CONSTRUCTOR UNIVERSITY

Establishing a Compliance Test for a Geo Datacube Query Language

by

Brook Balkachew Nigatu

Master Thesis in Computer Science and Software Engineering

Submission: April 27, 2026 Supervisor: Prof. P. Baumann

Constructor University | School of Computer Science and Engineering

Statutory Declaration

Family Name, Given/First Name	Nigatu, Brook Balkachew
Matriculation number	30007993
Kind of thesis submitted	Master Thesis

English: Declaration of Authorship

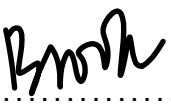
I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

27/04/2026 

Date, Signature

Abstract

The Web Coverage Processing Service (WCPS) Language Interface Standard defines a query language in the geospatial domain that can be used to request server-side processing of large-scale multi-dimensional datacubes. This language interface standard is maintained by the Open Geospatial Consortium (OGC), which also operates a conformance testing and validation program for its standards. Currently, no conformance test suite exists for the WCPS language standard. This thesis aims to address that gap by establishing such a test suite, built on the OGC TEAM (Test, Evaluation, And Measurement) Engine test harness, that can be used to validate WCPS implementations.

Contents

1 Introduction

Geospatial big data increasingly plays a vital role in various scientific and industrial domains. Geospatial technology serves as a powerful tool for measuring, modeling, mapping, and predicting real-world phenomena across a wide range of public and private sectors, including the environment and agriculture, urban planning and infrastructure, public health, economy, business and more [25].

A central concept that has emerged in working with such data is that of coverages, which digitally represent spatio-temporal data, such as 3D satellite imagery time-series. Due to their ubiquitous nature in different industrial and scientific domains, well-defined standards that govern the structure and handling of coverages are necessary to ensure interoperability among different systems and implementations. Indeed, standardization bodies such as the Open Geospatial Consortium (OGC) and the International Organization for Standardization (ISO) have developed standards that target several aspects of working with coverage data.

One such standard is OGC's Web Coverage Service (WCS) standard [15], which specifies how geospatial data is accessed over the internet. It comes with several extensions, one of which is the processing extension [6]. Web Coverage Processing Service (WCPS) [5], which is the focus of this thesis, is the standard query language used for the processing extension of WCS.

Along with the standards it maintains, the OGC also operates a conformance testing and validation program [12], where a given implementation of a standard (or part of a standard) can be validated against a dedicated test suite. Currently, there is no conformance test suite for the WCPS standard.

The Test, Evaluation, And Measurement (TEAM) Engine [7] is the OGC's official test harness for running conformance test suites. The ultimate goal of this thesis is to design and develop a conformance test suite for the WCPS standard that can be executed on the TEAM Engine test harness.

Section 2 provides an overview of the main concepts and background work related to this thesis. The scope of testing is presented in section 3, and implementation details are discussed in section 4. Finally, limitations are highlighted in section 5, and conclusions and future work are indicated in section 6.

The annexes provide insights into the development setup and process A, the queries used in the test suite B, and issues identified during development C.

2 Key Concepts and Background Work

This section provides an overview of the main concepts and standards relevant to this thesis, namely the coverage model, the Web Coverage Service (WCS) standard [15], the processing extension of WCS [6], the Web Coverage Processing Service (WCPS) standard [5], and the Coverage Implementation Schema (CIS) standard [13]. Detailed discussions of these concepts and standards are also presented in [3]. We also discuss the OGC's Compliance Interoperability & Testing Evaluation (CITE) program [12].

2.1 Coverage Model

Coverages are digital representations of data that vary across multi-dimensional spaces. They are widely used to represent spatio-temporal data, i.e., data that varies across space and time, such as satellite imagery, sensor and weather data, and more.

A general overview of the coverage model described in [2] and adopted by the OGC standards is presented here.

[3] identifies the following main components of coverages:

- **Domain Set**: This is the set of positions where a coverage is directly defined. Using interpolation, the values at these positions can also be used to derive values at other positions not included in the domain set.
- **Range Type**: This describes the type of data values associated with the positions. This can either be atomic values like integers, or composite values with multiple fields.
- **Range Set**: This is the set of data values associated with the positions in the domain set.
- **Metadata**: The coverage can also have associated metadata. The exact structure and constituents of this are not prescribed by the standards, and left entirely to implementers.

A helpful abstraction from [2] is to think of a coverage C as a mathematical function:

$$C : D \to \mathcal{P}(V) \setminus \emptyset$$

where D is the domain set and $\mathcal{P}(V)$ is the power set of the range set V. Note that multiple values can be associated with a single position, and at least one, possibly null, value with each position.

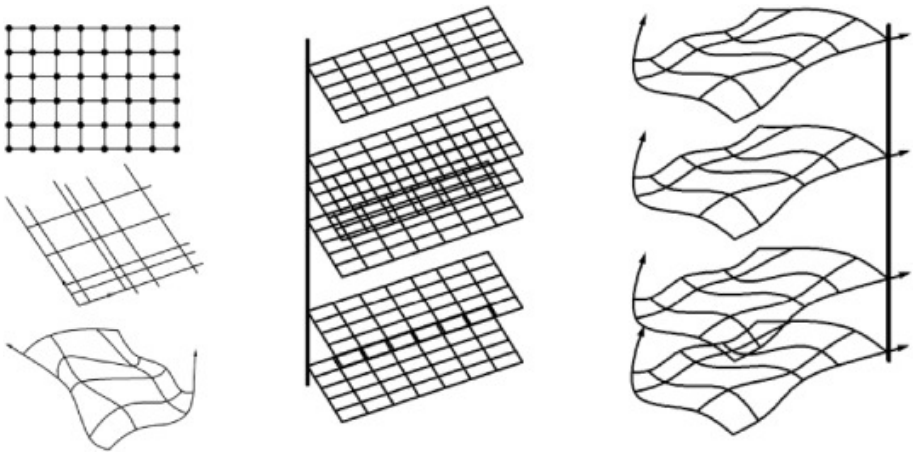
Based on the structure of the domain set, various types of coverages can be identified. Of these, grid coverages, also known as datacubes, are the most commonly used. They are primarily used to represent multi-dimensional raster data such as satellite imagery.

In a grid coverage, the domain set is defined in terms of a sequence of $n \geq 1$ axes. Keeping the notation from [2], a non-empty set of points G_i is sampled along each axis a_i. The domain set is then constructed as the set of all tuples $(g_1, g_2, \ldots, g_n)$ where $g_i \in G_i$.

The nature of the points in G_i can vary, leading to different axis types. For example, the points could be consecutive integers or generally equidistant numbers, or completely

irregular. The definition of a grid coverage is fine-grained in that different axis types can be combined to form the domain set, as the figure below from [2] shows.

The sequence of axes is defined by the coverage's Coordinate Reference System (CRS). Many available CRSs can be found in [10]. These can also be combined to form compound CRSs containing axes from multiple CRSs.



Grids arising from different combinations of axis types

2.2 OGC Standards

Here we discuss the OGC standards relevant to this thesis. Although the OGC versions are presented here, ISO has also adopted these standards, and the corresponding standards are kept in sync.

A reference implementation of these standards, made available by the rasdaman [9] team at [24], has been used for testing and validation during the development of the test suite.

2.2.1 Coverage Implementation Schema (CIS)

CIS [13] builds on the coverage model described above and consolidates it into a *"concrete, interoperable, conformance-testable coverage structure"*. This structure is not dependent on any specific encoding format; rather, it can be expressed in various formats such as GML (Geography Markup Language), JSON, NetCDF and more.

Different types of coverages are supported. Grid coverages, which are discussed above, are unified into a *GeneralGridCoverage* structure that handles various geometries.

The following is the CIS 1.1 GML encoding of a 3D grid coverage with two regular axes and one irregular axis. Some parts were removed for brevity.

```
<?xml version="1.0" encoding="UTF-8"?>
<cis11:GeneralGridCoverage gml:id="newCov" ...>
    <cis11:CoverageFunction>
        <gml:GridFunction>
            <gml:sequenceRule axisOrder="+3 +2 +1">Linear</
               gml:sequenceRule>
            <gml:startPoint>0 0 0</gml:startPoint>
        </gml:GridFunction>
```

```
</cis11:CoverageFunction>
<cis11:Envelope ...>
    <cis11:AxisExtent axisLabel="lat" uomLabel="degree"
       lowerBound="-2" upperBound="2"/>
    <cis11:AxisExtent axisLabel="long" uomLabel="degree"
       lowerBound="-2" upperBound="2"/>
    <cis11:AxisExtent axisLabel="day" uomLabel="d" lowerBound="
       2024-01-01T00:00:00.000Z"
    upperBound="2026-01-01T00:00:00.000Z"/>
</cis11:Envelope>
<cis11:DomainSet>
    <cis11:GeneralGrid ...>
        <cis11:RegularAxis axisLabel="lat" uomLabel="degree"
           lowerBound="-2"
        upperBound="2" resolution="-0.5"/>
        <cis11:RegularAxis axisLabel="long" uomLabel="degree"
           lowerBound="-2"
        upperBound="2" resolution="1"/>
        <cis11:IrregularAxis axisLabel="day" uomLabel="d">
            <cis11:C>"2024-01-01T00:00:00.000Z"</cis11:C>
            <cis11:C>"2025-01-01T00:00:00.000Z"</cis11:C>
            <cis11:C>"2026-01-01T00:00:00.000Z"</cis11:C>
        </cis11:IrregularAxis>
        <cis11:GridLimits srsName="http://www.opengis.net/def/
           crs/OGC/0/Index3D" ...>
            <cis11:IndexAxis axisLabel="i" lowerBound="0"
               upperBound="7"/>
            <cis11:IndexAxis axisLabel="j" lowerBound="0"
               upperBound="3"/>
            <cis11:IndexAxis axisLabel="k" lowerBound="0"
               upperBound="2"/>
        </cis11:GridLimits>
    </cis11:GeneralGrid>
</cis11:DomainSet>
<cis11:RangeSet>
    <cis11:DataBlock>
        <cis11:V>1</cis11:V>
                 .
                 .
                 .
        <cis11:V>1</cis11:V>
        <cis11:V>1</cis11:V>
    </cis11:DataBlock>
</cis11:RangeSet>
<cis11:RangeType>
    <swe:DataRecord>
        <swe:field name="elem">
            <swe:Quantity definition="http://www.opengis.net/
               def/dataType/OGC/0/signedInt">
                <swe:label>elem</swe:label>
                <swe:uom code="10^0"/>
            </swe:Quantity>
        </swe:field>
    </swe:DataRecord>
</cis11:RangeType>
<cis11:Metadata>
```

```
        <ras:covMetadata/>
    </cis11:Metadata>
</cis11:GeneralGridCoverage>
```

Since the standard defines a concrete structure, systems implementing it can be accurately tested for conformance, which ensures interoperability across different implementations exchanging coverage data.

2.2.2 Web Coverage Service (WCS) and Extensions

The Web Coverage Service (WCS) standard [15] defines how coverage data can be accessed over the internet. The standard defines the following core set of operations:

- **GetCapabilities**: This allows a client to request metadata about the services offered by a WCS server, including the list of available coverages, and supported functionality.
- **DescribeCoverage**: This allows a client to request a detailed description of a specific coverage such as its spatio-temporal extent and data structure.
- **GetCoverage**: This allows a client to request the retrieval of coverage data, with optional domain subsetting to retrieve only a portion of a, possibly large, coverage.

The standard provisions the existence of extensions that can be used to add functionality. In fact, according to the standard, it is mandatory for a WCS server to support at least one protocol extension and at least one coverage encoding format extension, because the core standard specifies neither a means of communication between a client and a server, nor a coverage format to use.

One protocol extension is the KVP (key/value pair) Protocol Binding Extension [4]. It defines the structures of HTTP GET requests that can be made to a server for the above mentioned operations.

For example, a *GetCapabilities* request in this protocol has the following form:

```
https://ows.rasdaman.org/rasdaman/ows?service=WCS&request=
    GetCapabilities
```

Another extension of particular interest to this thesis is the Processing Extension [6]. It adds the *ProcessCoverages* request type, which can be used for server-side processing of coverage data. This is important because coverages can be very large, and downloading them for client-side processing can be inefficient or even infeasible.

Among other protocol bindings, the extension supports the GET/KVP binding by extending the one for WCS mentioned just above with this new request type and a "query" field for the processing instructions. A sample request can be found in section 4.2.

Unlike simple subsetting operations allowed by the *GetCoverage* request type, this extension enables clients to use a declarative language to specify complex processing operations and retrieve the results. The standard defining this language is discussed next.

2.2.3 Web Coverage Processing Service (WCPS)

WCPS [5] defines the query language to be used for the processing extension of WCS, although it is not constrained to the WCS framework. Several constructs are defined to

support various operations on coverages. Many of these language features are discussed in the next section.

A simple query is shown below:

```
for $c in (coverage1)
return encode($c.red * 2, "application/gml+xml")
```

This query retrieves the red channel of the coverage named "coverage1", which could contain satellite imagery data for example, multiplies all range values by 2, and encodes the result in GML format.

Since the query language is declarative, clients need not concern themselves with implementation details of the processing logic, and can focus on specifying the desired high level operations.

Currently, only operations on grid coverages, which were discussed earlier, are supported, but enhancements to the standard are planned to support more coverage types.

The aim of this thesis is to establish a conformance test suite for this standard against which implementations can be validated. The standard comes with an Abstract Test Suite (ATS), which has guided the design of the test suite established in this thesis.

2.3 Compliance Interoperability & Testing Evaluation (CITE) Program

The CITE program [12] is OGC's official initiative dedicated to ensuring software compliance with OGC standards, with the core mission of ensuring interoperability across different systems in the geospatial domain. Tools and support are provided to developers to validate their implementations against OGC standards.

The initiative relies on two primary open-source components:

- **TEAM (Test, Evaluation, And Measurement) Engine**: The core platform that executes tests. It provides a web-based interface and API for running test sessions and viewing results.
- **Executable Test Suites (ETS)**: Specific sets of tests developed for individual OGC standards (e.g., WMS, WFS, GeoPackage, OGC APIs). These are implemented using frameworks like TestNG (for modern tests) or CTL (Compliance Test Language, for legacy tests).

The product of this thesis is an ETS for the WCPS standard that can be executed on the TEAM Engine test harness to validate WCPS implementations. The design and implementation details of this test suite are discussed in the next sections.

3 Scope of Testing

In this section, we discuss the scope of the test suite in terms of the features and requirements that are tested. These language elements are all described in greater detail in the WCPS standard [5]. A full list of the queries used in the test suite can be found in appendix B. Issues discovered in the standard and reference implementation during the design and development of the test suite are documented in appendix C.

3.1 xWcps Features

The test suite mostly focuses on xWcps features. xWcps is an extension of WCPS specifically for OGC CIS 1.1 coverages [13] and represents a small enough scope for initial testing efforts.

3.1.1 letExpr

A *letExpr* defines a named constant that can be used in the rest of the query. Unlike variables in common programming languages, these can not be reassigned. The semantics is simply that the name is replaced by the assigned expression wherever it appears in the rest of the query. Multiple let clauses can be declared in a query, and each clause can contain multiple constant definitions. The syntax can be observed from the following snippets:

```
let $x := 5
let $y := 10
```

```
let $x := 5, $y := 10
```

3.1.2 xWcpsCoverageConstructorExpr

An *xWcpsCoverageConstructorExpr* allows one to define arbitrary coverages within a query. The domain and range sets, range type, and even metadata can all be specified. The following snippet shows its main syntax elements:

```
coverage newCov
domain crs "OGC:Index2D" with
  x index (0:1),
  y index (0:1)
range type
  elem quantity int
range set
  1
```

The Coordinate Reference System (CRS) here is represented as a Compact URI (CURIE), which is specified in [14]. One list of available CRSs from the European Petroleum Survey Group (EPSG) can be found for example in [10].

3.1.3 switchExpr

A *switchExpr* allows choosing values based on case distinctions. The output can either be a scalar value or a coverage depending on the conditions and return values on each case. The following snippets show example usages:

```
switch
  case $sc
      > 0
    return
       1
  case $sc
      < 0
    return
       -1
  default
    return
       0
```

```
switch
  case $c0 > 0
    return $c1
  case $c0 < 0
    return $c2
  default
    return $c3
```

Expressions in the case conditions must all be boolean scalars or boolean-valued coverages with the same domains. Similarly, return expressions must all be scalars of the same type or coverages having the same range types with each other and the same domains as the case condition coverages. The output shall be a scalar if all expressions are scalars, or otherwise a coverage with the same domain as the case condition and/or return coverages.

The semantics is that of evaluating for each output position the case conditions in their order of declaration and selecting the value of the corresponding return expression at that position. In the context of this statement, a scalar expression should be understood to only have one position or the same value at all positions.

3.2 Induced Operations

From the standard: *"Induced operations allow to simultaneously apply a function originally working on a single value to all grid point values of a coverage. In case the range type contains more than one component, the function shall be applied to each point simultaneously."*

Many unary and binary operations that operate on scalars "induce" operations on coverages. For example, the addition operator "+" should enable adding two compatible coverages at each position.

3.2.1 Unary Induced Operations

These come from operators that operate on a single value. The following mathematical operations on a coverage C are specified in the standard and tested for:

- $+C$
- $-C$
- $sqrt(C)$
- $abs(C)$
- $sin(C)$
- $cos(C)$
- $tan(C)$
- $sinh(C)$
- $cosh(C)$
- $tanh(C)$
- $arcsin(C)$
- $arccos(C)$
- $arctan(C)$
- $exp(C)$
- $log(C)$
- $ln(C)$
- $pow(C, p)$
- $not(C)$
- $bit(C, p)$[1]

Range field extraction and type casting also induce unary operations. `C.fieldName` gives a single-valued coverage containing values for the `fieldName` range field at every point in the domain. `(target_type)C` casts each range value into the target type.

[1] $bit(x, p)$ returns a boolean value indicating whether the p-th bit of x is set.

3.2.2 Binary Induced Operations

These come from common binary arithmetic and logical operators. In addition to supporting operations between two compatible coverages (and of course two scalars), the standard also defines them between a coverage and a scalar. Naturally, the operation is applied with the same scalar across all positions of the coverage. The following operations are specified in the standard and tested for:

- $X + Y$
- $X - Y$
- $X * Y$
- X / Y
- $X \; and \; Y$
- $X \; or \; Y$
- $X \; xor \; Y$
- $X = Y$
- $X < Y$
- $X > Y$
- $X <= Y$
- $X >= Y$
- $X \; ! = \; Y$
- $X \; overlay \; Y$[2]

3.3 Condense Operations

Condense operations allow for the aggregation of several values into a single scalar value using binary operators.

3.3.1 generalCondenseExpr

A *generalCondenseExpr* follows the following syntax:

```
condense op
   over name1 axis1 (l1:r1),
        name2 axis2 (l2:r2), ...
        nameN axisN (lN:rN)
   where condition
   using expression
```

condition is a boolean expression that determines which positions are considered, and *expression* is a scalar expression of type compatible with *op*. The *name⟨i⟩* variables can be used in *condition* and *expression*, for example to obtain values of a coverage at specific coordinates.

The standard specifies +, *, *max*, *min*, *and*, and *or* as admissible operators.

3.3.2 reduceExpr

A *reduceExpr* is a special case of the *generalCondenseExpr*. It outputs a summary value for the entire coverage.

For a given coverage *C* of appropriate range type, the expressions *add(C), max(C), min(C), some(C)*, and *all(C)* respectively apply +, *max*, *min*, *or*, and *and* across the entire coverage. In addition, *avg(C)* gives the mean value of a numeric-valued coverage, and *count(C)* gives the number of *true* values in a boolean-valued coverage.

[2] $X \; overlay \; Y$ replaces values at positions where X is 0 with values of Y at those positions.

4 Implementation

The full source code of this test suite is available on GitHub at [17]. See appendix A for details on how the development environment was setup. A complete list of the queries used can be found in appendix B.

The test suite was built on the maven archetype [21] provided by TEAM Engine. The generated project consists of an example test suite implemented in Java using the TestNG framework, along with a wrapper CTL (Conformance Testing Language) script. The CTL script serves as the entry point for the test suite and defines UI elements that can be used to describe the test suite and receive arguments on the TEAM Engine web interface. All test logic resides in the Java code.

4.1 Test Data

In order to test some of the features discussed in the previous section, coverage data on which processing operations can be performed is needed. We explored two approaches for this: using predefined coverages, which would be required to be present in the test subjects, and generating test coverages within queries.

4.1.1 An Accompanying Set of Test Coverages

One approach is to provide an accompanying set of test coverages that must be uploaded to the test subject before conformance testing is run. This approach has the advantage of leading to simpler test queries that entirely focus on the exact features being tested without the additional noise of coverage generation. This is the approach taken for instance by the conformance test suite for the OGC Web Map Service (WMS) standard [18].

However, the disadvantage is that there is more setup required, and changes to the test suite may potentially need to be synchronized with an external data set.

4.1.2 Generating Test Coverages within Queries

As discussed above, the *xWcpsCoverageConstructorExpr* construct allows for the construction of coverages, which can then be used in processing operations. This approach has the advantage of enabling a fully self-contained test suite. Changes can be localized to a single codebase, and implementers need not preload specific data into their servers.

In the end, this approach was chosen, with each query constructing its own test coverage(s) as needed.

4.2 Sending Queries

Test queries and expected outputs (oracles) were defined within the test suite codebase in Java files, with each query-oracle pair corresponding to a test case.

Queries are sent using the GET/KVP binding defined in the processing extension of WCS [6], by populating the "query" parameter with the desired URL-encoded query.

The complete request URL for the query

```
for $c in ( AvgTemperatureColorScaled )
return encode($c[ansi("2014-07")].red, "application/json")
```

looks like

```
https://ows.rasdaman.org/rasdaman/ows?
query=for%20%24c%20in%20(%20AvgTemperatureColorScaled%20)
%0A%20%20%20%20return%20encode(%24c%5Bansi(%222014-07%22)
%5D.Red%2C%20%22application%2Fjson%22)
&REQUEST=ProcessCoverages
&SERVICE=WCS
&VERSION=2.0.1
```

Another important implementation detail comes from the fact that test coverages are generated within queries. While this removes the need for predefined coverages to be present in test subjects, the language grammar defined in the WCPS standard enforces a non-empty "for" clause in queries, which means that a coverage already present in the test subject needs to be referenced.

Test subjects must, therefore, contain at least one coverage and the test suite should somehow discover the name of that coverage. The way it is done in this test suite is by using XPath to obtain a coverage name from the response to a "GetCapabilities" request to the test subject. The structure of this request is specified in the WCS KVP extension [4]. The precise XPath expression evaluated against that response to obtain a coverage name is

```
(//*[local-name()='CoverageId'])[1]
```

A for-clause with the coverage name obtained this way is added to all test queries before they are sent.

4.3 Output Format and Validation

The test suite uses the widely popular JSON format for validating coverage outputs, so test subjects are expected to support it. JSON was chosen as it can be used to represent coverage data of arbitrary dimensionality as multidimensional JSON arrays. Range values with multiple fields are expected as space-separated strings in the JSON output.

Here are two example outputs:

```
                    [[["1 false", "1 true"], ["3 false",
[[1, 2],               "3 true"]],
   [3, 4]]          [["5 true", "5 false"], ["7 true", "7
                       false"]]]
```

A binary format, such as NetCDF, is not necessary here as output from the processing of test queries is expected to be small. Validation focuses on the data values in the range set, so metadata also need not be included in the output. Therefore, coverage outputs in the form of (multidimensional) JSON arrays were found to be sufficient for the needs of the test suite.

JSON validation (for coverage outputs) was implemented using the JSONAssert assertion library [26]. To deal with floating-point comparisons, which would be necessary for some test cases, a custom comparator that allows for a small margin of error (10^{-3}) when comparing numeric values was implemented and used in validating outputs.

The same holds for scalar outputs. Numeric values are validated within this margin of error to account for floating-point precision differences between oracles and outputs.

For simplicity, floating-point outputs in multi-field range values are not handled and are not expected in any test cases. Null values are also not tested for. However, support for these can be implemented by extending the custom comparator used in JSON validation.

Error cases are also tested for by sending invalid queries. The WCS Processing Extension [6] specifies that an exception should be signalled with an error code, so error cases are validated by checking for the status code of the incoming response.

5 Limitations and Future Work

The developed test suite provides a solid foundation for validating implementations of WCPS 1.1, particularly focusing on xWcps features. However, limitations remain that offer opportunities for future enhancement. While there may be others, the ones discussed here are the non-exhaustiveness of the test suite, specification and implementation issues, JSON-only coverage output validation, and the handling of floating-point comparisons. Although performance metrics are not part of the WCPS standard and are thus not necessary considerations for conformance testing, they are discussed here as potential optional additions to the test suite.

Despite these limitations, this work represents an important step towards a comprehensive conformance test suite for WCPS, as much of the heavy lifting involved in setting up a test suite has been done, and several core language features have been covered. Learnings presented in this thesis can also guide future progress.

5.1 Non-Exhaustiveness

While the current test suite covers a significant portion of the standard, it is not exhaustive. Not all language features defined in the standard are tested for. However, the project's setup makes it straightforward to add new test cases, so future work can expand the test suite to cover more language features and modify it as changes are made to the standard.

5.2 Specification and Implementation Issues

During the development of this suite, a few inconsistencies and ambiguities were also discovered in both the WCPS standard and the rasdaman reference implementation. These issues, which are detailed in appendix C, can guide future work to address these findings in subsequent revisions of the standard and/or the reference implementation, followed by corresponding updates to the test suite.

5.3 Coverage Output Format

The test suite currently relies exclusively on JSON for validating coverage data. While JSON is popular and efficient for the small test coverages used, it restricts the suite's applicability to servers that implement this specific encoding. Expanding support to include other standard formats, such as GML, could be worthwhile.

5.4 Floating-Point Comparisons

The threshold of 10^{-3} used for floating-point comparisons was somewhat arbitrary and may be too loose or too tight in different scenarios. A more principled approach to determining this threshold, depending possibly on the specific operations being tested or the values involved, could be explored. Making this threshold an externally configurable parameter may also be an option.

5.5 Performance Considerations

The current scope is strictly limited to functional conformance testing. Performance measurements, such as query execution time, were not considered part of this work. Inte-

grating performance benchmarks into the suite, for instance by failing some optional tests when performance thresholds are not met, could be considered if accurate measurements can be made, especially over the web.

6 Conclusion

This thesis has successfully established an executable conformance test suite for the WCPS 1.1 standard that runs on the OGC CITE program's TEAM Engine. It covers many of the language elements from the standard, with a special focus on xWcps features. The core arithmetic, logical, and aggregation operations are also tested for. Language constructs marked for deprecation, such as the original WCPS coverage constructor and coverage probing functions have been left out.

In line with CITE's vision, a primary objective of this test suite is to ensure interoperability across different systems reliant on the standard by providing a canonical means of validating conformance. Furthermore, the presence of an open-source test suite can help identify issues in implementations, leading to more reliable datacube services. This promotes adoption by boosting confidence among users, who will additionally have the advantage of not being tied to any particular implementation but instead to a widely accepted standard. The evolution of the standard also greatly benefits, because the feedback loop from concrete testing efforts can reveal specification problems or ambiguities.

A notable feature of the test suite is that it is self-contained, as it does not require predefined test coverages to be preloaded into test subjects. What the test suite does require from a test subject is for it to contain at least one coverage and support encoding coverage output as multidimensional JSON arrays.

As discussed in the previous section, the test suite is not without its limitations, which could be addressed in future iterations of the project. The findings from this work also offer feedback for the refinement of the WCPS specification and the rasdaman implementation.

References

[1] Apache Software Foundation. *Installation*. https://maven.apache.org/install.html. Apache Maven installation instructions. Accessed: 2026-04-26.

[2] Peter Baumann. "A general conceptual framework for multi-dimensional spatio-temporal data sets". In: *Environmental Modelling and Software* 143 (2021), p. 105096. ISSN: 1364-8152. DOI: https://doi.org/10.1016/j.envsoft.2021.105096. URL: https://www.sciencedirect.com/science/article/pii/S1364815221001390.

[3] Peter Baumann. "An introduction to the OGC/ISO coverage and datacube standard for modeling multi-dimensional, spatio-temporal Big Data". In: *Big Earth Data* 0.0 (2025), pp. 1–54. DOI: 10.1080/20964471.2025.2585732. eprint: https://doi.org/10.1080/20964471.2025.2585732. URL: https://doi.org/10.1080/20964471.2025.2585732.

[4] Peter Baumann. *OGC Web Coverage Service 2.0 Interface Standard – KVP Protocol Binding Extension*. 09-147r1. OpenGIS Interface Standard. Version 1.0.0. 2010. URL: http://www.opengis.net/doc/IS/WCS_protocol-binding_get-kvp/1.0.

[5] Peter Baumann. *Open Geospatial Consortium Web Coverage Processing Service (WCPS) Language Interface Standard*. 08-068r3. OGC Implementation Standard. Version 1.1. 2021. URL: https://docs.ogc.org/is/08-068r3/08-068r3.html.

[6] Peter Baumann and Jinsongdi Yu. *OGC Web Coverage Service (WCS) Interface Standard - Processing Extension*. OGC 08-059r4. OGC Interface Standard. Version 2.0. Feb. 2014. URL: http://www.opengis.net/doc/IS/wcs_processing_extension/2.0.

[7] Open Geospatial Consortium. *TEAM Engine: Test, Evaluation, And Measurement Engine*. https://github.com/opengeospatial/teamengine. Official test harness of the OGC Compliance Testing Program (CITE). Licensed under Apache 2.0.

[8] Centron GmbH. *Step-by-Step Guide: Install Tomcat on a Linux System*. 2025. URL: https://www.centron.de/en/tutorial/step-by-step-guide-install-tomcat-on-a-linux-system/ (visited on 02/05/2026).

[9] rasdaman GmbH. *rasdaman, the Big Data Analytics Server*. URL: https://www.rasdaman.com/.

[10] MapTiler. *EPSG.io: Coordinate Systems Worldwide*. Accessed: 2026-04-26. URL: https://epsg.io/.

[11] Brook Nigatu. *Correct target folder for mounting JAR dependencies in teamengine-web*. https://github.com/opengeospatial/teamengine/pull/649. GitHub pull request. Apr. 2026.

[12] Open Geospatial Consortium. *OGC Compliance Program (CITE) Wiki*. GitHub repository, accessed: 2026-04-26. URL: https://github.com/opengeospatial/cite/wiki.

[13] Open Geospatial Consortium. *OGC Coverage Implementation Schema with Corrigendum*. Ed. by Peter Baumann, Eric Hirschorn, and Joan Masó. OGC 09-146r8. Version 1.1.1. Oct. 2019. URL: https://docs.ogc.org/is/09-146r8/09-146r8.html.

[14] Open Geospatial Consortium. *OGC Name Type Specification - definitions - part 1 – basic name*. Ed. by Gobe Hobona and Simon Cox. OGC 09-048r7. Version 1.4. OGC Policy (not a Standard). July 2024. URL: https://docs.ogc.org/pol/09-048r7.html.

[15] Open Geospatial Consortium. *OGC Web Coverage Service (WCS) 2.1 Interface Standard – Core*. 17-089r1. OGC Implementation Standard. Version 2.1. 2018. URL: https://docs.ogc.org/is/17-089r1/17-089r1.html.

[16] Open Geospatial Consortium. *TEAM Engine Installation Guide*. https://opengeospatial.github.io/teamengine/installation.html. Accessed: 2026-04-26.

[17] Open Geospatial Consortium (OGC). *Executable Test Suite for WCPS 1.1*. https://github.com/opengeospatial/ets-wcps11. Repository for the OGC Web Coverage Processing Service 1.1 Test Suite.

[18] Open Geospatial Consortium (OGC). *OGC Web Map Service (WMS) 1.3.0 Conformance Test Suite*. https://opengeospatial.github.io/ets-wms13/. Accessed: 2026-04-26.

[19] Open Geospatial Consortium (OGC). *TEAM Engine – User Guide*. https://opengeospatial.github.io/teamengine/users.html. Accessed: 2026-04-26.

[20] Open Geospatial Consortium (OGC). *TEAM Engine – Conformance Testing with TestNG, Part 1: Essentials*. https://opengeospatial.github.io/teamengine/testng-essentials.html. Accessed: 2026-04-26.

[21] Open Geospatial Consortium (OGC). *Test Suite Archetype - TestNG Framework*. https://github.com/opengeospatial/ets-archetype-testng. Version 2.7. 2019.

[22] Oracle Corporation. *Java Downloads*. https://www.oracle.com/java/technologies/downloads/. Accessed: 2026-04-26.

[23] Postman, Inc. *The World's Leading API Platform*. https://www.postman.com/. Accessed: 2026-04-26.

[24] rasdaman GmbH. *rasdaman OGC Web Service Interface*. https://ows.rasdaman.org/rasdaman/ows.

[25] Salman Selmy, Dmitry Kucher, and Yujian Yang. "Geospatial Data: Acquisition, Applications and Challenges". In: Oct. 2024. ISBN: 978-1-83769-829-5. DOI: 10.5772/intechopen.1006635.

[26] Skyscreamer. *JSONassert: Write JSON unit tests in less code. Great for testing REST interfaces*. https://github.com/skyscreamer/JSONassert. Version 2.0-rc1. 2024.

A Development Setup

All work was done on Ubuntu 24.04 accessed via Windows Subsystem for Linux (WSL) on a Windows 11 machine.

A.1 TEAM Engine Local Setup

The Test, Evaluation And Measurement (TEAM) Engine is the OGC's official test harness of the OGC Compliance Testing Program [12]. It is used to run the conformance test suites for various OGC and ISO standards. Indeed, the WCPS test suite developed in this thesis is designed for the TEAM Engine test harness.

During development, a local instance of TEAM Engine was installed and set up. The web version of the TEAM Engine, running on a Tomcat server instance, was used throughout the project to execute tests.

The installation steps detailed in [16] were largely followed for this task. Detailed steps are described here.

1. Install Java

Java Development Kit (JDK) 17 is prescribed by [16]. Oracle JDK 17 was used. Specifically, the JDK Debian package for 64-bit Linux from the (.deb) file available at [22] was downloaded and installed by running:

```
sudo dpkg -i jdk-17_linux-x64_bin.deb
```

2. Install Git and Apache Maven

Git was installed by running:

```
sudo apt install git
```

The similar apt command for maven only installs maven 3.8, which is not the latest release. Instead the following sequence of commands, based on instructions in [1] was used:

```
wget https://dlcdn.apache.org/maven/maven-3/3.9.14/binaries/\
  apache-maven-3.9.14-bin.zip

unzip apache-maven-3.9.14-bin.zip
sudo mkdir -p /opt/maven
sudo mv apache-maven-3.9.14 /opt/maven/
```

3. Update Environment Variables

The following snippet was added to the end of the /.bashrc file to set the environment variables for maven and java:

```
export M2_HOME=/opt/maven/apache-maven-3.9.14
export JAVA_HOME=/usr/lib/jvm/jdk-17.0.12-oracle-x64/
export PATH=$M2_HOME/bin:$PATH
```

4. Clone and Build TEAM Engine

TEAM Engine was built from its source code available on GitHub [7]:

```
git clone https://github.com/opengeospatial/teamengine.git
cd teamengine
mvn package
```

5. Set up an Instance Directory (TE__BASE)

A directory to contain important resources such as scripts and dependencies for TEAM Engine needs to be set up:

```
mkdir ~/TE_BASE

unzip teamengine-console/target/\
  teamengine-console-*-base.zip -d ~/TE_BASE
```

6. Install Apache Tomcat

Steps detailed in [8] were followed to set up the Tomcat server instance, except that the systemd service file was modified to point to the right installation of Java and add TEAM Engine specific environment variables as prescribed in [16]:

```
[Unit]
Description=Apache Tomcat Web Application Container
After=network.target
[Service]
Type=forking
Environment="JAVA_HOME=/usr/lib/jvm/jdk-17.0.12-oracle-x64/"
Environment="CATALINA_PID=/opt/tomcat/temp/tomcat.pid"
Environment="CATALINA_HOME=/opt/tomcat"
Environment="CATALINA_BASE=/opt/tomcat"
Environment="CATALINA_OPTS=-Xms512M -Xmx1024M -server -XX:+
   UseParallelGC -DTE_BASE=[[path]]"
Environment="JAVA_OPTS=-Djava.awt.headless=true -Djava.security.egd
   =file:/dev/./urandom"
ExecStart=/opt/tomcat/bin/startup.sh
ExecStop=/opt/tomcat/bin/shutdown.sh
User=tomcat
Group=tomcat
UMask=0007
RestartSec=10
Restart=always
[Install]
WantedBy=multi-user.target
```

[[path]] should be replaced with the actual path to the TE_BASE directory created in the previous step.

In addition, permissions for the TE_BASE folder need to be set to allow the tomcat user read and write access to it. For simplicity, all permissions were granted:

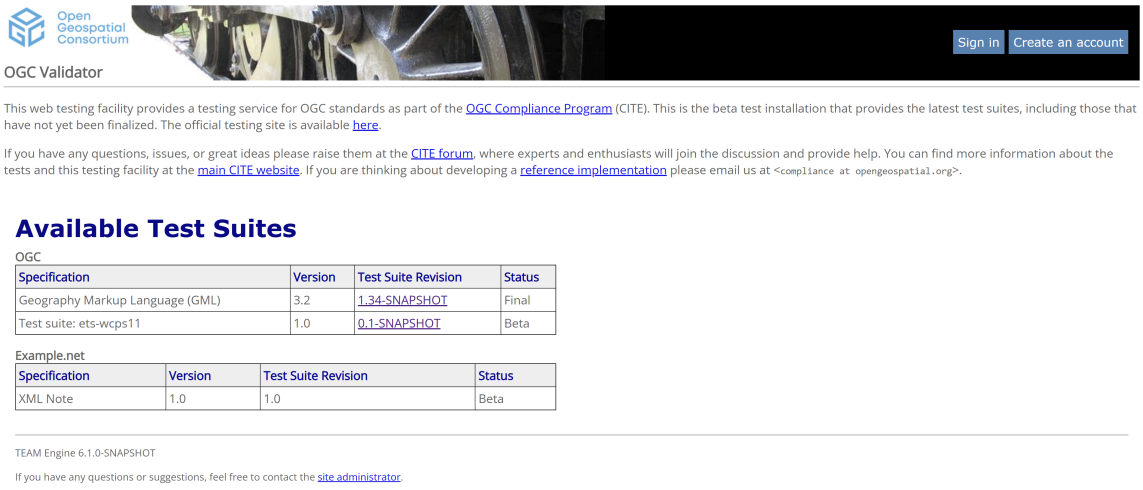
```
sudo chmod -R 777 ~/TE_BASE
```

7. Deploy TEAM Engine to Tomcat

Now TEAM Engine can be deployed to the Tomcat server by copying the generated .war file:

```
sudo cp ~/teamengine/teamengine-web/target/teamengine.war /opt/
    tomcat/webapps/
```

After this, teamengine should be accessible at localhost:8080/teamengine.



TEAM Engine web interface after deployment

8. Create an Account

An account needs to be created in order to run tests. The instructions on the web page can simply be followed.

A.2 Executable Test Suite Setup

The executable test suite was developed in Java using the TestNG framework. Initialization steps from the TEAM Engine website [20] were followed to set up the executable test suite.

1. Generate the Template Codebase

OGC provides a maven archetype, which was used to generate the initial codebase with the right structure and TEAM Engine compatible build artifacts.

```
mvn archetype:generate -B \
-DarchetypeGroupId=org.opengis.cite \
-DarchetypeArtifactId=ets-archetype-testng \
-DarchetypeVersion=2.5 \
-Dets-code=abc10 \
-DartifactId=ets-abc10 \
-Dpackage=org.opengis.cite.abc10
```

"abc10" needs to be replaced with the desired code for the test suite, which represents the name and version of the standard to be tested. "wcps11" was used in this case, as the test suite is for WCPS version 1.1.

The project generated by the archetype contains useful utilities for making HTTP requests, parsing and processing responses, validating against schematron validators, and more. Where needed these utilities were extended and adapted to fit the needs of the test suite.

2. Add the Desired Tests

The generated project is actually already complete in the sense that it is a valid runnable test suite. Therefore, it is convenient to follow the structure of the existing tests to modify the project. Detailed instructions about various aspects of developing an executable test suite can be found in [20].

3. Run the Test Suite Locally

The project can be built using maven:

```
mvn package
```

The test suite can then be run from the command line as a standalone JAR file, which is one of the build outputs, or via the TEAM Engine web interface as configured previously with Tomcat.

To run the test suite from the command line, the following command can be used:

```
java -jar ./target/ets-*-aio.jar \
  ./src/main/config/test-run-props.xml
```

The test-run-props.xml file should be populated with the arguments expected by the test suite. A sample file that can be used with the WCPS test suite looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties
    .dtd">
<properties version="1.0">
  <comment>Sample test run arguments (ets-wcps11)</comment>
  <entry key="iut">https://ows.rasdaman.org/rasdaman/ows</entry
     >
</properties>
```

This option can be used to conveniently run the test suite during development and iterate faster. However, since the test suite must run on TEAM Engine in production, it is also important to check that it runs on the instance set up in the previous section.

The first step to add the test suite to the TEAM Engine instance is to unzip the wrapper Conformance Testing Language (CTL) script to the right location:

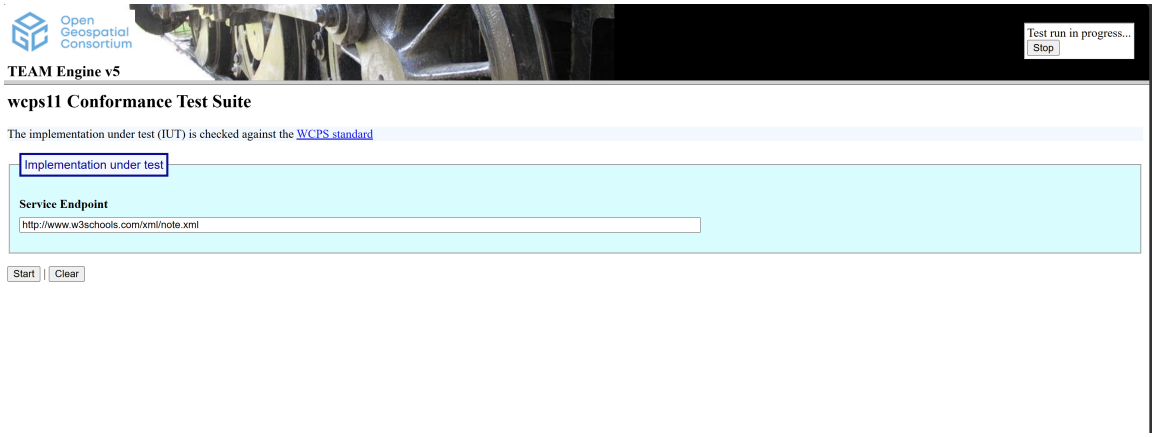
```
unzip ./target/ets-*-ctl.zip -d ~/TE_BASE/scripts/
```

Java dependencies must also be made available. Contrary to instructions given in [19], adding them to /TE_BASE/resources/lib/ doesn't work with the current version of TEAM Engine, unless the maven build output is modified to generate unpacked .class files instead of JAR files. A pull request addressing this configuration issue was opened on the TEAM Engine repository [11].

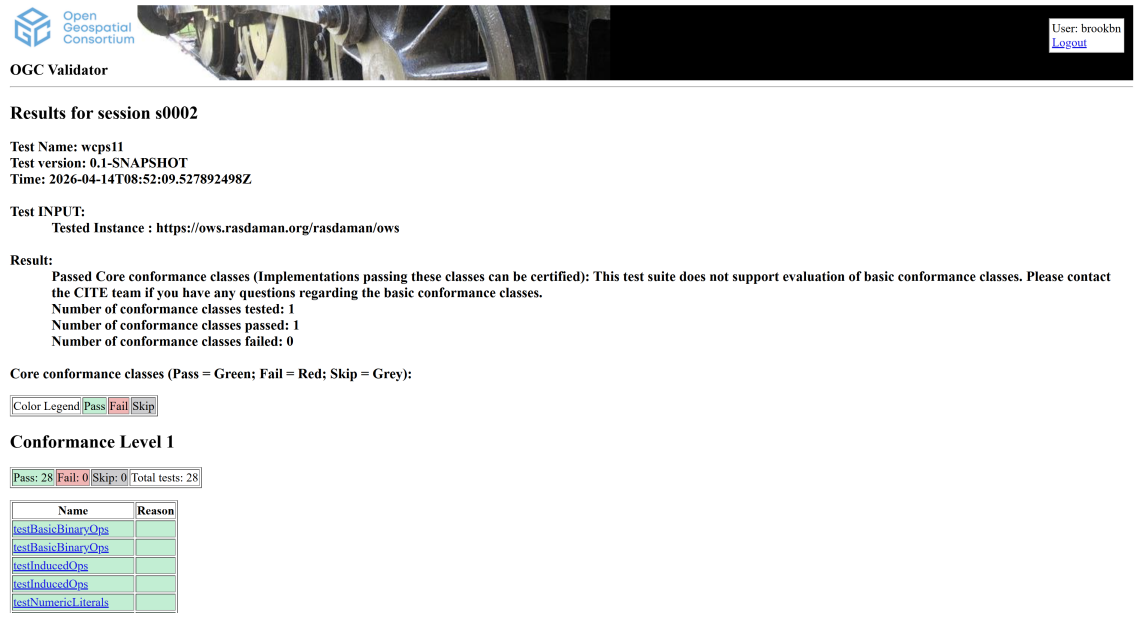
However, the JAR files can be added directly to the WEB-INF/lib/ directory of the deployed TEAM Engine web application:

```
sudo unzip ./target/ets-*-deps.zip -d \
  /opt/tomcat/webapps/teamengine/WEB-INF/lib/
```

After these steps, the test suite should be visible on the TEAM Engine web interface and can be executed by logging in with the account created previously.



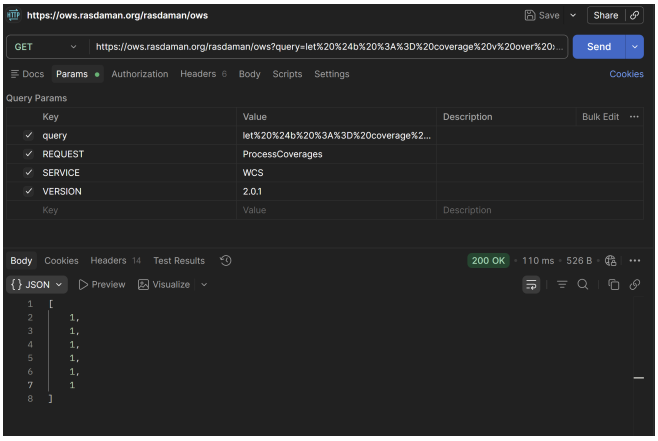
Test Run UI on TEAM Engine



Test Results on TEAM Engine

A.3 Reference Implementation

A reference implementation of WCPS by rasdaman, available at [24], was used in the development process to validate queries and expected outputs. Postman [23] was used to conveniently send queries and observe responses from the server. This helped a great deal to quickly check test queries and expected outputs while designing the test suite.



Postman interface for sending queries to the reference implementation

Notice that queries must be URL-encoded when using the GET/KVP binding.

B Test Queries

This appendix presents the test queries that make up the WCPS conformance test suite. The main goal is to have each test focus on an isolated language element. This means that tests using multiple features build on top of simpler ones and only introduce untested features individually. This makes sure that failures can be traced to specific language elements.

The for-clause (`for __cov__ in (<covName>)`) is prepended to every query before sending and is omitted here for brevity. `<covName>` is obtained from the test server using a *GetCapabilities* request as described in section 4. The comments with % signs are presented here only for illustration purposes and are not part of the language.

In addition to queries that are expected to succeed and produce a valid output, some queries that are expected to fail and produce an exception are also included. These are important for validating that the test subject correctly handles invalid queries and signals exceptions as specified in the standard.

Where appropriate, clauses from the Abstract Test Suite (ATS) defined in the standard have been referenced in the test suite in the descriptions of the test cases.

B.1 Some Basic Operations

```
return 1 + 3
```

```
return 10 + 1 + 17
```

```
return 11 > 32
```

```
return 11 < 32
```

B.2 xWcps Queries

B.2.1 xWcpsCoverageConstructorExpr

The *xWcpsCoverageConstructorExpr* is tested by constructing 1D-5D coverages. Things like accessing iterator variables in the range set definition, using a composite range type, and combining multiple CRSs are tested as well. The final query validates an exception that should occur when the number of dimensions doesn't match the chosen CRS.

```
% 1D inline syntax:
return encode(coverage newCov domain crs "OGC:Index1D"
  with x index (1:5) range type elem quantity int
  range set 1, "application/json")

% 1D, axis variable in range set:
return encode(coverage newCov domain crs "OGC:Index1D"
  with x index (1:5) range type elem quantity int
  range set x, "application/json")

% 1D, arithmetic in range set (constant and axis-based):
return encode(coverage newCov domain crs "OGC:Index1D"
  with x index (1:5) range type elem quantity int
  range set 1 + 1, "application/json")
return encode(coverage newCov domain crs "OGC:Index1D"
  with x index (1:5) range type elem quantity int
  range set x + 1, "application/json")
```

```
% 2D Index CRS:
return encode(coverage newCov
  domain crs "OGC:Index2D"
     with
    x index (1:5), y index
       (4:8)
  range type elem quantity int
  range set x + y, "
     application/json")
```

```
% 3D Index CRS:
return encode(coverage newCov
  domain crs "OGC:Index3D"
     with
    x index (0:2), y index
       (0:2),
    z index (0:2)
  range type elem quantity int
  range set x + y + z,
  "application/json")
```

```
% 4D Index CRS:
return encode(coverage newCov
  domain crs "OGC:Index4D"
     with
    x index (0:2), y index
       (0:2),
    z index (0:2), t index
       (0:2)
  range type elem quantity int
  range set x + y + z + t,
  "application/json")
```

```
% 5D Index CRS:
return encode(coverage newCov
  domain crs "OGC:Index5D"
     with
    x index (0:1), y index
       (0:1),
    z index (0:1), t index
       (0:1),
    s index (0:1)
  range type elem quantity int
  range set x + y + z + t + s,
  "application/json")
```

```
% Geographic CRS (EPSG:4326),
   regular:
return encode(coverage newCov
  domain crs "EPSG:4326" with
    lat regular (-2:2)
       resolution -0.5,
    long regular (-2:2)
       resolution 1
  range type elem quantity int
  range set 1, "application/
     json")
```

```
% Temporal CRS (OGC:AnsiDate),
    irregular:
return encode(coverage newCov
  domain crs "OGC:AnsiDate"
     with
    day irregular
       ("2024-01-01",
      "2025-01-01",
         "2026-01-01")
  range type elem quantity int
  range set 1, "application/
     json")
```

```
% Mixed spatiotemporal CRS (EPSG:4326+OGC:AnsiDate):
return encode(coverage newCov
  domain crs "EPSG:4326+OGC:AnsiDate" with
    lat regular (-2:2) resolution -0.5,
    long regular (-2:2) resolution 1,
    day irregular ("2024-01-01","2025-01-01","2026-01-01")
  range type elem quantity int range set 1, "application/json")
```

```
% 2D, two integer fields:
return encode(coverage newCov
  domain crs "OGC:Index2D"
     with
    x index (1:5), y index
       (4:8)
  range type
    field1 quantity int,
    field2 quantity int
  range set {field1: 1, field2
     : 2},
  "application/json")
```

```
% 3D, axis expressions per
    field:
return encode(coverage newCov
  domain crs "OGC:Index3D"
     with
    x index (0:4), y index
       (0:4),
    z index (0:4)
  range type
    field1 quantity int,
    field2 quantity int
  range set {field1: x+y,
     field2: y+z},
  "application/json")
```

```
% CRS dimension mismatch (invalid -- must signal an error):
return encode(coverage newCov domain crs "OGC:Index1D"
  with x index (1:5), y index (4:8)
  range type elem quantity int range set x + y,
  "application/json")
```

B.2.2 letExpr

```
% Scalar constant:
let $a := 2 return $a

% Two scalar constants:
let $a := 2, $b := 10
return $a + $b
```

```
% Coverage-valued constant:
let $a := (coverage newCov
  domain crs "OGC:Index1D"
     with
    x index (1:5)
  range type elem quantity int
  range set 1)
return encode($a, "application
   /json")
```

```
% Constant used in coverage
    constructor:
let $val := 1
return encode(coverage newCov
  domain crs "OGC:Index1D"
     with
    x index (1:5)
  range type elem quantity int
  range set $val, "application
     /json")
```

```
% Error: duplicate variable
    name:
let $a := 2, $a := 4 return $a
```

B.2.3 switchExpr

```
return (switch
  case 1 < 0 return 4
  default return 2)
```

```
return (switch
  case 1 < 0 return 4
  case 1 > 0 return 5
  default return 2)
```

```
let $a := (coverage newCov
  domain crs "OGC:Index3D"
     with
    x index (0:3), y index
       (0:3),
    z index (0:3)
  range type elem quantity int
  range set x + y + z)
return encode(switch
  case $a > 5 return 6
  case $a > 3 return 4
  case $a > 1 return 2
  default return 0,
  "application/json")
```

```
let $a := (coverage newCov
  domain crs "OGC:Index2D"
     with
    x index (1:5), y index
       (4:8)
  range type elem quantity int
  range set x + y),
  $b := ($a + 100)
return encode(switch
  case $a > 8 return $a
  default return $b,
  "application/json")
```

B.3 Unary Induced Operations

All but the last two queries construct a 1D coverage `newCov` with `x index (1:5)` and apply a unary operation. The last two queries test the field selection and range type casting induced operations.

```
return encode(+$a, "
    application/json")
  % range set -2
return encode(-$a, "
    application/json")
  % range set 1
return encode(abs($a), "
    application/json")
  % range set -3
return encode(sqrt($a), "
    application/json")
  % range set 9
return encode(sin($a), "
    application/json")
  % range set 1
return encode(cos($a), "
    application/json")
  % range set 1
return encode(tan($a), "
    application/json")
  % range set 1
```

```
return encode(sinh($a), "
    application/json")
  % range set 1
return encode(cosh($a), "
    application/json")
  % range set 1
return encode(tanh($a), "
    application/json")
  % range set 1
return encode(arcsin($a), "
    application/json")
  % range set 1
return encode(arccos($a), "
    application/json")
  % range set 1
return encode(arctan($a), "
    application/json")
  % range set 1
return encode(exp($a), "
    application/json")
  % range set 1
```

```
return encode(log($a), "
    application/json")
  % range set 2
return encode(ln($a), "
    application/json")
  % range set 2
```

```
return encode(pow($a, 3.5), "
    application/json")
  % range set 2
return encode(not($a), "
    application/json")
  % range set 13
```

```
let $a := (coverage newCov
  domain crs "OGC:Index2D"
     with
    x index (1:5),
    y index (4:8)
  range type
    field1 quantity int,
    field2 quantity int
  range set
    {field1: 1, field2: 2})
return encode($a.field1, "
   application/json")
```

```
let $a := (coverage newCov
  domain crs "OGC:Index2D"
     with
    x index (0:1),
    y index (0:2)
  range type
    elem quantity float
  range set
    2.44)
return encode((int)$a, "
   application/json")
```

B.4 Binary Induced Operations

B.4.1 Scalar with Coverage

Each query constructs a 1D coverage `newCov` with domain `x index (1:5)` and a constant range set value, then applies a binary operation between a scalar and the coverage. Instead of testing all operators twice, once with the scalar in the first position and then in the second position, scalars appear first in some queries and second in others.

```
let $a := (coverage newCov
  domain crs "OGC:Index1D"
     with
    x index (1:5)
  range type elem quantity int
  range set 2)
return encode($a + 3, "
   application/json")
```

```
let $a := (coverage newCov
  domain crs "OGC:Index1D"
     with
    x index (1:5)
  range type elem quantity int
  range set 2)
return encode(3 * $a, "
   application/json")
```

```
let $a := (coverage newCov
  domain crs "OGC:Index1D"
     with
    x index (1:5)
  range type elem quantity int
  range set 10)
return encode($a / 2, "
   application/json")
```

```
let $a := (coverage newCov
  domain crs "OGC:Index1D"
     with
    x index (1:5)
  range type elem quantity int
  range set 2)
return encode(2 = $a, "
   application/json")
```

```
let $a := (coverage newCov
  domain crs "OGC:Index1D"
     with
    x index (1:5)
  range type elem quantity int
  range set 2)
return encode($a != 2, "
   application/json")
```

```
let $a := (coverage newCov
  domain crs "OGC:Index1D"
     with
    x index (1:5)
  range type elem quantity int
  range set 2)
return encode(3 > $a, "
   application/json")
```

```
let $a := (coverage newCov
  domain crs "OGC:Index1D"
     with
    x index (1:5)
  range type elem quantity int
  range set 4)
return encode($a < 3, "
   application/json")
```

```
let $a := (coverage newCov
  domain crs "OGC:Index1D"
     with
    x index (1:5)
  range type elem quantity int
  range set 2)
return encode($a <= 2, "
   application/json")
```

```
let $a := (coverage newCov
  domain crs "OGC:Index1D"
     with
    x index (1:5)
  range type elem quantity int
  range set 1)
return encode(true and ($a =
   1),
  "application/json")
```

```
let $a := (coverage newCov
  domain crs "OGC:Index1D"
     with
    x index (1:5)
  range type elem quantity int
  range set 2)
return encode(1 overlay $a,
  "application/json")
```

```
let $a := (coverage newCov
  domain crs "OGC:Index1D"
     with
    x index (1:5)
  range type elem quantity int
  range set 2)
return encode(1 >= $a, "
   application/json")
```

```
let $a := (coverage newCov
  domain crs "OGC:Index1D"
     with
    x index (1:5)
  range type elem quantity int
  range set 1)
return encode(($a = 1) xor
   true,
  "application/json")
```

```
let $a := (coverage newCov
  domain crs "OGC:Index1D"
     with
    x index (1:5)
  range type elem quantity int
  range set 2)
return encode(false or ($a =
   1),
  "application/json")
```

```
let $a := (coverage newCov
  domain crs "OGC:Index1D"
     with
    x index (1:5)
  range type elem quantity int
  range set 0)
return encode($a overlay 3,
  "application/json")
```

B.4.2 Coverage with Coverage

Each query constructs two 3D coverages `newCov1` and `newCov2` with domain `x,y,z index (0:1)` and applies a binary operation between them. The range set values vary amongst the queries and are indicated.

```
let $a := (coverage newCov1
  domain crs "OGC:Index3D"
     with
    x index (0:1), y index
       (0:1),
    z index (0:1)
  range type
    elem quantity int
  range set
    2),
  $b := (coverage newCov2
  domain crs "OGC:Index3D"
     with
    x index (0:1), y index
       (0:1),
    z index (0:1)
  range type
    elem quantity int
  range set
    3)
return encode($a + $b,
      "application/json")
```

```
% same structure, varying op and
    values:
return encode($a * $b, "application/
    json")
  % $a range set 2, $b range set 2
return encode($a / $b, "application/
    json")
  % $a range set 9, $b range set 3
return encode($a = $b, "application/
    json")
  % $a range set 2, $b range set 2
return encode($a != $b, "application/
    json")
  % $a range set 2, $b range set 3
return encode($a > $b, "application/
    json")
  % $a range set 3, $b range set 2
return encode($a < $b, "application/
    json")
  % $a range set 5, $b range set 5
return encode($a >= $b, "application/
    json")
  % $a range set 2, $b range set 2
return encode($a <= $b, "application/
    json")
  % $a range set 2, $b range set 1
return encode(($a != 1)and($b = 1),"
    application/json")
  % $a range set 1, $b range set 1
return encode(($a != 1)or($b = 1),"
    application/json")
  % $a range set 1, $b range set 1
return encode(($a != 1)xor($b = 1),"
    application/json")
  % $a range set 1, $b range set 1
return encode($a overlay $b, "
    application/json")
  % $a range set 2, $b range set 3
```

B.5 generalCondenseExpr

```
return condense +
  over px x (1:5)
  using px
```

```
return condense *
  over px x (1:5)
  where px > 2
  using px
```

```
return condense max
  over px x (1:5), py y (1:5)
  where px != py
  using px + py
```

```
return condense min
  over px x (1:5), py y (1:5)
  where px != py
  using px + py
```

```
return condense or
  over px x (1:5), py y (1:5)
  where px != py
  using px > 3 and py > 3
```

```
return condense and
  over px x (1:5), py y (1:5)
  using px != py
```

B.6 reduceExpr

Each query constructs a 4D coverage with `x,y,z,t index (0:5)` and `range set x+y+z+t`.

```
return add($a)
return avg($a)
return min($a)
```

```
return max($a)
return count($a > 10)
return count($a)
```

C Spec and Implementation Issues

A few issues with the WCPS standard [5] and the rasdaman reference implementation were uncovered in the development of the test suite. These are discussed in this appendix.

C.1 File Uploads for Decode Operations

The xWcps *decodeCoverageExpr* specifies a way to decode a bytestream sent along with a query into a coverage that can be used within the query. The syntax is as follows:

```
decode(b)
decode(b, extraParams)
```

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

The specification works fine for ASCII encodings, which can be sent within the query or as extra query parameters to be referenced. However, the same doesn't hold for binary formats. This is because the WCPS Processing Extension [6] does not specify any protocol bindings that allow passing binary files as extra parameters. Indeed, none of the GET/KVP, XML/POST, and SOAP encodings support binary extra parameters.

Rasdaman does support sending queries as multipart form data, which can support file uploads. This is not currently standardized but could be considered in the future.

C.2 xWcps Language Grammar

The way the language grammar is currently written, xWcps features are part of neither the *xWcpsExpr* nor the *processCoveragesExpr*, which are the expressions defined to make up complete queries.

The test suite was made with a future corrigendum addressing this issue in mind and contains tests for various xWcps features.

Another issue is that the grammar requires let-clauses in *xWcpsExpr* to appear before for-clauses. This is the other way around in the rasdaman implementation. The tests currently follow the implementation's order, but this inconsistency should be resolved and the tests potentially updated.

C.3 *letExpr* Expression Isolation

The standard is not explicit about whether *letExpr* definitions should be isolated by effectively parenthesizing their expressions when replaced for their names. If the expected behavior is isolation, then this could, for example, lead to issues with operator precedence, where an expression containing a lower precedence operator is substituted into an expression containing a higher precedence operator.

letExpr definitions are all wrapped in parentheses in the test suite to avoid this ambiguity.

C.4 Overlay Semantics Mismatch

The semantics of the `overlay` binary operator in the rasdaman implementation does not match the one defined in the standard. The exact issue is that the roles of the two arguments are reversed in the implementation.

This difference should be reconciled from either the standard's side or the rasdaman implementation's side for compliance.

C.5 Boolean Scalar Output Encoding

While booleans are correctly encoded as *true* or *false* in coverage outputs from the rasdaman implementation, they are encoded as *t* or *f* as scalar outputs. This is not compliant with the standard and should be fixed in the implementation.