieldType(C_1,r)$\in$ {*complex,complex2*} then *complex2* else *double*	**X**
nullDefault(C_{exp}) = exp(nullDefault(C_1)) nullDefault(C_{log}) = log(nullDefault(C_1)) nullDefault(C_{ln}) = ln(nullDefault(C_1))	**X**
nullSet(C_{exp}) = exp(nullSet(C_1)) nullSet(C_{log}) = log(nullSet(C_1)) nullSet(C_{ln}) = ln(nullSet(C_1))	**X**
for all $r \in$ rangeFieldNames(C_2): interpolationDefault(C_2, r) = interpolationDefault(C_1, r)	
for all $r \in$ rangeFieldNames(C_2): interpolationSet(C_2, r) = interpolationSet(C_1, r)	

The server **shall** respond with an exception if one of the coverage's grid cell values or its null values is not within the domain of the function to be applied to it.

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

```
ln( C )
```

10.3.17 boolExpr

The **boolExpr** element specifies a unary induced Boolean operation. The only operation available is logical negation (logical "not").

Let

c_1 be a **coverageExpr**
where
for all fields $r \in$ rangeFieldNames(C_1): r = *Boolean*.

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**
for all p ∈ imageCrsDomain(c_2): value(c_{not} , p) = not(value(c_1,p)) value(c_{bit}, p) = (value(c_1,p) >> value(n)) mod 2	**X**
imageCrs(c_2) = imageCrs(c_1)	
imageCrsDomain(c_2) = imageCrsDomain(c_1)	
axisSet(c_2) = axisSet(c_1)	
for all $a \in$ axisSet(c_2): crsSet(c_2, a) = crsSet(c_1, a) axisType(c_2, a) = axisType(c_1, a)	
for all $a \in$ axisSet(c_2), $c \in$ crsSet(c_2, a): generalDomain(c_2, a, c) = generalDomain(c_2, a, c)	
for all fields $r \in$ rangeFieldNames(c_2): rangeFieldType(c_2,r) = *Boolean*	**X**
nullDefault(c_{not}) = not(nullDefault(c_1)) nullDefault(c_{bit}) = (nullDefault(c_1) >> value(n)) mod 2	**X**
nullSet(c_{not}) = not(nullSet(c_1)) nullSet(c_{bit}) = (nullSet(c_1) >> value(n)) mod 2	**X**
for all $r \in$ rangeFieldNames(c_2): interpolationDefault(c_2, r) = interpolationDefault(c_1, r)	
for all $r \in$ rangeFieldNames(c_2): interpolationSet(c_2, r) = interpolationSet(c_1, r)	

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.

10.3.18 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.

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

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**
for all p ∈ imageCrsDomain(c_2): value(c_2 , p) = (t) value(c_1,p)	**X**
imageCrs(c_2) = imageCrs(c_1)	
imageCrsDomain(c_2) = imageCrsDomain(c_1)	
axisSet(c_2) = axisSet(c_1)	
for all a ∈ axisSet(c_2): crsSet(c_2, a) = crsSet(c_1, a) axisType(c_2, a) = axisType(c_1, a)	
for all a ∈ axisSet(c_2), c ∈ crsSet(c_2, a): generalDomain(c_2, a, c) = generalDomain(c_2, a, c)	
for all fields r ∈ rangeFieldNames(c_2): rangeFieldType(C_2,r) = t	**X**

nullDefault(C_2) = (t) nullDefault(C_1)	**X**
nullSet(C_2) = { n | $n_0 \in$ nullSet(C_1), n = (t) n_0 }	**X**
for all $r \in$ rangeFieldNames(C_2): interpolationDefault(C_2, r) = interpolationDefault(C_1, r)	
for all $r \in$ rangeFieldNames(C_2): interpolationSet(C_2, r) = interpolationSet(C_1, r)	

The server **shall** respond with an exception if one of the coverage's grid cell values or its null values cannot be cast to the type specified (see Subclause 10.4.5).

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

```
(char) ( C / 2 )
```

10.3.19 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.

Let

C_1 be a **coverageExpr**,
comp be a **fieldName** which is a range field of type t within rangeType(C_1).

Then,

for any **coverageExpr** C_2
where
$C_2 = C_1.comp$

C_2 is defined as follows:

Coverage constituent	Changed?
identifier(C_2) = "" (empty string)	**X**
for all $p \in$ imageCrsDomain(C_2): value(C_2, p) = value($C_1.comp$, p)	
imageCrs(C_2) = imageCrs(C_1)	
ImageCrsDomain(C_2) = imageCrsDomain(C_1)	
axisSet(C_2) = axisSet(C_1)	
for all $a \in$ axisSet(C_2):	

crsSet(C_2, a) = crsSet(C_1, a) axisType(C_2, a) = axisType(C_1, a)	
for all $a \in$ axisSet(C_2), $c \in$ crsSet(C_2, a): generalDomain(C_2, a, c) = generalDomain(C_2, a, c)	
rangeFieldType(C_2,$comp$) = t	**X**
nullDefault(C_2) = nullDefault(C_1.$comp$)	
nullSet(C_2) = { n | $n_0 \in$ nullSet(C_1), n = n_0.$comp$ }	
interpolationDefault(C_2, $comp$) = interpolationDefault(C_1, $comp$)	
interpolationSet(C_2, $comp$) = interpolationSet(C_1, $comp$)	

Example Let C be a coverage with range type integer. Then the following request snippet describes a single-field, integer-type coverage where each cell value contains the difference between red and green band:

```
C.red - C.green
```

10.3.20 binaryInducedExpr

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

The coverage range types **shall** be numeric.

Let

C_1, C_2 be **coverageExpr**s,
S_1, S_2 be **rangeValue**s,
where
imageCrsDomain(C_1,a) = imageCrsDomain(C_2,a),
imageCrs(C_1,a) = imageCrs(C_2,a),
generalDomain(C_1,a) = generalDomain(C_2,a),
crsSet(C_1,a) = crsSet(C_2,a) for all $a \in$ axisSet(C_2),
crsSet(C_1) = crs(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),
null(C_1) = null(C_2),
S_1, S_2 are of type rangeType(C_1).

Then,

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

C_{plusCC} = C_1 **+** C_2 and rangeType(C_1), rangeType(C_2) numeric

C_{minCC} = C_1 **-** C_2 and rangeType(C_1), rangeType(C_2) numeric

C_{multCC} = C_1 ***** C_2 and rangeType(C_1), rangeType(C_2) numeric

C_{divCC} = C_1 **/** C_2 and rangeType(C_1), rangeType(C_2) numeric

C_{andCC} = C_1 **and** C_2 and rangeType(C_1)=rangeType(C_2)=Boolean

C_{orCC} = C_1 **or** C_2 and rangeType(C_1)=rangeType(C_2)=Boolean

C_{xorCC} = C_1 **xor** C_2 and rangeType(C_1)=rangeType(C_2)=Boolean

C_{eqCC} = C_1 **=** C_2 and rangeType(C_1), rangeType(C_2) numeric or Boolean

C_{ltCC} = C_1 **<** C_2 and rangeType(C_1), rangeType(C_2) numeric or Boolean

C_{gtCC} = C_1 **>** C_2 and rangeType(C_1), rangeType(C_2) numeric or Boolean

C_{leCC} = C_1 **<=** C_2 and rangeType(C_1), rangeType(C_2) numeric or Boolean

C_{geCC} = C_1 **>=** C_2 and rangeType(C_1), rangeType(C_2) numeric or Boolean

C_{neCC} = C_1 **!=** C_2 and rangeType(C_1), rangeType(C_2) numeric or Boolean

C_{ovlCC} = C_1 **overlay** C_2 and rangeType(C_1), rangeType(C_2) numeric or Boolean

C_{plusSC} = S_1 **+** C_2 and S_1, rangeType(C_2) numeric

C_{minSC} = S_1 **-** C_2 and S_1, rangeType(C_2) numeric

C_{multSC} = S_1 ***** C_2 and S_1, rangeType(C_2) numeric

C_{divSC} = S_1 **/** C_2 and S_1, rangeType(C_2) numeric

C_{andSC} = S_1 **and** C_2 and S_1, rangeType(C_2) Boolean

C_{orSC} = S_1 **or** C_2 and S_1, rangeType(C_2) Boolean

C_{xorSC} = S_1 **xor** C_2 and S_1, rangeType(C_2) Boolean

C_{eqSC} = S_1 **=** C_2 and S_1, rangeType(C_2) numeric or Boolean

C_{ltSC} = S_1 **<** C_2 and S_1, rangeType(C_2) numeric or Boolean

C_{gtSC} = S_1 **>** C_2 and S_1, rangeType(C_2) numeric or Boolean

C_{leSC} = S_1 **<=** C_2 and S_1, rangeType(C_2) numeric or Boolean

C_{geSC} = S_1 **>=** C_2 and S_1, rangeType(C_2) numeric or Boolean

C_{neSC} = S_1 **!=** C_2 and S_1, rangeType(C_2) numeric or Boolean

C_{ovlSC} = S_1 **overlay** C_2 and S_1, rangeType(C_2) numeric or Boolean

C_{plusCS} = C_1 **+** S_2 and rangeType(C_1), S_2 numeric

C_{mincS} = C_1 **-** S_2 and rangeType(C_1), S_2 numeric

C_{multCS} = C_1 ***** S_2 and rangeType(C_1), S_2 numeric

C_{divCS} = C_1 **/** S_2 and rangeType(C_1), S_2 numeric

C_{andCS} = C_1 **and** S_2 and rangeType(C_1), S_2 Boolean

C_{orCS} = C_1 **or** S_2 and rangeType(C_1), S_2 Boolean

C_{xorCS} = C_1 **xor** S_2 and rangeType(C_1), S_2 Boolean

C_{eqCS} = C_1 **=** S_2 and rangeType(C_1), S_2 numeric or Boolean

C_{ltCS} = C_1 **<** S_2 and rangeType(C_1), S_2 numeric or Boolean

C_{gtCS} = C_1 **>** S_2 and rangeType(C_1), S_2 numeric or Boolean

C_{leCS} = C_1 **<=** S_2 and rangeType(C_1), S_2 numeric or Boolean

C_{geCS} = C_1 **>=** S_2 and rangeType(C_1), S_2 numeric or Boolean

C_{neCS} = C_1 **!=** S_2 and rangeType(C_1), S_2 numeric or Boolean
C_{ovlCS} = C_1 **overlay** S_2 and rangeType(C_1), S_2 numeric or Boolean

C_2 is defined as follows:

Coverage constituent	Changed?
identifier(C_2) = "" (empty string)	**X**
for all p ∈ imageCrsDomain(C_3): value(C_{plusCC}, p) = value(C_1) + value(C_2) value(C_{minCC}, p) = value(C_1) - value(C_2) value(C_{multCC}, p)= value(C_1) * value(C_2) value(C_{divCC}, p) = value(C_1) / value(C_2) value(C_{andCC}, p) = value(C_1) and value(C_2) value(C_{orCC}, p) = value(C_1) or value(C_2) value(C_{xorCC}, p) = value(C_1) xor value(C_2) value(C_{eqCC}, p) = value(C_1) = value(C_2) value(C_{ltCC}, p) = value(C_1) < value(C_2) value(C_{gtCC}, p) = value(C_1) > value(C_2) value(C_{leCC}, p) = value(C_1) <= value(C_2) value(C_{geCC}, p) = value(C_1) >= value(C_2) value(C_{neCC}, p) = value(C_1) != value(C_2) value(C_{ovlCC}, p) = value(C_2) if value(C_1)=0 value(C_1) otherwise value(C_{plusSC}, p) = S_1 + value(C_2) value(C_{minSC}, p) = S_1 - value(C_2) value(C_{multSC}, p) = S_1 * value(C_2) value(C_{divSC}, p) = S_1 / value(C_2) value(C_{andSC}, p) = S_1 and value(C_2) value(C_{orSC}, p) = S_1 or value(C_2) value(C_{xorSC}, p) = S_1 xor value(C_2) value(C_{eqSC}, p) = S_1 = value(C_2) value(C_{ltSC}, p) = S_1 < value(C_2) value(C_{gtSC}, p) = S_1 > value(C_2) value(C_{leSC}, p) = S_1 <= value(C_2) value(C_{geSC}, p) = S_1 >= value(C_2) value(C_{neSC}, p) = S_1 != value(C_2) value(C_{ovlSC}, p) = value(C_2) if S_1=0 S_1 otherwise value(C_{plusCS}, p) = value(C_1) + S_2 value(C_{minCS}, p) = value(C_1) - S_2 value(C_{multCS}, p) = value(C_1) * S_2 value(C_{divCS}, p) = value(C_1) / S_2 value(C_{andCS}, p) = value(C_1) and S_2 value(C_{orCS}, p) = value(C_1) or S_2 value(C_{xorCS}, p) = value(C_1) xor S_2 value(C_{eqCS}, p) = value(C_1) = S_2	**X**

value(c_{ltCS}, p) = value(c_1) < s_2 value(c_{gtCS}, p) = value(c_1) > s_2 value(c_{leCS}, p) = value(c_1) <= s_2 value(c_{geCS}, p) = value(c_1) >= s_2 value(c_{neCS}, p) = value(c_1) != s_2 value(c_{ovlCS}, p) = s_2 if value(c_1)=0 value(c_1) otherwise Whenever necessary, appropriate cast operations are performed on the values prior to performing the binary value operation (cf. Subclause 10.4.5).	
imageCrs(c_3) = imageCrs(c_1)	
imageCrsDomain(c_3) = imageCrsDomain(c_1)	
axisSet(c_2) = axisSet(c_1)	
for all $a \in$ axisSet(c_2): crsSet(c_3, a) = crsSet(c_1, a) axisType(c_2, a) = axisType(c_1, a)	
for all $a \in$ axisSet(c_3), $c \in$ crsSet(c_3, a): generalDomain(c_3, a, c) = generalDomain(c_1, a, c)	
for all $r \in$ rangeFieldNames(c_3): rangeFieldType(c_3, r) = type(value(c_3))	**X**
nullDefault(c_{plusCC}) = nullDefault(c_1) + nullDefault(c_2) nullDefault(c_{minCC}) = nullDefault(c_1) - nullDefault(c_2) nullDefault(c_{multCC}) = nullDefault(c_1) * nullDefault(c_2) nullDefault(c_{divCC}) = nullDefault(c_1) / nullDefault(c_2) nullDefault(c_{andCC}) = nullDefault(c_1) and nullDefault(c_2) nullDefault(c_{orCC}) = nullDefault(c_1) or nullDefault(c_2) nullDefault(c_{xorCC}) = nullDefault(c_1) xor nullDefault(c_2) nullDefault(c_{eqCC}) = nullDefault(c_1) = nullDefault(c_2) nullDefault(c_{ltCC}) = nullDefault(c_1) < nullDefault(c_2) nullDefault(c_{gtCC}) = nullDefault(c_1) > nullDefault(c_2) nullDefault(c_{leCC}) = nullDefault(c_1) <= nullDefault(c_2) nullDefault(c_{geCC}) = nullDefault(c_1) >= nullDefault(c_2) nullDefault(c_{neCC}) = nullDefault(c_1) != nullDefault(c_2) nullDefault(c_{ovlCC}) = nullDefault(c_2) if nullDefault(c_1)=0 nullDefault(c_1) otherwise nullDefault(c_{plusSC}) = s_1 + nullDefault(c_2) nullDefault(c_{minSC}) = s_1 - nullDefault(c_2) nullDefault(c_{multSC}) = s_1 * nullDefault(c_2) nullDefault(c_{divSC}) = s_1 / nullDefault(c_2) nullDefault(c_{andSC}) = s_1 and nullDefault(c_2) nullDefault(c_{orSC}) = s_1 or nullDefault(c_2) nullDefault(c_{xorSC}) = s_1 xor nullDefault(c_2) nullDefault(c_{eqSC}) = s_1 = nullDefault(c_2)	**X**

nullDefault(c_{ltSC}) = s_1 < nullDefault(c_2) nullDefault(c_{gtSC}) = s_1 > nullDefault(c_2) nullDefault(c_{leSC}) = s_1 <= nullDefault(c_2) nullDefault(c_{geSC}) = s_1 >= nullDefault(c_2) nullDefault(c_{neSC}) = s_1 != nullDefault(c_2) nullDefault(c_{ovlSC}) = nullDefault(c_2) if s_1=0 s_1 otherwise nullDefault(c_{plusCS}) = nullDefault(c_1) + s_2 nullDefault(c_{minCS}) = nullDefault(c_1) - s_2 nullDefault(c_{multCS}) = nullDefault(c_1) * s_2 nullDefault(c_{divCS}) = nullDefault(c_1) / s_2 nullDefault(c_{andCS}) = nullDefault(c_1) and s_2 nullDefault(c_{orCS}) = nullDefault(c_1) or s_2 nullDefault(c_{xorCS}) = nullDefault(c_1) xor s_2 nullDefault(c_{eqCS}) = nullDefault(c_1) = s_2 nullDefault(c_{ltCS}) = nullDefault(c_1) < s_2 nullDefault(c_{gtCS}) = nullDefault(c_1) > s_2 nullDefault(c_{leCS}) = nullDefault(c_1) <= s_2 nullDefault(c_{geCS}) = nullDefault(c_1) >= s_2 nullDefault(c_{neCS}) = nullDefault(c_1) != s_2 nullDefault(c_{ovlCS}) = s_2 if nullDefault(c_1)=0 nullDefault(c_1) otherwise	
nullSet(c_{plusCC}) = nullSet(c_1) + nullSet(c_2) nullSet(c_{minCC}) = nullSet(c_1) - nullSet(c_2) nullSet(c_{multCC}) = nullSet(c_1) * nullSet(c_2) nullSet(c_{divCC}) = nullSet(c_1) / nullSet(c_2) nullSet(c_{andCC}) = nullSet(c_1) and nullSet(c_2) nullSet(c_{orCC}) = nullSet(c_1) or nullSet(c_2) nullSet(c_{xorCC}) = nullSet(c_1) xor nullSet(c_2) nullSet(c_{eqCC}) = nullSet(c_1) = nullSet(c_2) nullSet(c_{ltCC}) = nullSet(c_1) < nullSet(c_2) nullSet(c_{gtCC}) = nullSet(c_1) > nullSet(c_2) nullSet(c_{leCC}) = nullSet(c_1) <= nullSet(c_2) nullSet(c_{geCC}) = nullSet(c_1) >= nullSet(c_2) nullSet(c_{neCC}) = nullSet(c_1) != nullSet(c_2) nullSet(c_{ovlCC}) = nullSet(c_2) if nullSet(c_1)=0 nullSet(c_1) otherwise nullSet(c_{plusSC}) = s_1 + nullSet(c_2) nullSet(c_{minSC}) = s_1 - nullSet(c_2) nullSet(c_{multSC}) = s_1 * nullSet(c_2) nullSet(c_{divSC}) = s_1 / nullSet(c_2) nullSet(c_{andSC}) = s_1 and nullSet(c_2) nullSet(c_{orSC}) = s_1 or nullSet(c_2) nullSet(c_{xorSC}) = s_1 xor nullSet(c_2) nullSet(c_{eqSC}) = s_1 = nullSet(c_2) nullSet(c_{ltSC}) = s_1 < nullSet(c_2) nullSet(c_{gtSC}) = s_1 > nullSet(c_2)	**X**

nullSet(C_{leSC}) = S_1 <= nullSet(C_2) nullSet(C_{geSC}) = S_1 >= nullSet(C_2) nullSet(C_{neSC}) = S_1 != nullSet(C_2) nullSet(C_{ovlSC}) = nullSet(C_2) if S_1=0 S_1 otherwise nullSet(C_{plusCS}) = nullSet(C_1) + S_2 nullSet(C_{minCS}) = nullSet(C_1) - S_2 nullSet(C_{multCS}) = nullSet(C_1) * S_2 nullSet(C_{divCS}) = nullSet(C_1) / S_2 nullSet(C_{andCS}) = nullSet(C_1) and S_2 nullSet(C_{orCS}) = nullSet(C_1) or S_2 nullSet(C_{xorCS}) = nullSet(C_1) xor S_2 nullSet(C_{eqCS}) = nullSet(C_1) = S_2 nullSet(C_{ltCS}) = nullSet(C_1) < S_2 nullSet(C_{gtCS}) = nullSet(C_1) > S_2 nullSet(C_{leCS}) = nullSet(C_1) <= S_2 nullSet(C_{geCS}) = nullSet(C_1) >= S_2 nullSet(C_{neCS}) = nullSet(C_1) != S_2 nullSet(C_{ovlCS}) = S_2 if nullSet(C_1)=0 nullSet(C_1) otherwise	
for all $r \in$ rangeFieldNames(C_3): interpolationDefault(C_3, r) = if interpolationDefault(C_2, r) = interpolationDefault(C_1, r) then interpolationDefault(C_1, r) else **none**	**X**
for all $r \in$ rangeFieldNames(C_3): interpolationSet(C_3, r) = interpolationSet(C_2, r) $\cap$ interpolationSet(C_1, r)	**X**

Example The following expression describes a coverage composed of the sum of the red, green, and blue fields of coverage C:

```
C.red + C.green + C.blue
```

10.3.21 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.

It is allowed to list an input coverage more than once.

Let

n be an **integer** with $n \geq 1$,
$C_1, \ldots, C_n$ be **coverageExpr**s,

f_1, …, f_n be **fieldName**s
where, for $1 \le i,j \le n$,
$f_i \in$ rangeFieldNames(c_i),
imageCrs(c_i) = imageCrs(c_j),
imageCrsDomain(c_i) = imageCrsDomain(c_j),
crsSet(c_i) = crsSet(c_j),
generalDomain(c_i,a_i,c_i) = generalDomain(c_j,a_j,c_j)
for all $a_i \in$ axisSet(c_i), $a_j \in$ axisSet(c_j), $c_i \in$ crsSet(c_i), $c_j \in$ crsSet(c_j).

Then,

for any **coverageExpr** c'
where
$c' = \{ c_1 , \ldots , c_n \}$

c' is defined as follows:

Coverage constituent	**Changed?**
identifier(c') = "" (empty string)	**X**
for all $p \in$ imageCrsDomain(c'), i in {1, …, n}: value($c'.f_i$, p) = value($c_i.f_i$, p)	**X**
imageCrs(c') = imageCrs(c_1)	
imageCrsDomain(c') = imageCrsDomain(c_1)	
axisSet(c_2) = axisSet(c_1)	
for all $a \in$ axisSet(c_2): crsSet(c', a) = crsSet(c_1, a) axisType(c_2, a) = axisType(c_1, a)	
for all $a \in$ axisSet(c_1), $c \in$ crsSet(c_1): generalDomain(c', a, c) = generalDomain(c_1, a, c)	
for all fields $r \in \{ f_1, \ldots, f_n \}$: rangeFieldType(c_3, r) = rangeFieldType(c_i, f_i)	**X**
nullDefault(c') = { nullDefault(C_1), …, nullDefault(C_n) }	**X**
nullSet(c') = nullSet(c_1) × … × nullSet(c_n)	**X**
for all i in {1, …, n}: interpolationDefault(c', f_i) = interpolationDefault(c_i, f_i)	**X**
for all i in {1, …, n}: interpolationSet(c', f_i) = interpolationSet(c_i, f_i)	**X**

Example: 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:

```
{ (char) L.nir, (char) L.red, (char) L.green }
```

The following expression transforms a greyscale image G containing a single range field `panchromatic` into an RGB-structured image:

```
{ G.panchromatic, G.panchromatic, G.panchromatic }
```

10.3.22 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 10.3.23), slicing (i.e., cutting out a hyperplane from a coverage, Subclause 10.3.25), extending (Subclause 10.3.24), and scaling (Subclause 0) of a coverage expression.

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 general domain to image CRS (grid cell) coordinates does not need to be disclosed by the server, hence any coordinate transformation should be considered a "black box" by the client.

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

NOTE Range subsetting is accomplished via the unary induced **fieldExpr** (cf. Subclause 10.3.19).

10.3.23 trimExpr

The **trimExpr** element extracts a subset from a given coverage expression along the axis indicated, specified by a lower and upper bound for each axis affected. Interval limits can be expressed in the coverage's image CRS or any CRS which the the coverage supports.

Lower as well as upper limits **must** 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.

Let

c_1 be a **coverageExpr**,
n be an **integer** with $0 \leq n$,
$a_1, \ldots, a_n$ be pairwise distinct **axisName**s with $a_i \in \text{axisNameSet}(c_1)$ for $1 \leq i \leq n$,
$crs_1, \ldots, crs_n$ be pairwise distinct **crsName**s with $crs_i \in \text{crsList}(c_1)$ for $1 \leq i \leq n$,
$(lo_1:hi_1),\ldots,(lo_n:hi_n)$ be **axisPoint** pairs with $lo_i \leq hi_i$ for $1 \leq i \leq n$.

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 **}** **)**

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**
for all $p \in$ imageCrsDomain(c_2): value(c_2, p) = value(c_1,p)	
imageCrs(c_2) = imageCrs(c_1)	
axisSet(c_2) = axisSet(c_1)	
for all $a \in$ axisSet(c_2): if $a = a_i$ for some i then imageCrsDomain(c_2 , a) = ($lo_{i,img}$, $hi_{i,img}$) else imageCrsDomain(c_2 , a) = imageCrsDomain(c_1 , a) where ($lo_{i,img}$, $hi_{i,img}$) = (lo_i,hi_i) if no CRS is indicated, and the transform from crs_i into the image CRS if crs_i is indicated.	**X**
for all $a \in$ axisSet(c_2): crsSet(c_2, a) = crsSet(c_1, a) axisType(c_2, a) = axisType(c_1, a)	
for all $a \in$ axisSet(c_2), $c \in$ crsSet(c_2): if $a = a_i$ for some i then generalDomain(c_2 , a, c) = ($lo_{i,c}$, $hi_{i,c}$) else generalDomain(c_2 , a, c) = generalDomain(c_1 , a, c) where ($lo_{i,c}$,$hi_{i,c}$) represent the axis boundaries (lo_i,hi_i) transformed of (lo_i,hi_i) from the c_2 image CRS into CRS c.	**X**
for all $r \in$ rangeFieldNames(c_2): rangeFieldType(c_2, r) = rangeFieldType(c_1, r)	
nullSet(C_2) = nullSet(C_1)	
for all $r \in$ rangeFieldNames(c_2): interpolationDefault(c_2, r) = interpolationDefault(c_1, r)	

for all $r \in$ rangeFieldNames(c_2): interpolationSet(c_2, r) = interpolationSet(c_1, r)	

NOTE It is possible to mix different CRSs in one trim operation, however each axis must be addressed in exactly one CRS (either image CRS or another supported CRS).

Example A trim operation might simultaneously perform x/y trimming expressed in some geographic coordinate CRS, time trimming in a time CRS, and abstract axis trimming in (integer) grid cell coordinates.

10.3.24 extendExpr

The **extendExpr** element extends a coverage to the bounding box indicated. The new cells are filled with the coverage's default null value.

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.

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 $1 \le i \le n$,
$crs_1, \ldots, crs_n$ be pairwise distinct **crsName**s with $crs_i \in$ crsList(c_1) for $1 \le i \le n$,
$(lo_1:hi_1),\ldots,(lo_n:hi_n)$ be **axisPoint** pairs 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**
for all $p \in$ imageCrsDomain(c_2): value(c_2, p) = value(c_1,p) for $p \in$ imageCrsDomain(c_1)	**X**

value(C_2, p) = nullDefault(C_1) otherwise	
imageCrs(C_2) = imageCrs(C_1)	
axisSet(C_2) = axisSet(C_1)	
for all $a \in$ axisSet(C_2): if $a = a_i$ for some i then imageCrsDomain(C_2 , a) = ($lo_{i,img}$, $hi_{i,img}$) else imageCrsDomain(C_2 , a) = imageCrsDomain(C_1 , a) where ($lo_{i,img}$, $hi_{i,img}$) = (lo_i,hi_i) if no CRS is indicated, and the transform of (lo_i,hi_i) from crs_i into the C_2 image CRS if crs_i is indicated.	**X**
for all $a \in$ axisSet(C_2): crsSet(C_2, a) = crsSet(C_1, a) axisType(C_2, a) = axisType(C_1, a)	
for all $a \in$ axisSet(C_2), $c \in$ crsSet(C_2): if $a = a_i$ for some i then generalDomain(C_2 , a, c) = ($lo_{i,c}$, $hi_{i,c}$) else generalDomain(C_2 , a, c) = generalDomain(C_1 , a, c) where ($lo_{i,c}$,$hi_{i,c}$) represent the axis boundaries (lo_i,hi_i) transformed from their image CRS into CRS c.	**X**
for all $r \in$ rangeFieldNames(C_2): rangeFieldType(C_2, r) = rangeFieldType(C_1, r)	
nullDefault(C_2) = nullDefault(C_1)	
nullSet(C_2) = nullSet(C_1)	
for all $r \in$ rangeFieldNames(C_2): interpolationDefault(C_2, r) = interpolationDefault(C_1, r)	
for all $r \in$ rangeFieldNames(C_2): interpolationSet(C_2, r) = interpolationSet(C_1, r)	

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

10.3.25 sliceExpr

The **sliceExpr** element extracts a spatial slice (i.e., a hyperplane) from a given coverage expression along one of its axes, specified by one or more slicing axes and a slicing position thereon. For each slicing axis indicated, the resulting coverage has a dimension reduced by 1; its axes are the axes of the original coverage, in the same se-

quence, with the section axis being removed from the list. CRSs not used by any remaining axis are removed from the coverage's CRS set.

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.

Let

c_1 be a **coverageExpr**,
n be an **integer** with $0 \leq n$,
$a_1, \ldots, a_n$ be pairwise distinct **axisName**s with $a_i \in$ axisNameSet(c_1) for $1 \leq i \leq n$,
$crs_1, \ldots, crs_n$ be pairwise distinct **crsName**s with $crs_i \in$ crsList(c_1) for $1 \leq i \leq n$,
$s_1, \ldots, s_n$ be **axisPoint**s for $1 \leq i \leq n$.

Then,

for any **coverageExpr** c_2
where c_2 is one of
$c_{bracket} = c_1$ `[` s_1`,` $\ldots$`,` s_n `]`
$c_{func} =$ **`slice(`** c_1`,` **`,`** **`{`** s_1`,` $\ldots$`,` s_n **`}`** **`)`**
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**
for all $p \in$ imageCrsDomain(c_1) such that for all $a \in$ axisSet(c_1): if $a \in \{a_1, \ldots, a_n\}$ then let p_a be that component of p addressing axis a $p_a' = s_i$ for $s_{img,i}$ $p_a' = s_i$ transformed from crs_i for $s_{crs,i}$ else let p_a be that component of p addressing axis a let p_a' be that component of p' addressing axis a $p_a, p_a' \in$ imageCrsDomain(c_1,a) value(c_2, p) = value(c_1,p')	

imageCrs(c_2) = imageCrs(c_1)	
axisSet(c_2) = axisSetCrs(c_1) \ {$a_1, \ldots, a_n$}	**X**
for all $a \in$ axisSet(c_2): imageCrsDomain(c_2, a) = imageCrsDomain(c_1, a)	**X**
for all $a \in$ axisSet(c_2): crsSet(c_2, a) = crsSet(c_1, a) \ ({$crs_1, \ldots, crs_n$} \ crsSet(c_1, a)) axisType(c_2, a) = axisType(c_1, a)	**X**
for all $a \in$ axisSet(c_1) \ {$a_1, \ldots, a_n$}, $c \in$ crsSet(c_2, a): generalDomain(c_2, a, c) = generalDomain(c_1, a, c)	**X**
for all $r \in$ rangeFieldNames(c_2): rangeFieldType(c_2, r) = rangeFieldType(c_1, r)	
NullSet(C_2) = nullSet(C_1)	
for all $r \in$ rangeFieldNames(c_2): interpolationDefault(c_2, r) = interpolationDefault(c_1, r)	
for all $r \in$ rangeFieldNames(c_2): interpolationSet(c_2, r) = interpolationSet(c_1, r)	

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

NOTE In a future version of this document this function is likely to be .extended with multi-dimensional slicing.

10.3.26 scaleExpr

The **scaleExpr** element performs scaling along a subset of the source coverage's axes. For each of the coverage's range fields, an interpolation method can be chosen from the coverage's interpolation method list. If no interpolation is indicated for a field, then this field's default interpolation method.

A service exception **shall** be raised if for any of the coverage's range fields no appropriate interpolation method is available for the resampling/interpolation performed in the course of the transformation.

Let

c_1 be a **coverageExpr**,
m, n be **integer**s with $0 \le m$ and $0 \le n$,
$a_1, \ldots, a_m$ be pairwise distinct **axisName**s with $a_i \in$ axisNameSet(c_1) for $1 \le i \le m$,
$(lo_1, hi_1), \ldots, (lo_m, hi_m)$ be **axisPoint** pairs with $lo_i \le hi_i$ for $1 \le i \le m$,
$f_1, \ldots, f_n$ be pairwise distinct **fieldName**s, $it_1, \ldots, it_n$ be **interpolationType**s,

nr_1, ..., nr_n be **nullResistance**s with $f_i \in$ rangeFieldNames(c_1) and $(it_i, nr_i) \in$ interpolationSet(c_1, f_i) for $1 \le i \le n$.

Then,

For any **coverageExpr** c_2,
where

c_2 = **scale** (
c_1,
{ p_1, …, p_m })
{ $f_1(it_1, nr_1)$, …, $f_n(it_n, nr_n)$ }
)

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**
for all $p \in$ imageCrsDomain(c_2): value(c_2, p) is obtained by rescaling the coverage along axes a_i such that the coverage's extent along axis a_i is set to ($lo_i:hi_i$), expressed in the coverage's image CRS; all other axes remain unaffected. For every range field f_i listed, interpolation type it_i and null resistance nr_i are applied during evaluation; for all range fields not listed their resp. default interpolation is applied.	**X**
imageCrs(c_2) = imageCrs(c_1)	
axisSet(c_2) = axisSet(c_1)	
for all $a \in$ axisSet(c_1): if $a = a_i$ for some i then imageCrsDomain(c_2 , a) = (lo_i :hi_i) else imageCrsDomain(c_2 , a) = imageCrsDomain(c_1 , a)	**X**
for all $a \in$ axisSet(c_2): crsSet(c_2, a) = crsSet(c_1, a) axisType(c_2, a) = axisType(c_1, a)	
for all $a \in$ axisSet(c_2), $c \in$ crsSet(c_2, a): generalDomain(c_2,a,c) = generalDomain(c_1,a,c)	
for all $r \in$ rangeFieldNames(c_2):	

rangeFieldType(C_2, r) = rangeFieldType(C_1, r)	
nullSet(C_2) = nullSet(C_1)	
for all $r \in$ rangeFieldNames(C_2): interpolationDefault(C_2, r) = interpolationDefault(C_1, r)	
for all $r \in$ rangeFieldNames(C_2): interpolationSet(C_2, r) = interpolationSet(C_1, r)	

NOTE Scaling regularly involves cell interpolation, hence potential numerical instabilities have to be expected.

Example The following expression performs x/y scaling of some coverage C – which has one single range field, `temperature` – using interpolation type `cubic` and null resistance `full` in both `x` and `y` axis, assuming that the range field supports this method:

```
scale(
    C,
    { x:(lox:hix), y:(loy:hiy)  },
    { red:(cubic,full), nir:(linear,half) }
)
```

If the default interpolation method is undefined and no interpolation method is indicated expressly then the server **shall** respond with a runtime exception.

10.3.27 crsTransformExpr

The element performs reprojection of a coverage. For each axis, a separate CRS can be indicated; for any axis for which no CRS is indicated, no reprojection will be performed. For the resampling which usually is incurred the interpolation method and null resistance can be indicated per range field; for fields not mentioned the default will be applied.

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

NOTE A service may refuse to accept some CRS combinations (e.g., different CRSs handling for x and y axis).

NOTE As any coverage bearing a CRS beyond its image CRS is stored in some CRS, there will normally be a parameter combination which retrieves the coverage as stored, without any reprojection operation required.

Let

C_1 be a **coverageExpr**,
m, n be **integer**s with $1 \le m$ and $0 \le n$,
$a_1, \ldots, a_m$ be pairwise distinct **axisName**s with $a_i \in$ axisNameSet(C_1) for $1 \le i \le m$,
$crs_1, \ldots, crs_m$ be pairwise distinct **crsName**s with $crs_i \in$ crsList(C_1) for $1 \le i \le m$,
f_1, ..., f_n be pairwise distinct **fieldName**s,

it_1, ..., it_n be **interpolationType**s,
nr_1, ..., nr_n be **nullResistance**s with $f_i \in$ rangeFieldNames(C_1)
and $(it_i, nr_i) \in$ interpolationSet(C_1, f_i) for $1 \le i \le n$.

Then,

for any **coverageExpr** C_2
where

C_2 = **crsTransform(**
C_1, ,
{ a_1:crs_1, …, a_m:crs_m })
{ f_1(it_1,nr_1), …, f_n(it_n,nr_n) }
)

C_2 is defined as follows:

Coverage constituent	Changed?
identifier(C_2) = "" (empty string)	**X**
for all $p \in$ imageCrsDomain(C_2): value(C_2, p) is obtained by reprojecting coverage C_1 along axes a_i into CRS crs_i; all other axes remain unaffected. For every range field f_i listed, interpolation type it_i and null resistance nr_i are applied during evaluation; for all range fields not listed their resp. default interpolation is applied.	**X**
imageCrs(C_2) = imageCrs(C_1)	
axisSet(C_2) = axisSet(C_1)	
for all $a \in$ axisSet(C_1): imageCrsDomain(C_2 , a) = imageCrsDomain(C_1 , a)	
for all $a \in$ axisSet(C_2): crsSet(C_2, a) = crsSet(C_1, a) axisType(C_2, a) = axisType(C_1, a)	
for all $a \in$ axisSet(C_2), $c \in$ crsSet(C_2, a): generalDomain(C_2,a,c) = generalDomain(C_1,a,c)	
for all $r \in$ rangeFieldNames(C_2): rangeFieldType(C_2, r) = rangeFieldType(C_1, r)	
nullSet(C_2) = nullSet(C_1)	
for all $r \in$ rangeFieldNames(C_2): interpolationDefault(C_2, r) = interpolationDefault(C_1, r)	
for all $r \in$ rangeFieldNames(C_2):	

interpolationSet(c_2, r) = interpolationSet(c_1, r)	

10.3.28 coverageConstExpr

The **coverageConstExpr** element allows to indicate a coverage constant inline, as part of the expression. Its extent is defined in the header, followed by a list of the cell values. Cell values have to be coherent in number (matching with the extent defined in the header) and in range type, that is: all range field values must be able to be cast into one of the admissible range field data types defined in Table 4.

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 **setMetadataExpr** (Subclause 10.3.9) is available.

NOTE This constructor is useful whenever a small coverage is used in the context of a request.

Let

f be a **fieldName**,
d be an **integer** with $d>0$,
t_i be **axisType**s for $1 \le i \le d$, where only axis type **`abstract`** may occur more than once,
$name_i$ be pairwise distinct **identifier**s for $1 \le i \le d$, which additionally, in the request on hand, are not used already as a variable in this expression's scope,
lo_i and hi_i be **integer**s for $1 \le i \le d$ with $lo_i \le hi_i$,
V be a **scalarExpr** possibly containing occurrences of $name_i$.

Then,

For any **coverageExpr** C
where

```
C = coverage f
    over      t1 name1 in (lo1,hi1),
              …,
              td named in (lod,hid)
    values V
```

C is defined as follows:

Coverage constituent	Changed?
identifier(C) = "" (empty string)	**X**
for all $p \in$ imageCrsDomain(C): value(C, p) = V' where expression V' is obtained from expression V by substituting all occurrences of $name_i$ by v where ($name_i$, v)$\in p$	**X**
imageCrs(C) = c_0	**X**

(i.e., the WCPS standard image CRS, see Clause 7)	
imageCrsDomain(C) is set to a d-dimensional cube with axis names $name_1 \ldots name_d$ where the extent of axis $name_i$ ranges from lo_i to hi_i (including these boundary values).	**X**
axisSet(C_2) = axisSet(C_1)	
for all $a \in$ axisSet(C_2): crsSet(C, a) = {} axisType(C, a) = if $a = name_i$ then t_1	**X**
for all $a \in$ axisSet(C), $c \in$ crsSet(C, a): generalDomain(C, a, c) = undefined[4]	**X**
for $r \in \{ f \}$, rangeFieldType(C,r) = type(V) i.e., the single range field's type is equal to the result type of expression V	**X**
nullSet(C) = {}	**X**
for all $r \in$ rangeFieldNames(C_2): interpolationDefault(C, r) = **none**	**X**
for all $r \in$ rangeFieldNames(C_2): interpolationSet(C, r) = {}	**X**

10.3.29 coverageConstructorExpr

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

The domain definition consists, for each dimension, of a unique axis name plus lower and upper bound of the coverage, expressed in a fixed image CRS and using integer coordinates; for this image CRS one of the identifiers listed in [05-096r1] Table 1 **shall** be used.

The coverage's content is defined by a general expression. The result type of the expression defining the contents also determines the coverage range type.

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 **setMetadataExpr** (Subclause 10.3.9) is available.

NOTE This constructor is useful

[4] Note that, due to the empty crsSet, this "loop" anyway will not be "entered".

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

Let

f be a **fieldName**,
d be an **integer** with $d>0$,
t_i be **axisType**s for $1 \le i \le d$, where only axis type **`abstract`** may occur more than once,
$name_i$ be pairwise distinct **identifier**s for $1 \le i \le d$, which additionally, in the request on hand, are not used already as a variable in this expression's scope,
lo_i and hi_i be **integer**s for $1 \le i \le d$ with $lo_i \le hi_i$,
V be a **scalarExpr** possibly containing occurrences of $name_i$**.**

Then,

For any **coverageExpr** *C*
where

```
C = coverage f
    over      t1 name1 in (lo1,hi1),
              …,
              td named in (lod,hid)
    values V
```

C is defined as follows:

Coverage constituent	Changed?
identifier(*C*) = "" (empty string)	**X**
for all $p \in$ imageCrsDomain(*C*): value(*C*, *p*) = *V'* where expression *V'* is obtained from expression *V* by substituting all occurrences of $name_i$ by *v* where $(name_i, v) \in p$	**X**
imageCrs(*C*) = c_0 (i.e., the WCPS standard image CRS, see Clause 7)	**X**
imageCrsDomain(*C*) is set to a d-dimensional cube with axis names $name_1 \ldots name_d$ where the extent of axis $name_i$ ranges from lo_i to hi_i (including these boundary values).	**X**
axisSet(c_2) = { $name_1$,…, $name_d$ }	**X**
for all $a \in$ axisSet(*C*): crsSet(*C*, *a*) = {} axisType(*C*, *a*) = if *a* = $name_i$ then t_1	**X**

for all *a* ∈ axisSet(*C*), *c* ∈ crsSet(*C*, *a*): generalDomain(*C*, *a*, *c*) = undefined[5]	**X**
for *r* ∈ { *f* }, rangeFieldType(*C*,*r*) = type(*V*) i.e., the single range field's type is equal to the result type of expression *V*	**X**
nullSet(C) = {}	**X**
for all *r* ∈ rangeFieldNames(C_2): interpolationDefault(*C*, *r*) = **none**	**X**
for all *r* ∈ rangeFieldNames(C_2): interpolationSet(*C*, *r*) = {}	**X**

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

```
coverage bucket in [ 0 : 255 ]
values count( C = bucket )
```

10.3.30 condenseExpr

A **condenseExpr** is either a **reduceExpr** (see Subclause □) or a **generalCondenseExpr** (see Subclause 10.3.31). It takes a coverage and summarizes its values using some summarization function. The value returned is scalar.

10.3.31 generalCondenseExpr

The general **generalCondenseExpr** consolidates cell values of a coverage along selected axes 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.

Any summarisation function s() is admissible for a **generalCondenseExpr** over some coverage if it has the following properties:

- s() is a binary function between values of the coverage range type;
- s() is commutative and associative.

Example Binary "+" on floating point numbers is admissible for a condenser on a float coverage, while binary "-" is not.

Let

[5] Note that, due to the empty crsSet, this "loop" anyway will not be "entered".

op be a **condenseOpType**,
d be some **integer** with $d>0$,
$name_i$ be pairwise distinct **identifier**s which additionally, in the request on hand, are not used already as a variable in this expression's scope,
lo_i and hi_i be **integer**s for $1 \le i \le d$ with $lo_i \le hi_i$,
P be a **booleanExpr** possibly containing occurrences of $name_i$,
V be a **scalarExpr** possibly containing occurrences of $name_i$
where
$1 \le i \le d$.

Then,

For any **scalarExpr** *S*
where

```
S = condense op
    over name1 in [lo1:hi1],
         …,
         named in [lod:hid]
    [ where P ]
    using V
```

S is constructed as follows:

```
Let S = neutral element of type(V);
for all name1 ∈ {lo1,… ,hi1}
    for all name2 ∈ {lo2,… ,hi2}
        …
            for all named ∈ {lod,… ,hid}
                let predicate P' be obtained from expression P
                by substituting all occurrences of namei by v
                where (namei, v)∈p;
                if (P')
                then
                    let V' be obtained from expression V
                    by substituting all occurrences of namei by v
                    where (namei, v)∈p;
                    S = S op value(V');
return S
```

Null values encountered **shall** be treated according to the :

- if at least one non-null value is encountered in the repeated evaluation of *V*, then all null values **shall** be ignored;

- if *V* is not evaluated at least once, or if there are only null-valued input values, then the overall result **shall** be null.

Example For a filter kernel *k*, the condenser must summarise not only over the cell under inspection, but also some neighbourhood. The following applies a filter kernel to some coverage *C*:

```
coverage x in imageCrsDomain(C)
  values condense +
         over y in imageCrsDomain(k)
         using C[x+y] * k[y]
```

where k is a 3x3 matrix like

1	3	1
0	0	0
-1	-3	-1

NOTE See **coverageConstExpr** for a way to specify the k matrix.

NOTE 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 0) to phrase high-level imaging, signal processing and statistical operations.

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

- A WCPS implementation may offer further summarisation functions.
- The **condenseExpr** gives explicit access to the coordinate values; this makes summarisation considerably more powerful (see example below).

10.3.32 reduceExpr

A **reduceExpr** element derives a summary value from the coverage passed; in this sense it "reduces" a coverage to a scalar value. A **reduceExpr** is either an `add`, `avg`, `min`, `max`, `count`, `some`, or `all` operation.

Table 6 – reduceExpr definition via generalCondenseExpr (*a* is a numeric, *b* a Boolean coverageExpr)

reduceExpr definition	Meaning

`add(a) =`
`  condense +`
`  over x in sdom(a)`
`  using a[x]`
	sum over all cells in *a*

`avg(a) =`
`  add(a) /`
`  | imageCrsDomain(a) |`
	Average of all cells in *a*

`min(a) =`
`  condense min`
`  over x in sdom(a)`
`  using a[x]`
	Minimum of all cells in *a*

`max(a) =`
`  condense max`
`  over x in sdom(a)`
`  using a[x]`
	Maximum of all cells in *a*

`count(b) =`
`  condense +`
`  over x in sdom(b)`
`  where b[x]`
`  using 1`
	Number of cells in *b*

`some(b) =`
`  condense or`
`  over x in sdom(b)`
`  using b[x]`
	is there any cell in *b* with value true?

`all(b) =`
`  condense and`
`  over x in sdom(b)`
`  using b[x]`
	do all cells of *b* have value true?

10.4 Expression evaluation

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

10.4.1 Evaluation sequence

A Web Coverage Processing Server **shall** evaluate coverage expressions from left to right.

10.4.2 Nesting

A Web Coverage Processing Server **shall** allow to nest 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.

10.4.3 Parentheses

A Web Coverage Processing Server **shall** allow use of parentheses to enforce a particular evaluation sequence.

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 )`

10.4.4 Operator precedence rules

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 is done left to right.

10.4.5 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 7 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 7 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.

Table 7 – Type extension sequence.

Type extension rules
`Boolean > short`
`short > Boolean`
`Boolean > unsigned short`
`unsigned short > Boolean`
`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`

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
```

Extending Boolean to (unsigned) short **shall** map `false` to 0 and `true` to 1.

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_cells( I * B )
```

Before executing any binary operation where the two operands are of different type a cast operation **shall** be attempted to achieve equal types.

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

NOTE The cast operation is the same as in programming languages and database query languages.

10.4.6 Evaluation exceptions

Whenever a coverage expressions cannot be evaluated according to the rules specified in Clauses 10.3 and 10.4, the Web Coverage Processing Server **shall** respond with an exception.

Example The following expression will lead to an exception when used in a ProcessCoverage request (reasons: division by zero; square root of a negative number):

```
C / 0

sqrt( - abs( C ) )
```

10.5 ProcessCoverage encoding

A **ProcessCoverage** request **shall** contain exactly one valid WCPS expression, encoded in one of the structures as described in Subclause 10.5.1.

The server **shall** answer with a response as described in Subclause 10.5.2.

10.5.1 Request encodings

A WCPS server **may** support KVP request encoding, it **shall** support XML request encoding, and it **may** support SOAP request encoding. If SOAP request encoding is supported then the server **shall** also support SOAP response encoding.

10.5.1.1 KVP request encoding

The key-value pair encoding allows clients to use the HTTP GET method for transmitting **ProcessCoverage** requests.

Table 8 specifies the complete **ProcessCoverage** Request.

Table 8 – The ProcessCoverage Request expressed as Key-Value Pairs

URL Component	Description	Multiplicity
http://server_address/path/script?	URL of WCS server.	*Required*
SERVICE=WCPS	Service name. Must be "WCPS".	*Required*
VERSION=*m.n.p*	Request protocol version, *m*, *n*, *p* being non-negative integer numbers.	*Required*
REQUEST=ProcessCoverage	Name of the request. Must be "ProcessCoverage".	*Required*
RESULT=*expr*	The expression describing the result coverage(s) derived from the coverage offering. Must conform with Subclause 10.5.1.1.3.	*Required*
EXCEPTIONS= application/vnd.ogc.se_xml	The format in which exceptions are to be reported by the server. The currently only allowed format is XML. Default: application/vnd.ogc.se_xml	*Optional*
(Vendor-specific parameters)	Default: none	*Optional*

10.5.1.1.1 SERVICE=WCPS / VERSION=*version*

The SERVICE parameters is fixed to the string "WCPS" ; any upper/lower case combination **may** be used. The VERSION parameter shall refer to the WCPS protocol version the server implements.

10.5.1.1.2 REQUEST=ProcessCoverage

The Basic Service Elements clause defines this parameter. For **ProcessCoverage**, the value "ProcessCoverage" **shall** be used; any upper/lower case combination **may** be used.

10.5.1.1.3 RESULT=*expr*

The RESULT argument is a valid WCPS expression, in the abstract syntax as specified in Subclause 10.3 and with all characters which are not allowed in URLs properly escaped. For the URL encoding the pertaining IETF rules [6] **shall** be used.

10.5.1.1.4 EXCEPTIONS

A Web Coverage Processing Service **shall** offer the exception reporting format *application/vnd.ogc.se_xml* by listing it in its **GetCapabilities** XML response. The entire MIME type string in **Capability / Exceptions / Format** is used as the value of the EXCEPTIONS parameter.

Errors are reported using Service Exception XML, as specified in Subclause B.3. This is the default exception format if none is specified in the request.

10.5.1.2 XML request encoding

The XML encoding allows clients to use the HTTP GET or POST method for transmitting **ProcessCoverage** requests. See Annex B for the XML schema.

10.5.1.3 SOAP request encoding

The SOAP encoding allows clients to use the SOAP [] protocol for communication with the server. See Annex E for the SOAP transfer definitions.

10.5.2 Response encodings

The response to a valid **ProcessCoverage** request **shall** consist of one of the following alternatives:

- A coverage, encoded in a particular data format, or a sequence of encoded coverages;
- A record of values, or a sequence of such value records;
- A scalar numeric value, or a sequence of such values.

In an HTTP environment, the returned value **shall** have a Content-type entity header that matches the format of the return value.

10.5.2.1 Response structure

The encoding of a **ProcessCoverage** response consists of an XML structure plus, if URL forwarding has been specified for the result provision using the `store()` function (see Subclause 10.3.4), of one or more data files accessible through the URLs communicated by the server. The XML response type is `ProcessCoverageResponseType`, defined in schema file wcpsProcessCoverage.xsd (see Annex B).

Depending on the response type, the response to a WCPS request **shall** be one of the following:

- For an encoded coverage or a sequence of encoded coverages: a response as specified in WCS [4] Subclause 7.5.
- For an encoded coverage or a sequence of encoded coverages where function `store()` was used in the request: an XML response of type `ProcessCoverageResponseUrlType` containing a sequence of URLs where each URL refers to one response list coverage element, in proper sequence.
- For a scalar numeric value, or a sequence of such values (i.e., non-coverage results): an XML structure of type `ProcessCoverageResponseScalarType`.

10.5.2.2 Exceptions

An invalid **ProcessCoverage** request **shall** yield an error output, either as a WCPS exception reported in the requested Exceptions format (in case of a KVP or XML request), or as a SOAP Fault message (in case of a SOAP request), or as a network protocol error response.

A Web Coverage Processing server throwing an exception **shall** adhere to the value of the Exceptions parameter. Nonetheless, a Web Coverage Processing server **may**, due

to circumstances beyond its control, return nothing (this might result from the HTTP server's behavior caused by a malformed request, by an invalid HTTP request, by access violations, or any of several other conditions). WCPS clients should be prepared for this eventuality.

Annex A
(normative)

WCPS Abstract Syntax

A.1 Overview

The WCPS expression syntax is described below in EBNF grammar syntax according to [6].

Boldface tokens represent literals which appear as is in a valid WCPS expression ("terminal symbols"), tokens in italics represent sub-expressions to be substituted according to the grammar production rules ("non-terminals"). Any number of whitespace characters (blank, tabulator, newline) **may** appear between tokens, including none.

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.

A.2 WCPS syntax

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

coverageList:
        coverageName *( , coverageName )

processingExpr:
        encodedCoverageExpr
      | storeExpr
      | scalarExpr

encodedCoverageExpr:
        encode ( coverageExpr, formatName )
      | encode ( coverageExpr, formatName, extraParams )

formatName:
        string
```

```
extraParams:
        string

storeExpr:
        store ( encodedCoverageExpr )

scalarExpr:
        getMetaDataExpr
      | generalCondenseExpr
      | booleanScalarExpr
      | numericScalarExpr
      | ( scalarExpr )

getMetaDataExpr:
        identifier ( coverageExpr )
      | imageCrs ( coverageExpr )
      | imageCrsDomain ( coverageExpr )
      | crsSet ( coverageExpr )
      | generalDomain( coverageExpr )
      | nullDefault ( coverageExpr )
      | nullSet( coverageExpr )
      | interpolationDefault ( coverageExpr , fieldName
)
      | interpolationSet ( coverageExpr , fieldName )

booleanScalarExpr:
        booleanConstant
      | not booleanScalarExpr
      | booleanScalarExpr and booleanScalarExpr
      | booleanScalarExpr or booleanScalarExpr
      | booleanScalarExpr xor booleanScalarExpr

numericScalarExpr:
        integerConstant
      | floatConstant
      | - numericScalarExpr
      | + numericScalarExpr
      | numericScalarExpr + numericScalarExpr
      | numericScalarExpr - numericScalarExpr
      | numericScalarExpr * numericScalarExpr
      | numericScalarExpr / numericScalarExpr
      | abs ( numericScalarExpr )
      // an implementation may extend this with further numeric operations

coverageExpr:
        coverageName
      | setMetaDataExpr
      | inducedExpr
      | subsetExpr
      | crsTransformExpr
      | scaleExpr
      | coverageConstExpr
      | coverageConstructorExpr
      | ( coverageExpr )

 setMetaDataExpr:
      | setNullDefault ( coverageExpr , rangeValue )
```

```
        | setNullSet ( coverageExpr ,
            { [ rangeValue *( , rangeValue ) ] } )
        | setInterpolationDefault ( coverageExpr ,
 fieldName
            interpolationMethod )
        | setInterpolationSet ( coverageExpr , fieldName ,
            { [ interpolationMethod
                *( , interpolationMethod ) ] } )
        | setCrsSet ( coverageExpr ,
            { [ crsName *( , crsName ) ] } )

inducedExpr:
        | unaryInducedExpr
        | binaryInducedExpr

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

unaryArithmeticExpr:
            + coverageExpr
        | - coverageExpr
        | sqrt ( coverageExpr )
        | abs ( 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 , integerExpr )

castExpr:
            ( cellType ) coverageExpr

cellType:
            bool
        | char
        | unsigned char
        | short
```

```
        | unsigned short
        | long
        | unsigned long
        | float
        | double
        | complex
        | complex2

fieldExpr:
        coverageExpr . fieldName

binaryInducedExpr:
        | coverageExpr binaryInducedOp coverageExpr
        | coverageExpr binaryInducedOp rangeValue
        | rangeValue binaryInducedOp coverageExpr

binaryInducedOp:
        | +
        | -
        | *
        | /
        | and
        | or
        | xor
        | =
        | <
        | >
        | <=
        | >=
        |!=
        | overlay

subsetExpr:
        | trimExpr
        | sliceExpr
        | extendExpr

trimExpr:
        coverageExpr [ axisIntervalList ]
        | trim ( coverageExpr , axisIntervalList )

sliceExpr:
        coverageExpr [ axisPointList ]
        | slice ( coverageExpr , axisPointList )

extendExpr:
        extend ( coverageExpr , axisIntervalList )

scaleExpr:
        scale ( coverageExpr , axisIntervalList ,
                fieldInterpolationList )

crsTransformExpr:
        crsTransform ( coverageExpr , axisIntervalList ,
                       fieldInterpolationList )
```

```
axisIntervalList:
       { axisIntervalElement *( , axisIntervalElement ) }

axisIntervalElement:
         axisName [ : crsName ] ( axisPoint : axisPoint )

axisPointList:
       { axisPointElement *( , axisPointElement ) }

axisPointElement:
         axisName [ : crsName ] ( axisPoint )

axisPoint:
         integerConstant
       | floatConstant
       | stringConstant
         // for time values: cf. ISO 8601:2000 [10], WCS [4] Table 16, 17

axisCrsList:
       { axisCrsElement *( , axisCrsElement ) }

axisCrsElement:
         axisName : crsName

fieldInterpolationList:
         { fieldInterpolationListElement
           *( , fieldInterpolationListElement ) }

fieldInterpolationListElement:
         fieldName : interpolationMethod

interpolationMethod:          // taken from WCS [4]
         ( interpolationType : nullResistance )

interpolationType:            // taken from WCS [4] Table I.7
         nearest
       | linear
       | quadratic
       | cubic

nullResistance:               // taken from WCS [4]
         full
       | none
       | half
       | other

coverageConstructorExpr:
         coverage fieldName
         over variableList
         values scalarExpr

variableList:
         axisType variableName
             in ( integerExpr : integerExpr )
         *( , axisType variableName
             in ( integerExpr : integerExpr ) )
```

```
axisType:
          x
        | y
        | z
        | time
        | abstract

condenseExpr:
          reduceExpr
        | generalCondenseExpr

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

generalCondenseExpr:
          condense condenseOpType
          over variableList
          [ where booleanScalarExpr ]
          using scalarExpr

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

coverageName:
          name

variableName:
          name

crsName:
         name                  // containing a valid CRS name

axisName:
          name

fieldName:
          name                  // as defined in WCS [4] Table 19

rangeValue:
          structuredLiteral
        | atomicLiteral

structuredLiteral:
          { rangeValueList }
        | struct { rangeValueList }
```

```
rangeValueList:
         rangeValue *( , rangeValue )

atomicLiteral:
         booleanLiteral
       | integerLiteral
       | floatLiteral
       | complexLiteral
       | stringLiteral

complexLiteral:
         complex ( floatLiteral , floatLiteral )
```

A identifier **shall** be a consecutive sequence consisting 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, and the first character **shall not** be a decimal digit.

NOTE WCS [4] allows more freedom in the choice of identifiers; for the sake of simplicity this is tightened for now, but may be adapted to the WCS identifier definition in a future version of this standard.

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

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

A *stringLiteral* **shall** represent a character sequence expressed by enclosing it into double quotes ('"').

Annex B
(normative)

WCPS XML Schemas

B.1 GetCapabilities request Schema

See file wcpsCapabilities.xsd

B.2 GetCapabilities response schema

See file wcpsCapabilities.xsd

B.3 DescribeCoverage request schema

See file wcpsDescribeCoverage.xsd

B.4 DescribeCoverage response schema

See file wcpsDescribeCoverage.xsd

B.5 ProcessCoverage request schema

See file wcpsProcessCoverage.xsd

B.6 Service exception schema

See file wcpsException.xsd.

Annex C
(normative)

UML Diagrams

C.1 Introduction

This annex provides a UML model of the WCS interface, using the OGC/ISO profile of UML summarized in Subclause 5.2 of the OWS Common [OGC 05-008].

The UML model of WCPS is based on the model of WCS [4].

- to be provided -

Annex D
(normative)

Conformance

D.1 Introduction

Specific conformance tests for a Web Coverage Processing Service will be added in a future revision of this specification. At the moment, a WCS implementation must satisfy the following system characteristics to be minimally conformant with this specification:

a) WCPS Clients and servers **must** support the GetCapabilities, DescribeCoverage, and ProcessCoverage operations.

b) WCPS clients **must** issue GetCapabilities requests conforming to WCS [4].

c) WCPS servers **must** respond to a GetCapabilities request with an XML document that conforms to WCS [4].

d) WCPS clients **must** issue DescribeCoverage requests conforming to WCS [4].

e) WCPS servers **must** respond to a DescribeCoverage request with an XML document that conforms to WCS [4].

f) WCPS clients **must** issue ProcessCoverage requests in Key-Value Pair (KVP) or XML form. ProcessCoverage KVP requests must conform to Subclause 10.5.1.1. ProcessCoverage XML requests must conform to Subclause10.5.1.2, and must be valid against the XML Schema definition in Subclause A.5.

g) WCPS servers **must** be able to respond to a ProcessCoverage operation with an XML document that conforms to the WCS GetCoverage response ([4] Subclause 10.3).

h) A WCS server **must** be able to deliver responses which are not exceptions according to the functionality specified in this standard and on the data provided by its GetCapbilities and DescribeCoverage responses without any vendor specific parameters in the requests.

i) All paragraphs in the normative clauses of this specification that use the keywords "required", "shall", and "shall not" must be satisfied.

Annex E
(normative)

SOAP transfer

NOTE This section is copied from WCS [4], thus ensuring compatibility between WCPS and WCS.

All compliant WCS servers may implement SOAP 1.2 transfer of all WCS operation requests and responses, using the XML encodings specified in the body of this document. When SOAP is implemented, the SOAP Request-Response message exchange pattern shall be used with the HTTP POST binding.

For SOAP transfer, each XML-encoded operation request shall be encapsulated in the body of a SOAP envelope, which shall contain only a body and only this request in that body. Similarly, each XML-encoded operation response shall be encapsulated in the body of a SOAP envelope, which shall contain only a body and only this response in that body. A WCS server shall return operation responses and error messages using only SOAP transfer when the operation request is sent using SOAP.

All compliant WCS servers shall specify the URLs to which SOAP operation requests may be sent, within the OperationsMetadata section of a service metadata (Capabilities) XML document, as specified in Subclause 8.3.2.

When an error is detected while processing an operation request encoded in a SOAP envelope, the WCS server shall generate a SOAP response message where the content of the **Body** element is a **Fault** element containing an ExceptionReport element. This shall be done using the following XML fragment:

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
    <soap:Body>
        <soap:Fault>
            <soap:Code>
                <soap:Value>soap:Server</soap:Value>
            </soap:Code>
            <soap:Reason>
                <soap:Text>A server exception was encountered.<soap:Text>
            </soap:Reason>
            <soap:Detail>
                <ows:ExceptionReport>
                    ...
                </ows:ExceptionReport>
            </soap:Detail>
        </soap:Fault>
    </soap:Body>
</soap:Envelope>
```

The Code element shall have the Value “soap:server” indicating that this is a server exception. The Reason element shall have the Text “Server exception was encountered.” This fixed string is used since the details of the exception shall be specified in the Detail element using an ows:ExceptionReport element as specified in OWS Common [OGC 05-008].

Annex F
(informative)

WCPS and WPS

F.1 Introduction

This Subclause describes the relation between WCPS and the OGC Web Processing Service (WPS) [WPS].

WPS has been designed to offer any sort of GIS functionality to clients across a network, including access to pre-programmed calculations and/or computation models that operate on spatially referenced data. WPS is targeted at processing both vector and raster data.

As such, there is an overlap between WCPS (and WCS) on the one hand and WPS on the other hand.

The difference can roughly be termed as WCPS offering a tight client/server coupling, while the WPS client/server coupling is loose. Tight coupling in this context means: the semantics is well defined, and the client knows exactly the mechanisms and their effects. Loose coupling refers to the WPS SOAP specification of services which formalizes only the funciton name and its input and output parameter types, while the semantics is described textually, i.e., understandable only by humans and not machine readable. WPS hence is excellently suited to bring online complex legacy code which is hard to describe in all details. WCPS, on the other hand, offers the potential for automatic capability detection and understanding, dynamic request composition and distribution, and hence automatic request cascading.

WCPS requests can be transcoded into WPS requests as described below. Discussion is limited to WCPS's **ProcessCoverage** request; **GetCapabilities** and **DescribeCoverage** requests are relying on WCS, hence WPS mapping needs to be described there.

NOTE WCPS primarily follows lock-step synchronization with WCS and OWS Common. Further, it attempts to be consistent with other OGC standards, such as WPS. Should there ever be a conflict between WCS and WPS, then WCPS will follow WCS.

NOTE These are not the only possible mappings, and they are not particularly endorsed – they serve solely for explanatory purposes. Further, they are not complete.

F.2 Process description

The WPS **DescribeProcess** operation allows to retrieve information about process specifics offered by the service. This description includes the input parameters and formats, plus the output formats.

A WCPS **ProcessCoverage** process description can be obtained from a WPS server by using the identifier "ProcessCoverage".

Example Information about the WCPS ProcessCoverage process can be requested from a suitably configured WPS KVP encoded for HTTP GET as follows (based on [WPS]):

```
http://foo.bar/foo?
  Service=WPS&
  Request=DescribeProcess&
  Version=0.4.0&
  Identifier=ProcessCoverage
```

Example The same in XML encoded for HTTP POST (based on [WPS]):

```
<?xml version="1.0" encoding="UTF-8"?>
<DescribeProcess service="WPS"
  version="0.4.0"
xmlns="http://www.opengeospatial.net/wps"
  xmlns:ows="http://www.opengeospatial.net/ows"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opengeospatial.net/wps
    ..\wpsDescribeProcess.xsd">
  <ows:Identifier>ProcessCoverage</ows:Identifier>
</DescribeProcess>
```

The **DescribeProcess** XML response can look as in the example below. The output description refers to the WCS **GetCoverage** response and, therefore, needs to be described there.

Example A possible DescribeProcess XML response (based on [WPS]):

```
<?xml version="1.0" encoding="UTF-8"?>
<ProcessDescriptions
  xmlns="http://www.opengeospatial.net/wps"
  xmlns:wps="http://www.opengeospatial.net/wps"
  xmlns:ows="http://www.opengeospatial.net/ows"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opengeospatial.net/wps
    ..\wpsDescribeProcess.xsd">
  <ProcessDescription processVersion="1"
    storeSupported="true" statusSupported="false">
  <ows:Identifier>ProcessCoverage</ows:Identifier>
  <ows:Title>Process one or more coverages.</ows:Title>
  <DataInputs>
    <Input>
      <ows:Identifier>Request</ows:Identifier>
      <ComplexData defaultFormat="text/XML"
        defaultEncoding="base64"
        defaultSchema=
            "http://foo.bar/wcps/0.3.0/
              wcpsProcessCoverage.xsd">
        <SupportedComplexData>
          <Format>text/XML</Format>
          <Encoding>UTF-8</Encoding>
          <Schema>
            http://foo.bar/wcps/0.3.0/
              wcpsProcessCoverage.xsd
          </Schema>
        </SupportedComplexData>
      </ComplexData>
      <MinimumOccurs>1</MinimumOccurs>
      </Input>
```

```
    </DataInputs>
    <ProcessOutputs>
      <Output>
        <ows:Identifier>
          ProcessCoverageResultList
        </ows:Identifier>
        <ComplexOutput …>
          … <!—to be described by WCS -->
        </ComplexOutput>
      </Output>
    </ProcessOutputs>
  </ProcessDescription>
</ProcessDescriptions>
```

F.3 Process execution

A WCPS **ProcessCoverage** request can be mapped to a WPS **Execute** request. The corresponding WPS request structure may look as follows.

The WCPS **ProcessCoverage** request can be described as a WPS **Execute** response structure as follows.

Example In WPS KVP notation, a WCPS **ProcessCoverage** request can be phrased as follows:

```
http://foo.bar/foo?
  request="Execute"&
  service="WPS"&
  version="0.3.0"&
  Identifier="ProcessCoverage"&
  DataInput=
    "for%20C%20in(A)%20
       return%20
         store(
           encode(C.red%20+%20C.nir,%22tiff%22)
           )"
```

Example An example ProcessCoverage encoding as WPS Execute operation request using XML encoding is:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Execute service="WPS" version="0.4.0"
  status="false"
  xmlns="http://www.opengeospatial.net/wps"
  xmlns:ows="http://www.opengeospatial.net/ows"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opengeospatial.net/wps
    ..\wpsExecute.xsd"
>
  <ows:Identifier>ProcessCoverage</ows:Identifier>
  <DataInputs>
    <Input>
      <ows:Identifier>Request</ows:Identifier>
    </Input>
  </DataInputs>
  <OutputDefinitions>
    <Output>
      <ows:Identifier>
```

```
            ProcessCoverageResultList
          </ows:Identifier>
        </Output>
      </OutputDefinitions>
    </Execute>
```

The response of a WCPS **ProcessCoverage** request adheres to the WCS **GetCoverage** resposne and, therefore, needs to be described there.

Bibliography

[1] –unused-

[2] ISO 19103, Geographic Information – Conceptual schema language

[3] OMG Unified Modeling Language Specification (UML), Version 1.5, March 2003, http://www.omg.org/docs/formal/03-03-01.pdf

[4] (Whiteside, Evans 2006) OGC 03-083r8, Web Coverage Service (WCS) Implementation Specification, Version 1.1.0

[5] European Petroleum Survey Group, EPSG Geodetic Parameter Set, Version 6.8

[6] IETF RFC 2396

[7] IETF RFC 2616

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

[9] OGC 05-007r4, Web Processing Service Implementation Specification, Version 0.4.0

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