

OGC® DOCUMENT: 24-014

External identifier of this OGC® document: <http://www.opengis.net/doc/IS/SWE/3.0>



Open
Geospatial
Consortium

OGC SWE COMMON DATA MODEL ENCODING STANDARD

STANDARD
Implementation

APPROVED

Version: 3.0.0

Submission Date: 2025-03-19

Approval Date: 2025-06-02

Publication Date: 2025-07-16

Editor: Alexandre Robin

Notice: This document is an OGC Member approved international standard. This document is available on a royalty free, non-discriminatory basis. Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

License Agreement

Use of this document is subject to the license agreement at <https://www.ogc.org/license>

Suggested additions, changes and comments on this document are welcome and encouraged. Such suggestions may be submitted using the online change request form on OGC web site: <http://ogc.standardstracker.org/>

Copyright notice

Copyright © 2025 Open Geospatial Consortium

To obtain additional rights of use, visit <https://www.ogc.org/legal>

Note

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

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

CONTENTS

I. ABSTRACT	xi
II. KEYWORDS	xi
III. PREFACE	xii
IV. SECURITY CONSIDERATIONS	xiii
V. SUBMITTING ORGANIZATIONS	xiv
VI. SUBMITTERS	xiv
VII. FOREWORD	xv
1. SCOPE	2
2. CONFORMANCE	4
3. NORMATIVE REFERENCES	7
4. TERMS AND DEFINITIONS	10
6. CONVENTIONS	14
6.1. Identifiers	14
6.2. Abbreviated terms	14
6.3. UML Notation	15
7. REQUIREMENTS CLASS: CORE CONCEPTS (NORMATIVE CORE)	17
7.1. Introduction	17
7.2. Data Representation	18
7.3. Nature of Data	22
7.4. Data Quality	25
7.5. Data Structure	26
7.6. Data Encoding	28
8. UML CONCEPTUAL MODELS (NORMATIVE)	30
8.1. Package Dependencies	30
8.2. Requirements Class: Basic Types and Simple Components Packages	31
8.3. Requirements Class: Record Components Package	54
8.4. Requirements Class: Choice Components Package	59
8.5. Requirements Class: Block Components Package	61

8.6. Requirements Class: Geometry Components Package	69
8.7. Requirements Class: Simple Encodings Package	71
8.8. Requirements Class: Advanced Encodings Package	74
9. JSON IMPLEMENTATION (NORMATIVE)	78
9.1. Requirements Class: Basic Types and Simple Components JSON Schemas	78
9.2. Requirements Class: Record Components JSON Schema	92
9.3. Requirements Class: Choice Components JSON Schema	96
9.4. Requirements Class: Block Components JSON Schema	98
9.5. Requirements Class: Geometry Components JSON Schema	105
9.6. Requirements Class: Simple Encodings JSON Schema	107
9.7. Requirements Class: Advanced Encodings JSON Schema	109
10. DATA BLOCKS AND STREAMS ENCODING RULES (NORMATIVE)	116
10.1. Requirements Class: General Encoding Rules	116
10.2. Requirements Class: JSON Encoding Rules	119
10.3. Requirements Class: Text Encoding Rules	125
10.4. Requirements Class: Binary Encoding Rules	132
ANNEX A (NORMATIVE) CONFORMANCE CLASS ABSTRACT TEST SUITE	140
A.1. Core Conformance Classes	140
A.2. UML Conformance Classes	144
A.3. JSON Conformance Classes	160
A.4. Datastream Encoding Conformance Classes	171
ANNEX B (INFORMATIVE) EXAMPLES	182
B.1. Text Encoding Rules Examples	182
B.2. JSON Encoding Rules Examples	190
ANNEX C (INFORMATIVE) RELATIONSHIP WITH OTHER ISO MODELS	197
C.1. Feature model	197
C.2. Coverage model	197
ANNEX D (INFORMATIVE) REVISION HISTORY	199
BIBLIOGRAPHY	202

LIST OF TABLES

Table 1 – Requirements Classes	4
Table 2 – Allowed Binary Data Types	111
Table 3 – Simple Component to JSON Value Types Mapping	121
Table 4 – Range Component to JSON Mapping	122

LIST OF FIGURES

Figure 2 – Internal Package Dependencies	30
Figure 3 – External Package Dependencies	31
Figure 4 – Scalar Data Components	33
Figure 5 – Range Data Components	33
Figure 6 – Basic types for pairs of scalar types	35
Figure 7 – AbstractDataComponent Class	35
Figure 8 – AbstractSimpleComponent Class	37
Figure 9 – Boolean Class	40
Figure 10 – Text Class	40
Figure 11 – Category Class	41
Figure 12 – Count Class	42
Figure 13 – Quantity Class	43
Figure 14 – Time Class	44
Figure 15 – CategoryRange Class	47
Figure 16 – CountRange Class	48
Figure 17 – QuantityRange Class	49
Figure 18 – TimeRange Class	49
Figure 19 – Quality Union	50
Figure 20 – NilValues Class	51
Figure 21 – AllowedTokens Class	52
Figure 22 – AllowedValues Class	52
Figure 23 – AllowedTimes Class	53
Figure 24 – Simple Component Unions	54
Figure 25 – Record Data Components	55
Figure 26 – DataRecord Class	56
Figure 27 – Vector Class	57
Figure 28 – DataChoice Class	60
Figure 29 – Array Components	62
Figure 30 – DataArray Class	63
Figure 31 – Matrix Class	67
Figure 32 – DataStream Class	68
Figure 33 – Geometry Class	70
Figure 34 – Simple Encodings	72
Figure 35 – TextEncoding Class	73
Figure 36 – BinaryEncoding Class	75

LIST OF RECOMMENDATIONS

- REQUIREMENTS CLASS 1: CORE CONCEPTS 17
- REQUIREMENTS CLASS 2: SIMPLE COMPONENTS UML PACKAGE 31
- REQUIREMENTS CLASS 3: RECORD COMPONENTS UML PACKAGE54
- REQUIREMENTS CLASS 4: CHOICE COMPONENTS UML PACKAGE59
- REQUIREMENTS CLASS 5: BLOCK COMPONENTS UML PACKAGE61
- REQUIREMENTS CLASS 6: GEOMETRY COMPONENTS UML PACKAGE69
- REQUIREMENTS CLASS 7: SIMPLE ENCODINGS UML PACKAGE 71
- REQUIREMENTS CLASS 8: ADVANCED ENCODINGS UML PACKAGE 74
- REQUIREMENTS CLASS 9: BASIC TYPES AND SIMPLE COMPONENTS JSON SCHEMAS
.....78
- REQUIREMENTS CLASS 10: RECORD COMPONENTS JSON SCHEMA 92
- REQUIREMENTS CLASS 11: CHOICE COMPONENTS JSON SCHEMA96
- REQUIREMENTS CLASS 12: BLOCK COMPONENTS JSON SCHEMA98
- REQUIREMENTS CLASS 13: GEOMETRY COMPONENTS JSON SCHEMA 105
- REQUIREMENTS CLASS 14: SIMPLE ENCODINGS JSON SCHEMA 107
- REQUIREMENTS CLASS 15: ADVANCED ENCODINGS JSON SCHEMA109
- REQUIREMENTS CLASS 16: GENERAL ENCODING RULES116
- REQUIREMENTS CLASS 17: JSON ENCODING RULES119
- REQUIREMENTS CLASS 18: TEXT ENCODING RULES 125
- REQUIREMENTS CLASS 19: BINARY ENCODING RULES 132
- REQUIREMENT 1 18
- REQUIREMENT 2 19
- REQUIREMENT 3 19
- REQUIREMENT 4 20
- REQUIREMENT 5 21
- REQUIREMENT 6 21
- REQUIREMENT 7 23
- REQUIREMENT 8 23
- REQUIREMENT 9 24
- REQUIREMENT 10 24
- REQUIREMENT 1126

REQUIREMENT 12	27
REQUIREMENT 13	28
REQUIREMENT 14	33
REQUIREMENT 15	34
REQUIREMENT 16	34
REQUIREMENT 17	37
REQUIREMENT 18	37
REQUIREMENT 19	38
REQUIREMENT 20	38
REQUIREMENT 21	39
REQUIREMENT 22	39
REQUIREMENT 23	41
REQUIREMENT 24	42
REQUIREMENT 25	42
REQUIREMENT 26	44
REQUIREMENT 27	45
REQUIREMENT 28	45
REQUIREMENT 29	47
REQUIREMENT 30	47
REQUIREMENT 31	48
REQUIREMENT 32	49
REQUIREMENT 33	51
REQUIREMENT 34	51
REQUIREMENT 35	52
REQUIREMENT 36	55
REQUIREMENT 37	56
REQUIREMENT 38	57
REQUIREMENT 39	58
REQUIREMENT 40	58
REQUIREMENT 41	58
REQUIREMENT 42	60
REQUIREMENT 43	61
REQUIREMENT 44	62

REQUIREMENT 45	63
REQUIREMENT 46	64
REQUIREMENT 47	67
REQUIREMENT 48	69
REQUIREMENT 49	69
REQUIREMENT 50	70
REQUIREMENT 51	71
REQUIREMENT 52	72
REQUIREMENT 53	74
REQUIREMENT 54	79
REQUIREMENT 55	79
REQUIREMENT 56	80
REQUIREMENT 57	81
REQUIREMENT 58	81
REQUIREMENT 59	84
REQUIREMENT 60	86
REQUIREMENT 61	92
REQUIREMENT 62	96
REQUIREMENT 63	98
REQUIREMENT 64	103
REQUIREMENT 65	103
REQUIREMENT 66	105
REQUIREMENT 67	107
REQUIREMENT 68	108
REQUIREMENT 69	108
REQUIREMENT 70	109
REQUIREMENT 71	110
REQUIREMENT 72	111
REQUIREMENT 73	111
REQUIREMENT 74	111
REQUIREMENT 75	113
REQUIREMENT 76	113
REQUIREMENT 77	114

REQUIREMENT 78	117
REQUIREMENT 79	118
REQUIREMENT 80	119
REQUIREMENT 81	119
REQUIREMENT 82	120
REQUIREMENT 83	120
REQUIREMENT 84	121
REQUIREMENT 85	122
REQUIREMENT 86	123
REQUIREMENT 87	124
REQUIREMENT 88	124
REQUIREMENT 89	125
REQUIREMENT 90	126
REQUIREMENT 91	126
REQUIREMENT 92	128
REQUIREMENT 93	129
REQUIREMENT 94	131
REQUIREMENT 95	132
REQUIREMENT 96	133
REQUIREMENT 97	134
REQUIREMENT 98	135
REQUIREMENT 99	135
REQUIREMENT 100	137
RECOMMENDATION 1	84
CONFORMANCE CLASS A.1	140
CONFORMANCE CLASS A.2: BASIC TYPES AND SIMPLE COMPONENTS UML PACKAGES	144
CONFORMANCE CLASS A.3: RECORD COMPONENTS UML PACKAGE	152
CONFORMANCE CLASS A.4: CHOICE COMPONENTS UML PACKAGE	155
CONFORMANCE CLASS A.5: BLOCK COMPONENTS UML PACKAGE	156
CONFORMANCE CLASS A.6: GEOMETRY COMPONENTS UML PACKAGE	158
CONFORMANCE CLASS A.7: SIMPLE ENCODINGS UML PACKAGE	159
CONFORMANCE CLASS A.8: ADVANCED ENCODINGS UML PACKAGE	160

CONFORMANCE CLASS A.9: BASIC TYPES AND SIMPLE COMPONENTS JSON SCHEMAS	160
CONFORMANCE CLASS A.10: RECORD COMPONENTS JSON SCHEMA	163
CONFORMANCE CLASS A.11: CHOICE COMPONENTS JSON SCHEMA	164
CONFORMANCE CLASS A.12: BLOCK COMPONENTS JSON SCHEMA	164
CONFORMANCE CLASS A.13: GEOMETRY COMPONENTS JSON SCHEMA	166
CONFORMANCE CLASS A.14: SIMPLE ENCODINGS JSON SCHEMA	166
CONFORMANCE CLASS A.15: ADVANCED ENCODINGS JSON SCHEMA	168
CONFORMANCE CLASS A.16: GENERAL ENCODING RULES	171
CONFORMANCE CLASS A.17: JSON ENCODING RULES	173
CONFORMANCE CLASS A.18: TEXT ENCODING RULES	176
CONFORMANCE CLASS A.19: BINARY ENCODING RULES	178



ABSTRACT

The primary focus of the SWE Common Data Model is to define and package data in a self-describing and semantically enabled way. The main objective is to achieve interoperability, first at the syntactic level, and later at the semantic level (by using ontologies and probably semantic mediation) so that (sensor) data can be better understood by machines, processed automatically in complex workflows and easily shared between intelligent nodes.

This standard is one of several implementation standards produced under OGC's Connected Systems activity, and is a revision of content that was previously developed in the context of the Sensor Web Enablement initiative. These common data models are intended to be used in various OGC standards, and in particular, SensorML and OGC API — Connected Systems.



KEYWORDS

The following are keywords to be used by search engines and document catalogues.

ogcdoc, OGC document, html, SWE, sensor, sensorweb, connected systems, json, encoding, observation, command, tasking, property



PREFACE

The OGC SWE Common Data Model Encoding Standard is the result of work done by the Connected Systems Working Group of the OGC, with the aim of modernizing the Sensor Web Enablement (SWE) suite of Standards.

To increase the brevity and readability of this Standard, many OGC document titles are shortened and/or abbreviated. Therefore, in the context of this document, the following phrases are defined:

- “this Standard” shall be interpreted as equivalent to “OGC SWE Common Data Model Encoding Standard;” and
- “SensorML” or “SensorML Standard” shall be interpreted as equivalent to “OGC SensorML Encoding Standard.”



SECURITY CONSIDERATIONS

No security considerations have been made for this Standard.

V

SUBMITTING ORGANIZATIONS

The following organizations submitted this Document to the Open Geospatial Consortium (OGC):

- GeoRobotix, Inc.
- Botts Innovative Research, Inc.
- Cesium GS, Inc.
- 52°North Spatial Information Research GmbH
- Pelagis Data Solutions
- National Geospatial-Intelligence Agency (NGA)

VI

SUBMITTERS

All questions regarding this submission should be directed to the editor or the submitters:

NAME	AFFILIATION
Alex Robin (Editor)	GeoRobotix, Inc.
Christian Autermann	52° North Spatial Information Research GmbH
Chuck Heazel	Heazeltech
Mike Botts	Botts Innovative Research, Inc.

Additional contributors to this Standard include the following:

NAME	AFFILIATION
Arne Broering	52° North Initiative
Barry Reff	US Department of Homeland Security (DHS)
Ingo Simonis	iGSI

NAME	AFFILIATION
Johannes Echterhoff	iGSI
John Herring	Oracle USA
Luis Bermudez	SURA
Nick Garay	Botts Innovative Research, Inc.
Peter Taylor	CSIRO

VII

FOREWORD

This document supersedes version 2.0 of the OGC® SWE Common Data Model Encoding Standard (OGC 08-094r2).

The following changes have been made:

- Addition of the JSON encodings and schemas;
- Addition of the Geometry data component (modeled on OGC simple feature geometries);
- Removal of the XML encodings; and
- Technical revision and improved explanations in various clauses.

This release is not fully backward compatible with version 2.0, this concerns especially the removal of the XML encodings.



1

SCOPE

This standard defines low level data models for exchanging sensor related data between nodes of the OGC® Connected Systems framework (previously Sensor Web Enablement (SWE) framework). These models allow applications and/or servers to structure, encode and transmit sensor datasets in a self-describing and semantically enabled way.

More precisely, the SWE Common Data Model is used to define the representation, nature, structure and encoding of sensor related data. These four pieces of information, essential for fully describing a data stream, are further defined in Clause 7.

The SWE Common Data Model is intended to be used for describing static data (files) as well as dynamically generated datasets (on the fly processing), data subsets, process and web service inputs and outputs, and real time streaming data and commands. All categories of sensor observations are in scope ranging from simple in-situ temperature data to satellite imagery and full motion video.

This standard defines JSON encodings for the dataset/datastream description, while the data itself can be encoded in JSON, text or binary form. This standard is used by other existing OGC standards such as the Sensor Model Language (SensorML) and OGC API — Connected Systems. One goal of the SWE Common Data Model is to maintain the functionality and consistency between these related standards.



2

CONFORMANCE

This Standard was written to be compliant with the OGC Specification Model – A Standard for Modular Specification (OGC 08-131r3). Extensions of this Standard shall themselves be conformant to the OGC Specification Model.

This Standard defines the following requirements classes and standardization targets:

Table 1 — Requirements Classes

REQUIREMENTS CLASS	STANDARDIZATION TARGET
Core	
Clause 7, Requirements Class: Core Concepts (normative core)	Derived Models and Software Implementations
UML Models	
Clause 8.2, Requirements Class: Basic Types and Simple Components Packages	
Clause 8.3, Requirements Class: Record Components Package	
Clause 8.4, Requirements Class: Choice Components Package	Software Implementation or Encoding of the Conceptual Models
Clause 8.5, Requirements Class: Block Components Package	
Clause 8.6, Requirements Class: Geometry Components Package	
Clause 8.7, Requirements Class: Simple Encodings Package	
Clause 8.8, Requirements Class: Advanced Encodings Package	
JSON Encodings	
Clause 9.1, Requirements Class: Basic Types and Simple Components JSON Schemas	JSON Document
Clause 9.2, Requirements Class: Record Components JSON Schema	

REQUIREMENTS CLASS	STANDARDIZATION TARGET
Clause 9.3, Requirements Class: Choice Components JSON Schema	
Clause 9.4, Requirements Class: Block Components JSON Schema	
Clause 9.5, Requirements Class: Geometry Components JSON Schema	
Clause 9.6, Requirements Class: Simple Encodings JSON Schema	
Clause 9.7, Requirements Class: Advanced Encodings JSON Schema	
Datastream Encoding Rules	Encoded Values Instance
Clause 10.3, Requirements Class: Text Encoding Rules	
Clause 10.4, Requirements Class: Binary Encoding Rules	
Clause 10.2, Requirements Class: JSON Encoding Rules	

Different types of implementations can seek conformance with this OGC® Standard.

- An implementation that defines a new data model shall at least conform with the core requirements class.
- An encoding of the conceptual models (e.g., a protobuf encoding) shall implement at least one of the requirements classes listed in the “UML Models” section of the table.
- An implementation that produces or consumes SWE Common data components encoded in JSON shall implement at least one of the requirements classes listed in the “JSON Encodings” section of the table.
- An implementation that produces or consumes datastreams encoded according to a schema defined using SWE Common components shall implement at least one of the requirements classes listed in the “Datastream Encoding Rules” section of the table.

The conformance classes corresponding to these requirements classes are presented in Annex A (normative). Conformance with this Standard shall be checked using all the relevant tests specified in Annex A. The framework, concepts, and methodology for testing, and the criteria to be achieved to claim conformance are specified in the OGC Compliance Testing Policies and Procedures and the OGC Compliance Testing web site.



3

NORMATIVE REFERENCES

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

Policy SWG: OGC 08-131r3, *The Specification Model – Standard for Modular specifications*. Open Geospatial Consortium (2009). https://portal.ogc.org/files/?artifact_id=34762&version=2.

John Herring: OGC 06-103r4, *OpenGIS Implementation Specification for Geographic information – Simple feature access – Part 1: Common architecture*. Open Geospatial Consortium (2011). <http://www.opengis.net/doc/is/sfa/1.2.1>.

Linda van den Brink, Clemens Portele, Panagiotis (Peter) A. Vretanos: OGC 10-100r3, *Geography Markup Language (GML) simple features profile (with Corrigendum)*. Open Geospatial Consortium (2011). https://portal.ogc.org/files/?artifact_id=42729.

ISO: ISO 8601:2019, *Date and time – Representations for information interchange – Part 1: Basic rules*. International Organization for Standardization, Geneva (2019). <https://www.iso.org/standard/70907.html>. ISO (2019).

ISO: ISO 8601:2019, *Date and time – Representations for information interchange – Part 2: Extensions*. International Organization for Standardization, Geneva (2019). <https://www.iso.org/standard/70908.html>. ISO (2019).

ISO/IEC: ISO/IEC 11404:2007, *Information technology – General-Purpose Datatypes (GPD)*. International Organization for Standardization, International Electrotechnical Commission, Geneva (2007). <https://www.iso.org/standard/39479.html>.

ISO: ISO 19101-1:2014, *Geographic information – Reference model – Part 1: Fundamentals*. International Organization for Standardization, Geneva (2014). <https://www.iso.org/standard/59164.html>.

ISO: ISO 19103:2005, *Conceptual Schema Language*. ISO (2005).

ISO: ISO 19107:2003, *Geographic information – Spatial schema*. International Organization for Standardization, Geneva (2003). <https://www.iso.org/standard/26012.html>.

ISO: ISO 19108:2002, *Geographic information – Temporal schema*. International Organization for Standardization, Geneva (2002). <https://www.iso.org/standard/26013.html>.

ISO: ISO 19111:2007, *Geographic information – Spatial referencing by coordinates*. International Organization for Standardization, Geneva (2007). <https://www.iso.org/standard/41126.html>.

Unified Code for Units of Measure (UCUM), Version 2.1, November 2017, <https://ucum.org/ucum>

- Unicode Technical Std #18, Unicode Regular Expressions, Version 19, Oct. 2016
- The Unicode Standard, Version 10.0, December 2017
- T. Bray (ed.): IETF RFC 8259, *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC Publisher (2017). <https://www.rfc-editor.org/info/rfc8259>.
- H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, T. Schaub: IETF RFC 7946, *The GeoJSON Format*. RFC Publisher (2016). <https://www.rfc-editor.org/info/rfc7946>.
- M. Nottingham: IETF RFC 8288, *Web Linking*. RFC Publisher (2017). <https://www.rfc-editor.org/info/rfc8288>.
- JSON Schema Validation: A Vocabulary for Structural Validation of JSON, Version 2020-12, <https://json-schema.org/draft/2020-12/json-schema-validation.html>
- N. Freed, N. Borenstein: IETF RFC 2045, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. RFC Publisher (1996). <https://www.rfc-editor.org/info/rfc2045>.
- P. Overell: IETF RFC 5234, *Augmented BNF for Syntax Specifications: ABNF*. RFC Publisher (2008). <https://www.rfc-editor.org/info/rfc5234>.
- IEEE: IEEE 754™-2008, *IEEE Standard for Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers (2013). <https://ieeexplore.ieee.org/document/4610935>.
- L. Masinter: IETF RFC 2397, *The "data" URL scheme*. RFC Publisher (1998). <https://www.rfc-editor.org/info/rfc2397>.



4

TERMS AND DEFINITIONS

This document uses the terms defined in OGC Policy Directive 49, which is based on the ISO/IEC Directives, Part 2, Rules for the structure and drafting of International Standards. In particular, the word “shall” (not “must”) is the verb form used to indicate a requirement to be strictly followed to conform to this document and OGC documents do not use the equivalent phrases in the ISO/IEC Directives, Part 2.

This document also uses terms defined in the OGC Standard for Modular specifications (OGC 08-131r3), also known as the ‘ModSpec’. The definitions of terms such as standard, specification, requirement, and conformance test are provided in the ModSpec.

For the purposes of this document, the following additional terms and definitions apply.

4.1. Data Component

Element of sensor data definition corresponding to an atomic or aggregate data type

Note 1 to entry: A data component is a part of the overall dataset definition. The dataset structure can then be seen as a hierarchical tree of data components.

4.2. Feature

Abstraction of real-world phenomena

[SOURCE: ISO 19101-1:2014, definition 4.1.11]

4.3. Observation

Act of measuring or otherwise determining the value of a property

[SOURCE: ISO 19156:2011, definition 4.11]

4.4. Observation Procedure

Method, algorithm or instrument, or system of these, which may be used in making an observation

Note: In the context of the sensor web, an observation procedure is often composed of one or more sensors that transform a real world phenomenon into digital information, plus additional processing steps.

[SOURCE: ISO 19156:2011, definition 4.12]

4.5. Property

Facet or attribute of an object referenced by a name Example : Abby's car has the color red, where "color red" is a property of the car instance

[SOURCE: ISO 19143:2010]

4.6. Sensor

An entity capable of observing a phenomenon and returning an observed value. Type of observation procedure that provides the estimated value of an observed property at its output.

Note 1 to entry: A sensor uses a combination of physical, chemical or biological means in order to estimate the underlying observed property. At the end of the measuring chain electronic devices often produce signals to be processed.

4.7. Sensor Data

List of digital values produced by a sensor that represents estimated values of one or more observed properties of one or more features.

Note 1 to entry: Sensor data is usually available in the form of data streams or computer files.

4.8. Sensor-Related Data

List of digital values produced by a sensor that contains ancillary information that is not directly related to the value of observed properties

EXAMPLE: sensor status, quality of measure, quality of service, battery life, etc. Such data can be sent in the same data stream with measured values and when measured is sometimes indistinguishable from sensor data.



6

CONVENTIONS

6.1. Identifiers

The normative provisions in this standard are denoted by the URI

<http://www.opengis.net/spec/SWE/3.0>

All requirements and conformance tests that appear in this document are denoted by partial URIs which are relative to this base.

6.2. Abbreviated terms

In this document the following abbreviations and acronyms are used or introduced:

- CRS: Coordinate Reference System
- CSML: Climate Science Modeling Language
- GPS: Global Positioning System
- ISO International Organization for Standardization
- MISB Motion Imagery Standards Board
- OGC Open Geospatial Consortium
- SAS Sensor Alert Service
- SensorML Sensor Model Language
- SI Système International (International System of Units)
- SOS Sensor Observation Service
- SPS Sensor Planning Service
- SWE Sensor Web Enablement
- TAI Temps Atomique International (International Atomic Time)
- UML Unified Modeling Language
- UTC Coordinated Universal Time

- XML eXtensible Markup Language
- 1D One Dimensional
- 2D Two Dimensional
- 3D Three Dimensional

6.3. UML Notation

The diagrams that appear in this standard are presented using the Unified Modeling Language (UML) static structure diagram. The UML notations used in this standard are described in the diagram below.

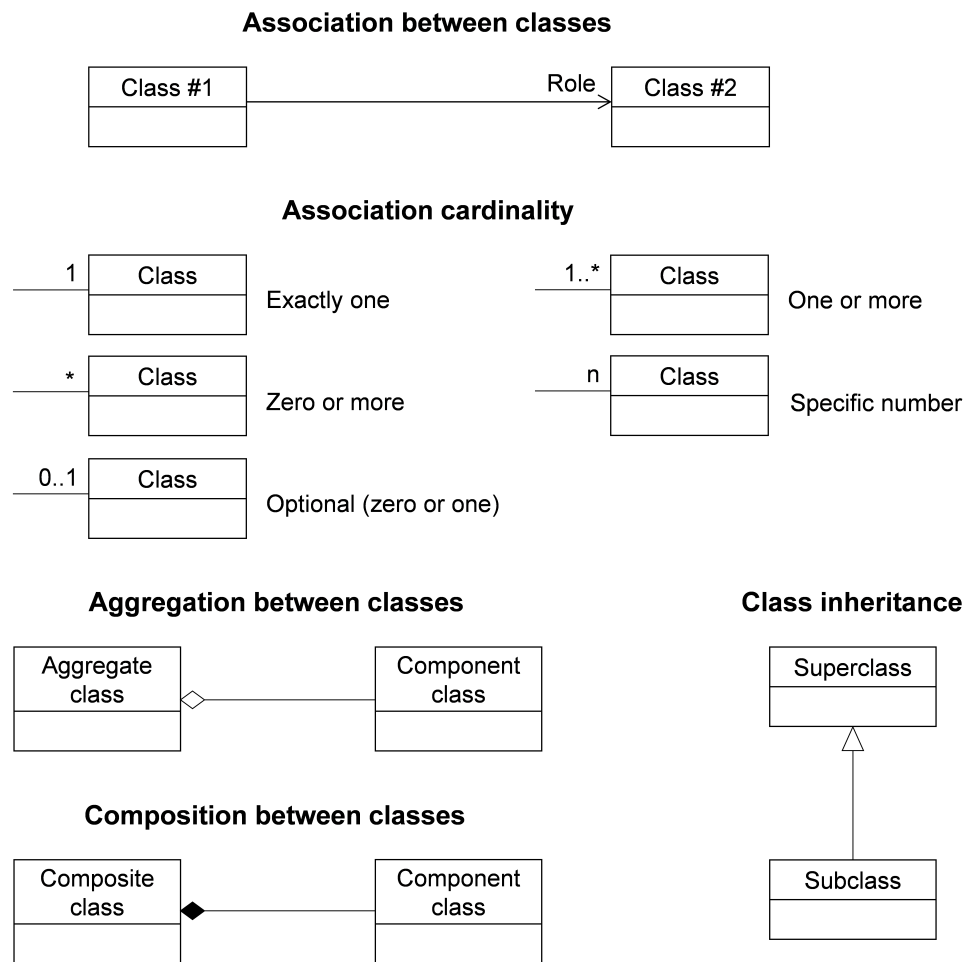


Figure 1



7

REQUIREMENTS CLASS: CORE CONCEPTS (NORMATIVE CORE)

REQUIREMENTS CLASS: CORE CONCEPTS (NORMATIVE CORE)

REQUIREMENTS CLASS 1: CORE CONCEPTS	
IDENTIFIER	/req/core
TARGET TYPE	Derived Models and Software Implementations
CONFORMANCE CLASS	Conformance class A.1: /conf/core
NORMATIVE STATEMENTS	Requirement 1: /req/core/core-concepts-used Requirement 2: /req/core/boolean-rep-valid Requirement 3: /req/core/categorical-rep-valid Requirement 4: /req/core/numerical-rep-valid Requirement 5: /req/core/countable-rep-valid Requirement 6: /req/core/textual-rep-valid Requirement 7: /req/core/semantics-defined Requirement 8: /req/core/semantics-resolvable Requirement 9: /req/core/temporal-frame-defined Requirement 10: /req/core/spatial-frame-defined Requirement 11: /req/core/nil-reasons-defined Requirement 12: /req/core/aggregates-model-valid Requirement 13: /req/core/encoding-method-valid

7.1. Introduction

The generic SWE Common data model defined by this standard aims at providing verbose information to robustly describe sensor related datasets. This Standard defines Sensor Data as data resulting from the observation of properties of virtual or real world objects (or features) by any type of Observation Procedure (See the Observation and Measurements specification OGC 07-022r1 for a more complete description of the observation model used in SWE).

Sensor related datasets however are not limited to sensor observation values, but can also include auxiliary information such as status or ancillary data. In the following sections, this Standard will use the term 'property' in a broader sense, which does not necessarily imply "property measured by a sensor."

A dataset is composed of Data Components whose values need to be put into context in order to be fully understood and interpreted, by either humans or machines. The SWE Common Data

Model provides several pieces of information that are necessary to achieve this goal. More precisely, the SWE Common Data Model covers the following aspects of datasets description:

- Representation
- Nature of data and semantics (by using identifiers pointing to external semantics)
- Quality
- Structure
- Encoding

This requirement class constitutes the core of this standard. The standardization target types of this core are all models or software implementations seeking compliance with this standard.

REQUIREMENT 1

IDENTIFIER /req/core/core-concepts-used

INCLUDED IN Requirements class 1: /req/core

STATEMENT A derived model or software implementation shall correctly implement the concepts defined in the core of this standard.

7.2. Data Representation

Data representation deals with how property values are represented and stored digitally. Each component (or field) in a dataset carries a value that represents the state of a property. This representation will vary depending on the nature of the method used to capture the data and/or the target usage. For instance, a fluid temperature can be represented as a decimal number expressed in degrees Celsius (i.e., 25.4 °C), or as a categorical value taken from a list of possible choices (such as “freezing, cold, normal, warm, hot”).

The following types of representations have been identified: Boolean, Categorical, Continuous Numerical, Discrete Countable and Textual. The paragraphs below explain basic features of each of these representation types.

7.2.1. Boolean

A Boolean representation of a property can take only two values that should be “true/false” or “yes/no”. In a sense, this type of representation is a particular case of the categorical representation with only two predefined options.

Example: Examples

Motion detectors output can be represented by a boolean value – TRUE if there is motion in the room, FALSE otherwise.

On/Off status of a measurement system can be represented by a boolean value – TRUE if the system is on, FALSE if the system is off.

REQUIREMENT 2

IDENTIFIER	/req/core/boolean-rep-valid
INCLUDED IN	Requirements class 1: /req/core
STATEMENT	A boolean representation shall at least consist of a boolean value.

The “Boolean” class described in Clause 8.2.4 is used to define a data component with a Boolean representation.

7.2.2. Categorical

A categorical representation is a type of discrete representation of a property that only allows picking a value from a well defined list of possibilities (i.e., categories). This list is called a code space in this standard, following ISO 19103 terminology.

The different possible values constituting a code space are usually listed explicitly in an out-of-band dictionary or ontology. This is necessary because each value should be defined formally and unambiguously, so that it can be interpreted correctly.

Example: Examples

Biological or chemical species data is usually represented by a categorical data component that can leverage an existing controlled vocabulary.

A camera mode can be represented by a categorical value: AUTO_FOCUS, MANUAL_FOCUS, etc.

REQUIREMENT 3

IDENTIFIER	/req/core/categorical-rep-valid
INCLUDED IN	Requirements class 1: /req/core
STATEMENT	A categorical representation shall at least consist of a category identifier and information describing the value space of this identifier.

The “Category” class described in Clause 8.2.6 is used to define a data component with a categorical representation.

7.2.3. Numerical (continuous)

Perhaps the most used representation of a property value, especially in the science and technical communities, is the numerical one, as the majority of properties measured by sensors can be represented by numbers.

Numerical representation is often used for continuous values and, in this case, the representation consists of a decimal (often floating point) number associated to a scale or unit of measure. The unit specification is mandatory even for quantities such as ratios that have no physical unit (in this case a scale factor is provided such as 1, 1/100 for percents, 1/1000 for per thousands, etc.).

Example: Examples

Temperature measurements can be represented by a number associated to a unit such as degrees Celsius or Fahrenheit: 23.51°C, 94°F.

A velocity vector is composed of several values (usually 2 or 3) associated to a unit of speed: [1.0 2.0 3.0] m/s.

REQUIREMENT 4

IDENTIFIER /req/core/numerical-rep-valid

INCLUDED IN Requirements class 1: /req/core

STATEMENT A continuous numerical representation shall at least consist of a decimal number and the scale (or unit) used to express this number.

The “Quantity” class described in Clause 8.2.8 is used to define a data component with a decimal representation and a unit of measure.

7.2.4. Countable (discrete)

Discrete countable properties are also of interest and are most accurately captured with a numerical integer representation. They do not require a unit since the unit is always the unit of count (i.e., the person if we are counting persons, the pixel if we are counting pixels, etc). Note that continuous properties can also be represented as integers with certain combinations of scale and precision. This case should not be confused with the countable properties described here.

Example: Examples

Array indices and sizes are countable properties with no unit.

There are numerous other countable properties such as number of persons, number of bytes, number of frames, etc. for which the unit is obvious from the definition of the property itself.

A discrete countable representation should not be confused with a continuous numerical representation whose scale and precision allow encoding the property value as an integer.

REQUIREMENT 5

IDENTIFIER	/req/core/countable-rep-valid
INCLUDED IN	Requirements class 1: /req/core
STATEMENT	A countable representation shall at least consist of an integer number.

The “Count” class described in Clause 8.2.7 is used to define a data component with an integer representation and no unit of measure.

7.2.5. Textual

A textual representation is useful for providing human readable data, expressed in natural language, as well as various alpha numeric tokens that cannot be assigned to well-defined categories.

Example: Examples

Comments or notes written by humans (ex: data annotations, quality assessments).

Machine generated messages for which there is no taxonomy (ex: automatic alert messages).

Alphanumeric identifier schemes leading to a large number of possibilities that cannot be explicitly enumerated (ex: UUID, ISBN code, URN).

REQUIREMENT 6

IDENTIFIER	/req/core/textual-rep-valid
INCLUDED IN	Requirements class 1: /req/core
STATEMENT	A textual representation shall at least consist of a character string.

The “Text” class described in Clause 8.2.5 is used to define a data component with a textual representation.

7.2.6. Constraints

Constraints can be added to some representation types to further restrict the set of possible values allowed for a given property.

- A boolean representation cannot be restricted further since it is already limited to only two possibilities.
- A numerical representation can be constrained by a list of allowed values and/or bounded or unbounded intervals. A decimal representation can also be constrained by the number of significant digits after the decimal point.
- A categorical representation can be constrained by a list of possible choices, which should be a subset of the list of possibilities defined by the code space.
- A textual representation can be constrained by a pattern expressed in a well known language such as regular expression syntax.

These constraints apply only to the value of the data component to which they are associated. They shall not be used to express constraints on other data components or on any other information than the value.

Example: Examples

A decimal representation of an angular property such as latitude can be constrained to the $[-90^{\circ} \ 90^{\circ}]$ interval.

A temperature reading produced by a sensor can be constrained to the $[-50^{\circ}\text{C} \ +250^{\circ}\text{C}]$ range.

7.3. Nature of Data

We define “Nature of data” as the information needed to understand what property the value represents. It is thus connected to semantics and the semantic details are often provided by external sources such as dictionaries, taxonomies or ontologies. Note that it is independent of the type of representation used and it does not include information about how the data was actually measured or acquired. This lineage information should be described by other means as explained in Clause 7.4.3.

7.3.1. Human readable information

The first means by which nature of data can be communicated is through human readable text. The data component’s description, which is present in all data types defined in this specification, can hold any length of text for this purpose. The data component’s label is used to carry short

human readable information (i.e., a short name); this is useful to allow data consumers to quickly identify the represented property.

It is not recommended to use the concepts of “description” and “label” in a way that they contain robust semantic information (i.e., that machines can rely upon). The content of such fields is intended to be interpretable solely by humans.

7.3.2. Robust semantics

All SWE Common data types allow for associating each data component in a dataset with the definition of the Property that it represents.

REQUIREMENT 7

IDENTIFIER /req/core/semantics-defined

INCLUDED IN Requirements class 1: /req/core

STATEMENT All data values shall be associated with a clear definition of the property that the value represents.

It is recommended that a model uses references to out-of-band dictionaries rather than inline information because semantics are supposed to be shared by multiple datasets. Using references also helps by providing a framework that is independent from the actual semantic technology used.

The SWE Common UML models and JSON schemas described in this standard can be used in combination with any semantic web technology. It is thus possible to connect a SWE dataset description to an existing taxonomy provided the external register exposes a unique identifier for each entry.

These semantic references point to out-of-band semantic information that can be encoded in various languages, such as the Ontology Web Language (OWL) or GML dictionary.

REQUIREMENT 8

IDENTIFIER /req/core/semantics-resolvable

INCLUDED IN Requirements class 1: /req/core

STATEMENT If semantic information is provided by referencing out-of-band data, the locators or identifiers used to point to this information shall be resolvable by some well-defined method.

7.3.3. Time, space and projected quantities

Temporal, spatial and other projected quantities need to be further defined by specifying the reference frame and axis with respect to which the quantity is expressed. In SWE Common, any simple component type can be associated to a particular axis of a given reference frame.

Example: Examples

Satellite position data can be defined as a vector of 3 components, expressed in the J2000 ECI Cartesian frame, the 1st component being associated to the X axis, the 2nd to the Y axis and the 3rd to the Z axis.

Angular velocity data from an Inertial Measurement Unit can be defined as a vector of 3 components, expressed in the plane reference frame (for instance ENU defined by local East, North, Up directions), the Euler components being mapped to X, Y, Z respectively.

Relative time data can be given with respect to an arbitrary epoch itself positioned in a well defined reference frame such as TAI (from the French “Temps Atomique International” = International Atomic Time).

REQUIREMENT 9

IDENTIFIER /req/core/temporal-frame-defined

INCLUDED IN Requirements class 1: /req/core

STATEMENT A temporal quantity shall be expressed with respect to a well defined temporal reference frame and this frame shall be specified.

REQUIREMENT 10

IDENTIFIER /req/core/spatial-frame-defined

INCLUDED IN Requirements class 1: /req/core

STATEMENT A spatial quantity shall be expressed with respect to the axes of a well defined spatial reference frame and this frame shall be specified.

The “Time” class described in Clause 8.2.9 is designed for carrying a temporal reference frame or a time of reference in the case of relative time data.

The “Vector” class detailed in Clause 8.3.2 is a special type of record used to assign a reference frame to all its child-components.

The “Matrix” class defined in Clause 8.5.2 allows the definition of higher order tensor quantities.

This standard does not impose requirements on the type of reference frames that a standardization target shall support. Standards that are dependent on this specification can (and often should) however define a minimum set of reference frames that shall be supported by all implementations.

7.4. Data Quality

Quality information can be essential to the data consumer and the SWE Common Data Model provides simple and flexible ways to associate qualitative information with each component of a dataset.

7.4.1. Simple quality information

Simple quality information can be associated with any scalar data component, in the form of another scalar or range value. The quality information defined here applies solely to the value of the associated data component (i.e., the measurement value) and, depending on its data type, quality can be represented by a numerical, categorical or textual value, or by a range of values.

This quality information can be static, i.e., constant over the whole dataset, or dynamic and provided with the data itself. In this case, the quality value is in fact carried by another component of the dataset (and described in SWE Common as such).

The exact type of quality information provided should be specified via semantic tagging just like with any other property in SWE Common.

Example: Examples

Examples of quality measures are “absolute accuracy”, “relative accuracy”, “absolute precision”, “tolerance”, and “confidence level.”

Quality related comments can also describe operating conditions, such as “sensor contained blockage and was removed” or “engineer on site, values may be affected.” This information can inform the user of potential inaccuracy in the data across certain periods.

7.4.2. Nil Values

The concept of NIL value is used to indicate that the actual value of a property cannot be given in the data stream, and that a special code (i.e., reserved value) is used instead. It is thus a kind of quality information. The reason for which the value is not included is essential for a good interpretation of the data, so each reserved value is associated to a well-defined reason. In that sense, a NIL value definition is essentially a mapping between a reserved value and a reason.

Each component of a dataset can define one or several NIL values corresponding to one or more reasons.

Example: Example

In low level satellite imagery with, for instance, 8-bits per channel, the imagery metadata often defines: - A reserved value to indicate that a pixel value was “Below Detection Limit” usually set to ‘0’; and - A reserved value to indicate that a pixel value was “Above Detection Limit” usually set to ‘255’.

REQUIREMENT 11

IDENTIFIER /req/core/nil-reasons-defined

INCLUDED IN Requirements class 1: /req/core

STATEMENT A model of a NIL value shall always include a mapping between the selected reserved value and a well-defined reason.

7.4.3. Full lineage and traceability

Full lineage and traceability is not in the scope of this specification. It is fully addressed by the OGC® Sensor Model Language Standard, which allows robust definition of measurement chains, with detailed information about the processing that takes place at each stage of the chain. This means that complex lineage guarantying full traceability can be recorded in a SensorML process chain, separately from the data itself.

Datasets can be associated to lineage information described using the Sensor Model Language by using a metadata wrapper such as the “Observation” object defined in the OGC® Observations, Measurements and Samples Specification (OMS). In this standard, the “procedure” and “observer” properties of the “Observation” class allows attaching detailed information about the measurement process (that is to say a description of how the data was obtained, i.e., lineage), to the data itself.

7.5. Data Structure

Data structure defines how individual pieces of data are grouped, ordered, repeated and interleaved to form a complete data stream. The SWE Common models are based on data structures commonly accepted in computer science and formalized in ISO 11404. Classical aggregate datatypes are defined below:

- Record: consists of a list of fields, each of them being keyed by a field identifier and defining its own type that can be any scalar or aggregate structure.

- **Array:** consists of many elements of the same type, usually indexed by an integer. The element type can be any data structure including scalars and aggregates. The array size constitutes the upper bound of the index.
- **Choice:** consists of a list of alternatives, each of them being keyed by a tag value and having its own type. Only values for one alternative at a time are actually present in the data stream described by such a structure.

REQUIREMENT 12

IDENTIFIER /req/core/aggregates-model-valid

**INCLUDED
IN** Requirements class 1: /req/core

STATEMENT Aggregate data structures shall be implemented in a way that is consistent with definitions of ISO 11404.

This standard also defines the concept of “data component” as any part of the structure of a dataset, aggregate or not. It is thus the superset of all the aggregate structures described above and of all scalar elements implementing the representations described in Clause 7.2.

Example: Example

A dataset representing a time series of observations acquired by a mobile sensor can be encoded with various methods depending on the requirements:

- JSON encoding can be used when data needs to be easily styled to other markup formats (such as HTML) or when precise error localization (in the case of an error in the stream) is needed;
- ASCII encoding can be used to achieve a good compromise between readability and size efficiency; and
- Binary encoding can be used (eventually with embedded compression) when pure performance (i.e., size but also reading and writing throughput) is the main concern.

A data component can be both a data descriptor and a data container:

- A data component used as a data descriptor defines the structure, representation, semantics, quality, and other metadata of a data set but does not include the actual data values; and
- A data component used as a data container equally defines the dataset but also includes the actual property values.

7.6. Data Encoding

A key concept of the SWE Common Data Model is the ability to separate data values themselves from the description of the data structure, semantics and representation. This allows verbose metadata to be used in order to robustly define the content and meaning of a dataset while still being able to package the data values in very efficient manners.

Data encoding methods define how the data is packed as blocks that can efficiently be transferred or stored using various protocols and formats. Different methods allow encoding the data as JSON, text (CSV like), binary and even compressed or encrypted formats in a way that is agnostic to a particular structure. This allows any of the encoding methods to be selected and used based on a particular requirement, such as performance, re-use of tools, alignment with existing standards and so on.

Requirement 13	
IDENTIFIER	/req/core/encoding-method-valid
INCLUDED IN	Requirements class 1: /req/core
STATEMENT	All encoding methods shall be applicable to any arbitrarily complex data structures as long as they are made of the data components described in Clause 7.5.



8

UML CONCEPTUAL MODELS (NORMATIVE)

This standard defines normative UML models with which derived encoding models as well as all future separate extensions should be compliant. The standardization target type for the UML requirements classes defined in this clause is thus a software implementation or an encoding model that directly implements the conceptual models defined in this standard.

8.1. Package Dependencies

The following packages are defined by the SWE Common Data Model:

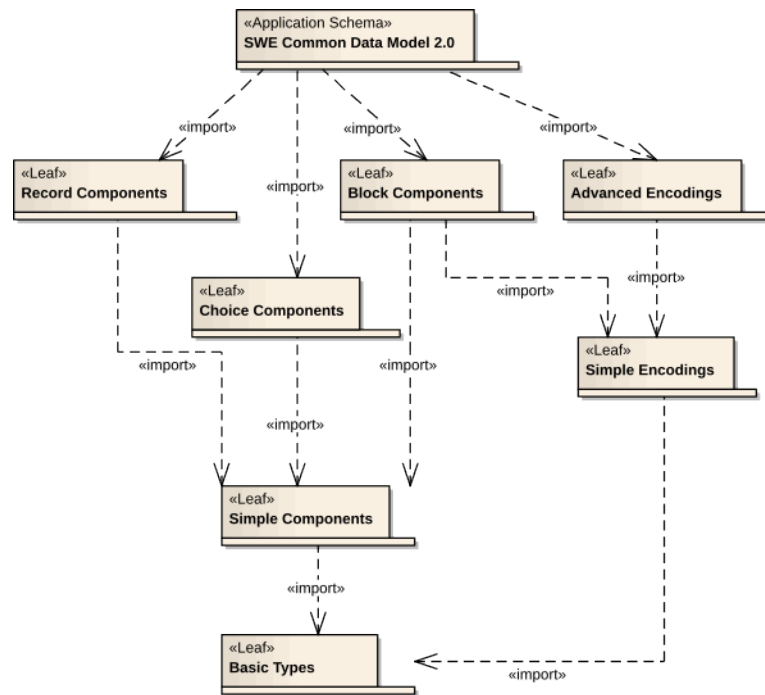


Figure 2 – Internal Package Dependencies

This standard also has dependencies on external packages defined by other standards, namely ISO 19103, ISO 19108 and ISO 19111, as show below:

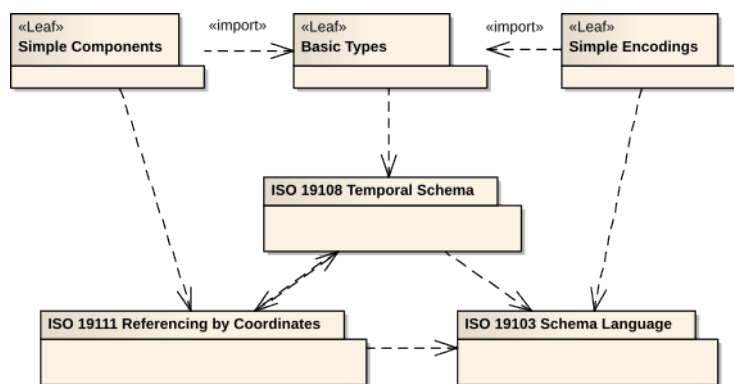


Figure 3 – External Package Dependencies

8.2. Requirements Class: Basic Types and Simple Components Packages

REQUIREMENTS CLASS 2: SIMPLE COMPONENTS UML PACKAGE

IDENTIFIER	/req/uml-simple-components
TARGET TYPE	Software Implementation or Encoding of the Conceptual Models
CONFORMANCE CLASS	Conformance class A.2: /conf/uml-simple-components
PREREQUISITE	Requirements class 1: /req/core
NORMATIVE STATEMENTS	<p>Requirement 14: /req/uml-simple-components/package-fully-implemented</p> <p>Requirement 15: /req/uml-simple-components/iso19103-implemented</p> <p>Requirement 16: /req/uml-simple-components/iso19108-implemented</p> <p>Requirement 17: /req/uml-simple-components/definition-present</p> <p>Requirement 18: /req/uml-simple-components/axis-valid</p> <p>Requirement 19: /req/uml-simple-components/axis-defined</p> <p>Requirement 20: /req/uml-simple-components/ref-frame-defined</p> <p>Requirement 21: /req/uml-simple-components/value-constraint-valid</p> <p>Requirement 22: /req/uml-simple-components/value-attribute-present</p>

REQUIREMENTS CLASS 2: SIMPLE COMPONENTS UML PACKAGE

Requirement 23: /req/uml-simple-components/category-constraint-valid
Requirement 24: /req/uml-simple-components/category-enum-defined
Requirement 25: /req/uml-simple-components/category-value-valid
Requirement 26: /req/uml-simple-components/time-ref-frame-defined
Requirement 27: /req/uml-simple-components/time-ref-time-valid
Requirement 28: /req/uml-simple-components/time-local-frame-valid
Requirement 29: /req/uml-simple-components/range-value-valid
Requirement 30: /req/uml-simple-components/category-range-valid
Requirement 31: /req/uml-simple-components/category-range-codespace-order
Requirement 32: /req/uml-simple-components/time-range-valid
Requirement 33: /req/uml-simple-components/nil-reason-resolvable
Requirement 34: /req/uml-simple-components/nil-value-type-coherent
Requirement 35: /req/uml-simple-components/allowed-values-unit-coherent

Data components are the most essential part of the SWE Common Data Model. They are used to describe all types of data structures, whether they represent data stream contents, tasking messages, alert messages or process inputs/outputs.

The “Simple Components” UML package contains classes modeling simple data components, that is to say scalar components and range components (i.e., value extents). These classes implement concepts defined in the core section of this standard, and are designed to collect information about nature, representation and quality of data. These include six scalar types – Boolean, Text, Category, Count, Quantity, and Time – as well as four range types – CategoryRange, CountRange, QuantityRange and TimeRange.

The “Basic Types” UML package from which the “Simple Components” package is dependent is also included in this requirements class.

As an overview, conceptual models of the six scalar component types are shown on the following UML class diagram:

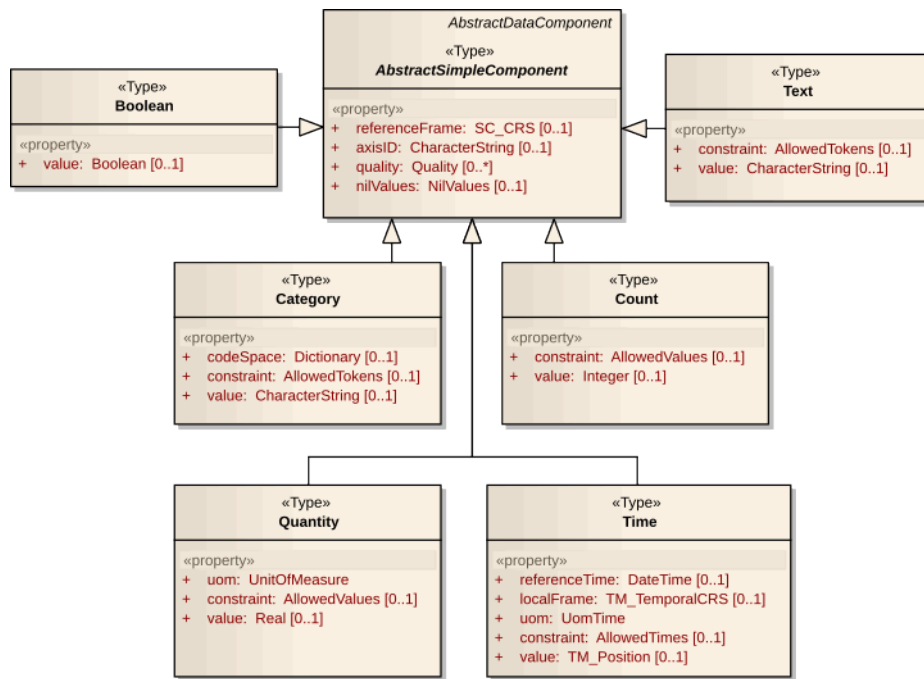


Figure 4 – Scalar Data Components

Classes representing the four range data components are shown on the diagram below:

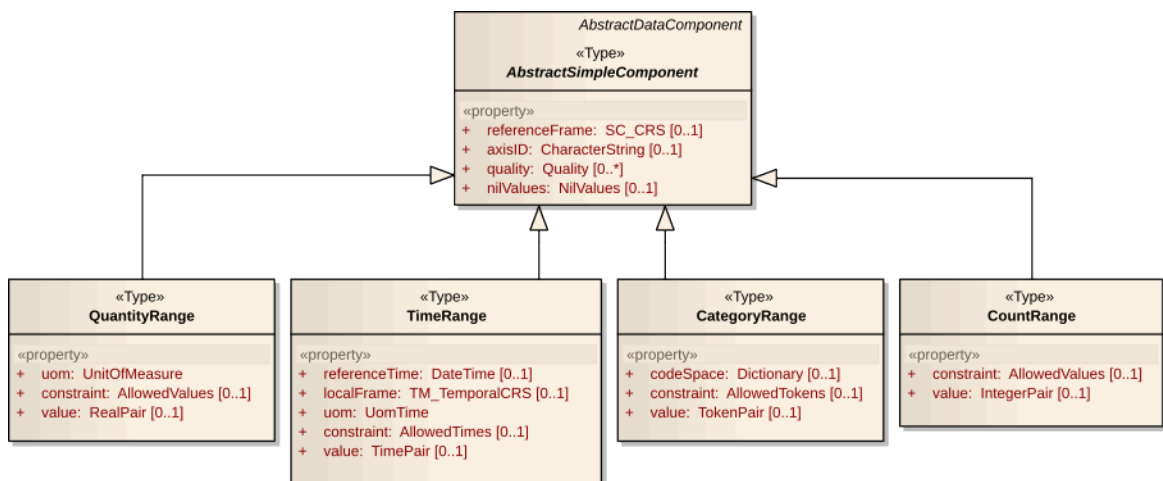


Figure 5 – Range Data Components

Details and requirements about each of these classes are given in the following sections.

REQUIREMENT 14

IDENTIFIER /req/uml-simple-components/package-fully-implemented

INCLUDED IN Requirements class 2: /req/uml-simple-components

REQUIREMENT 14

STATEMENT The encoding or software shall correctly implement all classes defined in the “Simple Components” and “Basic Types” UML packages.

Several dependencies to ISO standards exist and are detailed below.

Data types from several packages of the ISO 19103 standard are used directly which makes this requirement class dependent on it. These data types are “CharacterString,” “Boolean,” “Real,” “Integer,” “Date,” “Time,” “DateTime,” “ScopedName,” “UnitOfMeasure,” and “UomTime.”

REQUIREMENT 15

IDENTIFIER /req/uml-simple-components/iso19103-implemented

INCLUDED IN Requirements class 2: /req/uml-simple-components

STATEMENT The encoding or software shall correctly implement all UML classes defined in ISO 19103 that are referenced directly or indirectly by this standard.

The “TM_Position” data type from the “Temporal Reference System” package of the ISO 19108 standard is also used.

REQUIREMENT 16

IDENTIFIER /req/uml-simple-components/iso19108-implemented

INCLUDED IN Requirements class 2: /req/uml-simple-components

STATEMENT The encoding or software shall correctly implement all UML classes defined in ISO 19108 that are referenced directly or indirectly by this standard.

The “SC_CRS” and “TM_Temporal_CRS” classifiers are referenced conceptually from ISO 19111 but their implementation is not required by this standard. Implementations are allowed to simply use a Coordinate Reference System (CRS) identifier as a means of recognizing predefined coordinate reference systems. The use of identifiers from the EPSG database is recommended in this case. However, when new CRS definitions need to be created (e.g., engineering CRS attached to sensors or platforms), the models defined in ISO 19111 shall be used.

8.2.1. Basic Data Types

This requirement class also includes requirements for the “Basic Types” UML package. This package defines low level data types that are used as property types by classes defined in the other packages.

Data types defined in this package relate to defining pairs of data types defined in ISO 19103 for use within classes describing value extents:



Figure 6 — Basic types for pairs of scalar types

8.2.2. Attributes shared by all data components

All SWE Common data component classes carry standard attributes inherited (transitively) from the “AbstractDataComponent” and “AbstractSWEIdentifiable” classes (The “AbstractSWEIdentifiable” class is actually defined in the “Basic Types” package but is shown here for clarity). The class hierarchy is shown on the following UML diagram:

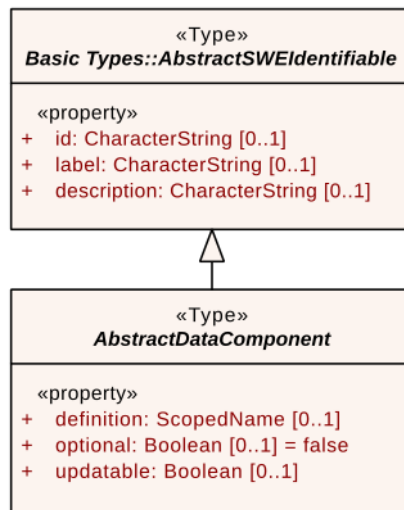


Figure 7 — AbstractDataComponent Class

Label and Description

The optional “label” and “description” attributes can be used to provide human readable information describing what property the component represents. The “label” is meant to hold a short descriptive name whereas “description” can carry any length of plain text. These two fields should not be used to specify robust semantic information (see Clause 7.3.2). Instead, the “definition” attribute described below should be used for that purpose.

Identifier

The optional “id” attribute allows assigning a unique identifier to the component, so that it can be referenced later on. It can be used, for example, when defining the unique identifier of a universal constant.

Definition

The “definition” attribute identifies the property (often an observed property in our context) that the data component represents by using a scoped name. It should map to a controlled term defined in a (web accessible) dictionary, registry or ontology. Such terms provide the formal textual definition agreed upon by one or more communities, eventually illustrated by pictures and diagrams as well as additional semantic information such as relationships to units and other concepts, ontological mappings, etc.

Example: Examples

The definition may indicate that the value represents an atmospheric temperature using a URN such as “urn:ogc:def:property:OGC::SamplingTime” referencing the complete definition in a register.

The definition may also be a URL linking to a concept defined in an ontology such as [<http://www.opengis.net/def/OGC/0/SamplingTime>]. The label could be “Sampling Time”, which allows quick identification by human data consumers.

The description could be “Time at which the observation was made as measured by the on-board clock” which adds contextual details.

Flags

The “optional” attribute is an optional flag indicating if the component value can be omitted in the data stream. It is only meaningful if the component is used as a schema descriptor (i.e., not for a component containing an inline value). It is ‘false’ by default.

The “updatable” attribute is an optional flag indicating if the component value is fixed or can be updated. It is only applicable if the data component is used to define the input of a process (i.e., when used to define the input or parameter of a service, process or sensor, but not when used to define the content of a dataset).

Example: Examples

The “updatable” flag can be used to identify what parameters of a system are changeable. The exact semantics depends on the context. For example:

- In SensorML process chains, the “updatable” flag is used to identify process parameters that can accept an incoming connection (and thus can get changed while the process is in execution);
- In a SensorML System it is used to indicate whether or not a system parameter is changeable, either by an operator (i.e., by turning a screw or inserting a jumper) or remotely by sending a command; and
- In the Sensor Planning Service it is used to indicate if tasking parameters are changeable by the client (i.e., by using the Update operation) after a task has been submitted.

8.2.3. Attributes shared by all simple data components

As shown on Figures 4 and 5, classes modeling simple data components inherit attributes from the “AbstractSimpleComponent” class from which they are directly derived. This abstract class is shown again below:

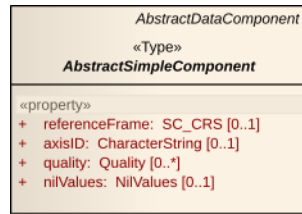


Figure 8 – AbstractSimpleComponent Class

The definition attribute inherited from the “AbstractDataComponent” class is mandatory on this class and thus on all its descendants.

REQUIREMENT 17

IDENTIFIER /req/uml-simple-components/definition-present

INCLUDED IN Requirements class 2: /req/uml-simple-components

STATEMENT The “definition” attribute shall be specified by all instances of concrete classes derived from “AbstractSimpleComponent”.

Reference Frames and Axes

It provides two attributes allowing the association of a data component to a reference frame and an axis and thus implements core concepts introduced in Clause 7.3.3. These attributes are used for a component which value is the projection of a property along a temporal or spatial axis.

The “referenceFrame” attribute identifies the reference frame (as defined by the “SC_CRS” object) relative to which the coordinate value is given. The “axisID” attribute takes a string that uniquely identifies one of the reference frame’s axes along which the coordinate value is given.

REQUIREMENT 18

IDENTIFIER /req/uml-simple-components/axis-valid

INCLUDED IN Requirements class 2: /req/uml-simple-components

STATEMENT The value of the “axisID” attribute shall correspond to the “axisAbbrev” attribute of one of the coordinate system axes listed in the specified reference frame definition.

The union of these two attributes thus uniquely identifies one axis of one given reference frame along which the value of the component is expressed. Note that even though the ISO 19111 model assigns units to CRS axes in addition to a direction, only the direction is used in this standard and the unit is defined by the data component itself. This allows expressing other quantities than the one predefined along the CRS's axes such as velocity, acceleration or rotation.

A component representing a projected quantity can be defined in isolation or can be contained within a "Vector" or "Matrix" aggregate when it contributes to the specification of a multi-dimensional quantity (see Clauses 8.3.2 and 8.5.2). In this last case the reference frame definition is usually inherited from the parent "Vector" or "Matrix" instance and is thus omitted from the scalar component itself. However, the "axisID" attribute still needs to be specified on "Vector" components.

REQUIREMENT 19

IDENTIFIER /req/uml-simple-components/axis-defined

INCLUDED IN Requirements class 2: /req/uml-simple-components

STATEMENT The "axisID" attribute shall be specified by all instances of concrete classes derived from "Abstract SimpleComponent" and representing a property projected along a spatial axis.

REQUIREMENT 20

IDENTIFIER /req/uml-simple-components/ref-frame-defined

INCLUDED IN Requirements class 2: /req/uml-simple-components

STATEMENT The "referenceFrame" attribute shall be specified by all instances of concrete classes derived from "AbstractSimpleComponent" and representing a property projected along a spatial or temporal axis, except if it is inherited from a parent aggregate (Vector or Matrix).

Quality

The optional "quality" attribute is used to provide simple quality information as discussed in Clause 7.4.1. It is of type "Quality" which is a union of several classes as defined in Clause 8.2.15. Its multiplicity is more than one which means that several quality measures can be given on for a single data component.

Example: Example

Both precision and accuracy of the value associated to a data component can be specified concurrently (see http://en.wikipedia.org/wiki/Accuracy_and_precision for a good explanation of the difference between the two).

Nil Values

The optional “nilValues” attribute is used to provide a list (i.e., one or more) of NIL values as defined in Clause 7.4.2. The model of the “NilValues” class is detailed in Clause 8.2.16.

Concrete sub-classes of “AbstractSimpleComponent” can also define a “constraint” attribute that allows further restriction of the possible values allowed by the corresponding representation. This implements concepts defined in Clause 7.2.6. These constraints always apply to the value of the property as represented by the corresponding data component whether this value is given inline (data container case) or out-of-band (data descriptor case).

Constraints

REQUIREMENT 21

IDENTIFIER /req/uml-simple-components/value-constraint-valid

INCLUDED IN Requirements class 2: /req/uml-simple-components

STATEMENT The property value (formally the representation of the property value) attached to an instance of a class derived from “AbstractSimpleComponent” shall satisfy the constraints specified by this instance.

All concrete sub-classes of “AbstractSimpleComponent” also define a “value” attribute. This attribute is not defined in this abstract class because it has a different primitive type in each concrete data component class (See following clauses).

REQUIREMENT 22

IDENTIFIER /req/uml-simple-components/value-attribute-present

INCLUDED IN Requirements class 2: /req/uml-simple-components

STATEMENT All concrete classes derived from the “AbstractSimpleComponent” class (directly or indirectly) shall define an optional “value” attribute and use it as defined by this standard.

The “value” attribute is always optional on any simple data component in order to allow for both data descriptor and data container cases.

- When the data component is used as a data container, this attribute always carries the value of the associated property (formally the representation of the estimated or asserted value of the property). Quality information, nil values definitions and constraints thus apply to the value taken by this attribute.
- When the data component is used as a data descriptor, its actual value is provided somewhere else, often encoded as part of a larger data block. In this case, quality information, nil values definitions and constraints apply to the out-of-band value and not

to the “value” attribute. Instead, the “value” attribute can then be used to specify a default value.

Whether the data component is used as a descriptor or a container depends on the context and should be explicitly stated by any standard that makes use of the SWE Common Data Model.

All UML classes in this package that derive from “AbstractSimpleComponent” define a “value” attribute with the adequate primitive type and whose meaning is the one explained above.

8.2.4. Boolean Class

The “Boolean” class is used to specify a scalar data component with a Boolean representation as defined in Clause 7.2.1. It derives from “AbstractSimpleComponent” and is shown below:

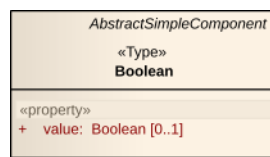


Figure 9 – Boolean Class

The “value” attribute of this class is of the boolean primitive type.

NOTE: The boolean primitive type is defined in ISO19103 and is not to be confused with the “Boolean” class defined in this standard. This clause is the only place in this standard where the ISO 19103 boolean data type is referenced. All other occurrences of the “Boolean” class in this standard refer to the class defined in this clause.

8.2.5. Text Class

The “Text” class is used to specify a component with a textual representation as defined in Clause 7.2.5. It derives from “AbstractSimpleComponent” and is shown below:

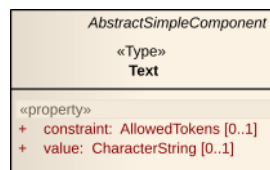


Figure 10 – Text Class

Constraints

The “constraint” attribute allows further restricting the range of possible values by using the “AllowedTokens” class defined in Clause 8.2.17. This class allows the definition of the constraint by either enumerating the allowed tokens and/or by specifying a pattern that the value must match.

Value

The “value” attribute (or the corresponding value in out-of-band data) is a string that must match the constraint.

NOTE: The “Text” component can be used to wrap a string representing complex content such as an expression in a programming language, xml or html content. This practice should however be used only for systems that don’t require high level of interoperability since the client must know how to interpret the content. Also care must be taken to properly escape such content before it is inserted in a JSON document or in a SWE Common data stream.

8.2.6. Category Class

The “Category” class is used to specify a scalar data component with a categorical representation as defined in Clause 7.2.2. It derives from “AbstractSimpleComponent” and is shown below:

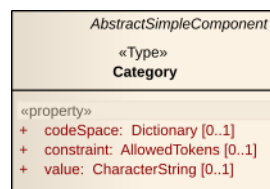


Figure 11 – Category Class

Code Space

The “codeSpace” attribute is of type “Dictionary” and allows listing and defining the meaning of all possible values for this component. It is expected that instances of the “Dictionary” class will usually be referenced (rather than included inline) by implementations of this class since the code space definition is usually obtained from a controlled vocabulary maintained at a remote location. This type of implementation is the one chosen in the JSON encodings defined by this standard.

Constraints

The “constraint” attribute allows further restricting the list of possible values by using the “AllowedTokens” class defined in Clause 8.2.17. This is usually done by specifying a limited list of possible values, which have to be extracted from the code space.

REQUIREMENT 23

IDENTIFIER /req/uml-simple-components/category-constraint-valid

INCLUDED IN Requirements class 2: /req/uml-simple-components

STATEMENT When an instance of the “Category” class specifies a code space, the list of allowed tokens provided by the “constraint” property of this instance shall be a subset of the values listed in this code space.

It is also possible to use this class without a code space, even though it is not recommended as values allowed in the component would then not be formally defined. However, as the intent of this class is to always represent a value extracted from a set of possible options, a constraint shall be defined if no code space is specified.

Requirement 24	
IDENTIFIER	/req/uml-simple-components/category-enum-defined
INCLUDED IN	Requirements class 2: /req/uml-simple-components
STATEMENT	An instance of the “Category” class shall either specify a code space or an enumerated list of allowed tokens, or both.

Value

The “value” attribute (or the corresponding value in out-of-band data) is a string that must be one of the items of the code space and also match the constraint.

Requirement 25	
IDENTIFIER	/req/uml-simple-components/category-value-valid
INCLUDED IN	Requirements class 2: /req/uml-simple-components
STATEMENT	When an instance of the “Category” class specifies a code space, the value of the property represented by this instance shall be equal to one of the entries of the code space.

8.2.7. Count Class

The “Count” class is used to specify a scalar data component with a discrete countable representation as defined in Clause 7.2.4. It derives from “AbstractSimpleComponent” and is shown below:

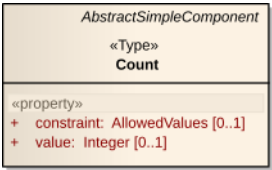


Figure 12 – Count Class

Constraints

The “constraint” attribute can be used to restrict the range of possible values to a list of inclusive intervals and/or single values using the “AllowedValues” class defined in Clause 8.2.18. Numbers

used to define these constraints should be integers and expressed in the same scale as the count value itself. The “significantFigures” constraint allowed by the “AllowedValues” class is not applicable to the “Count” class.

Value

The “value” attribute (or the corresponding value in out-of-band data) is an integer that must be within one of the constraint intervals or exactly one of the enumerated values.

8.2.8. Quantity Class

The “Quantity” class is used to specify a component with a continuous numerical representation as defined in Clause 7.2.3. It derives from “AbstractSimpleComponent” and is shown below:

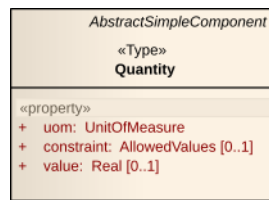


Figure 13 — Quantity Class

Unit of Measure (UoM)

In addition to attributes inherited from the “AbstractSimpleComponent” class, this class provides a unit of measure declaration through the “uom” attribute. This unit is essential for the correct interpretation of data represented as decimal numbers and is thus mandatory. Quantities with no physical unit still have a scale (such as unity, percent, per thousands, etc.) that must be specified with this property.

Constraints

The “constraint” attribute is used to restrict the range of possible values to a list of inclusive intervals and/or single values using the “AllowedValues” class defined in Clause 8.2.18. Numbers used to define these constraints must be expressed in the same unit as the quantity value itself. Additionally, it is possible to constrain the number of significant digits that can be added after the decimal point.

Value

The “value” attribute (or the corresponding value in out-of-band data) is a real value that is within one of the constraint intervals or exactly one of the enumerated values, and most importantly is expressed in the unit specified.

8.2.9. Time Class

The “Time” class is used to specify a component with a date-time representation and whose value is projected along the axis of a temporal reference frame. This class is also necessary

to specify that a time value is expressed in a calendar system. This class derives from “AbstractSimpleComponent” and is shown below:

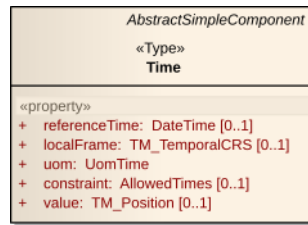


Figure 14 – Time Class

Time is treated as a special type of continuous numerical quantity that can be either expressed as a scalar number with a temporal unit or a calendar date with or without a time of day. Consequently, this class has all properties of the “Quantity” class, plus some others that are specific to the treatment of time.

Reference Frame

As time is always expressed relative to a particular reference frame, the “referenceFrame” attribute inherited from the parent class “AbstractSimpleComponent” shall always be set on instances on this class unless the default ‘UTC’ is meant.

REQUIREMENT 26

IDENTIFIER /req/uml-simple-components/time-ref-frame-defined

INCLUDED IN Requirements class 2: /req/uml-simple-components

STATEMENT The “referenceFrame” attribute inherited from “AbstractSimple Component” shall always be set on instance of the “Time” class unless the UTC temporal reference system is used.

Note that specifying the frame of reference is required even when using ISO notation because there can be ambiguities between several universal time references such as UTC, TAI, GPS, UT1, etc... Differences between these different time reference systems are indeed in the order of a few seconds (and increasing), that is to say not negligible in various situations.

Example: Example

J2000 is a well known epoch in astronomy and is equal to:

- January 1, 2000, 11:59:27.816 in the TAI time reference system
- January 1, 2000, 11:58:55.816 in the UTC time reference system
- January 1, 2000, 11:59:08.816 in the GPS time reference system

These offsets are not always constant and depend on the irregular insertion of leap seconds in UTC.

The “axisID” attribute inherited from the parent class does not need to be set since a time reference system always has a single dimension. However it can be set to ‘T’ for consistency with spatial axes.

Reference Time

The “referenceTime” attribute is used to specify a different time origin than the one sometimes implied by the “referenceFrame”. This is used to express a time relative to an arbitrary epoch (i.e., different from the origin of a well known reference frame). The new time origin specified by “referenceTime” shall be expressed with respect to the reference frame specified and is of type “DateTime”. This forces the definition of this origin as a calendar date/time combination.

REQUIREMENT 27

IDENTIFIER /req/uml-simple-components/time-ref-time-valid

INCLUDED IN Requirements class 2: /req/uml-simple-components

STATEMENT The value of the “referenceTime” attribute shall be expressed with respect to the system of reference indicated by the “referenceFrame” attribute.

Example: Example

This class can be used to define a value expressed as a UNIX time (i.e., number of seconds elapsed since January 1, 1970, 00:00:00 GMT) by:

- Specifying that the reference frame is the UTC reference system;
- Setting the reference time to January 1, 1970, 00:00:00 GMT; and
- Setting the unit of measure to seconds.

See definitions of some commonly accepted time standards at http://en.wikipedia.org/wiki/Time_standard or <http://stjarnhimlen.se/comp/time.html>.

Local Frame

The optional “localFrame” attribute allows for the definition of a local temporal frame of reference through the value of the component (i.e., we are specifying a time origin), as opposed to the referenceFrame which specifies that the value of the component is in reference to this frame.

REQUIREMENT 28

IDENTIFIER /req/uml-simple-components/time-local-frame-valid

INCLUDED IN Requirements class 2: /req/uml-simple-components

REQUIREMENT 28

STATEMENT The “localFrame” attribute of an instance of the “Time” class shall have a different value than the “referenceFrame” attribute.

This feature allows chaining several relative time positions. This is similar to what is done with spatial position in a geopositioning algorithm (and which is also supported by this standard using the “Vector” class).

Example: Example 1

In the case of a whiskbroom scanner instrument, the “sampling time” is often expressed relative to the “scan start time” which is itself given relative to the “mission start time”. It is important to properly identify the chain of time reference systems at play so that the adequate process can compute the absolute time of every measurement made (Note that it is often not practical to record the absolute time of each single measurement when high sampling rates are used).

Example 2

A model forecast may represent its result times relative to the “run time” of the model for efficient encoding. The values of the output will be in reference to this base epoch. In this example the “referenceFrame” attribute of the model time is set to UTC and the “localFrame” set as “ModelTime”. The model result would then define its “referenceFrame” as “ModelTime”, allowing the time values to be encoded relative to the specified time origin.

Unit of Measure (UoM)

The “uom” attribute is mandatory since time is a continuous property that shall always be expressed in a well defined scale. The only units allowed are obviously time units.

Constraints

Similarly to the “Quantity” class, the “constraint” attribute allows further restricting the range of possible time values by using the “AllowedTimes” class defined in Clause 8.2.19.

Value

The “value” attribute (or the corresponding value in out-of-band data) is of type “TimePosition” (see Clause 8.2.1) and must match the constraint.

8.2.10. Requirements applicable to all range classes

This UML package defines four classes “CategoryRange,” “CountRange,” “QuantityRange,” and “TimeRange” that are used for representing extents of property values. These classes have common requirements that are expressed in this clause.

The “value” attribute of all these classes takes a pair of values (with a datatype corresponding to the representation) that represent the inclusive minimum and maximum bounds of the extent. These values must both satisfy the constraints specified by an instance of the class, and be expressed in the unit specified when applicable.

REQUIREMENT 29

IDENTIFIER /req/uml-simple-components/range-value-valid

INCLUDED IN Requirements class 2: /req/uml-simple-components

STATEMENT Both values specified in the “value” property of an instance of a class representing a property range (i.e., “CategoryRange”, “CountRange”, “QuantityRange” and “TimeRange”) shall satisfy the same requirements as the scalar value used in the corresponding scalar classes.

NOTE: *These classes are intentionally not derived from their scalar counterparts because they are aggregates of two values and should be treated as such by implementations (especially by encoding methods defined in this standard).*

8.2.11. CategoryRange Class

The “CategoryRange” class is used to express a value extent using the categorical representation of a property. It defines the same attributes as the “Category” class and those should be used in the same way (see Clause 8.2.6):

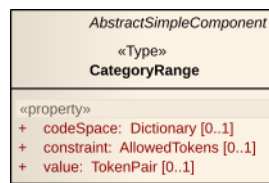


Figure 15 — CategoryRange Class

REQUIREMENT 30

IDENTIFIER /req/uml-simple-components/category-range-valid

INCLUDED IN Requirements class 2: /req/uml-simple-components

STATEMENT All requirements associated to the “Category” class defined in Clause 8.2.6 apply to the “Category Range” class.

Code Space

The “CategoryRange” class also requires that the underlying code space is well-ordered (i.e., the ordering of the different categories in the code space is clearly defined) so that the range is meaningful.

REQUIREMENT 31

IDENTIFIER /req/uml-simple-components/category-range-codespace-order

INCLUDED IN Requirements class 2: /req/uml-simple-components

STATEMENT The code space specified by the “codeSpace” attribute of an instance of the “CategoryRange” class shall define a well-ordered set of categories.

Example: Example

A “CategoryRange” can be used to specify the approximate time of a geological event by using names of geological eons, eras or periods such as [Archean — Proterozoic] or [Jurassic — Cretaceous].

Value

The “value” attribute of the “CategoryRange” class takes a pair of tokens representing the inclusive minimum and maximum bounds of the extent.

8.2.12. CountRange Class

The “CountRange” class is used to express a value extent using the discrete countable representation of a property. It defines the same attributes as the “Count” class and those should be used in the same way (see Clause 8.2.7):

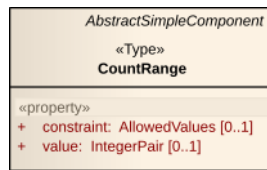


Figure 16 — CountRange Class

Value

The “value” attribute of the “CountRange” class takes a pair of integer numbers representing the inclusive minimum and maximum bounds of the extent.

8.2.13. QuantityRange Class

The “QuantityRange” class is used to express a value extent using the discrete countable representation of a property. It defines the same attributes as the “Quantity” class and those should be used in the same way (see Clause 8.2.8):

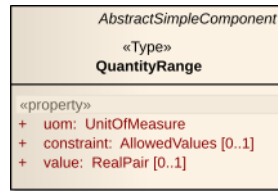


Figure 17 – QuantityRange Class

Value

The “value” attribute of the “QuantityRange” class takes a pair of real numbers representing the inclusive minimum and maximum bounds of the extent.

8.2.14. TimeRange Class

The “TimeRange” class is used to express a value extent of a time property. It defines the same attributes as the “Time” class and those should be used in the same way (see Clause 8.2.9):

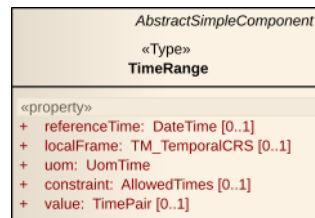


Figure 18 – TimeRange Class

REQUIREMENT 32

IDENTIFIER /req/uml-simple-components/time-range-valid

INCLUDED IN Requirements class 2: /req/uml-simple-components

STATEMENT All requirements associated to the “Time” class defined in Clause 8.2.9 apply to the “TimeRange” class.

The “value” attribute of the “TimeRange” class takes a pair of values of type “TimePosition” representing the inclusive minimum and maximum bounds of the extent.

8.2.15. Quality Union

The “Quality” class is a union allowing the use of different representations of quality.

Quality can indeed be specified as a decimal value, an interval, a categorical value or a textual statement. In our model, quality objects are in fact data components used in a recursive way, as shown on the following diagram:

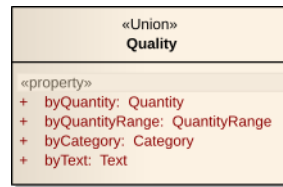


Figure 19 – Quality Union

These different representations of quality are useful to cover most use cases where simple quality information is provided with the data.

Example: Examples

“Quantity” is used to specify quality as a decimal number such as accuracy, variance and mean, or probability.

“QuantityRange” is used to specify a bounded interval of variation such as a bi-directional tolerance.

“Category” is used for a quality statement based on a well defined taxonomy such as certification levels.

“Text” is used to include a textual quality statement such as a comment written by a field operator.

The “definition” attribute of the chosen quality component helps to further define the type of quality information given just like any other data component, and the “uom” should be specified in the case of a decimal quality value or interval.

NOTE: Reusing data components to specify quality also allows the inclusion of quality values in the data stream itself. This is useful if the quality is varying and re-estimated for each measurement. This is for example the case in a GPS receiver where both horizontal and vertical errors are given along with the geographic position.

8.2.16. NilValues Class

The “NilValues” class is used by all classes deriving from “AbstractSimpleComponent”. It allows the specification of one or more reserved values that may be included in a data stream when the normal measurement value is not available (see Clause 7.4.2). The UML model of this class is given below:

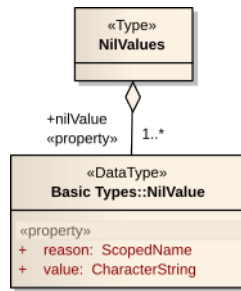


Figure 20 — NilValues Class

An instance of the “NilValues” class is composed of one to many “NilValue” objects, each of which specifies a mapping between a reserved value and a reason.

The mandatory “reason” attribute indicates the reason why a measurement value is not available. It is a resolvable reference to a controlled term that provides the formal textual definition of this reason (usually agreed upon by one or more communities).

REQUIREMENT 33

IDENTIFIER /req/uml-simple-components/nil-reason-resolvable

INCLUDED IN Requirements class 2: /req/uml-simple-components

STATEMENT The “reason” attribute of an instance of the “NilValue” class shall map to the complete human readable definition of the reason associated with the NIL value.

The mandatory “value” attribute specifies the data value that would be found in the stream to indicate that a measurement value is missing for the corresponding reason. The range of values allowed here is the range of values allowed by the datatype of the parent data component.

REQUIREMENT 34

IDENTIFIER /req/uml-simple-components/nil-value-type-coherent

INCLUDED IN Requirements class 2: /req/uml-simple-components

STATEMENT The value used in the “value” property of an instance of the “NilValue” class shall be compatible with the datatype of the parent data component object.

This means that when specifying NIL values for a “Quantity” component, only real values are allowed (in most implementations, this includes NaN, -INF and INF) and for a “Count” component only integer values are allowed.

Consequently, it is also impossible to specify NIL values for a “Boolean” data component since it allows only two possible values. In this case a “Category” component should be used.

There are no restrictions on the choice of NIL values for “Category” and “Text” components since their datatype is String.

8.2.17. AllowedTokens Class

The “AllowedTokens” class is used to express constraints on the value of a data component represented by a “Text” or a “Category” class. The UML class is shown below:

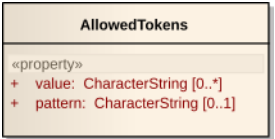


Figure 21 – AllowedTokens Class

This class allows defining the constraint either by enumerating a list of allowed values by using one or more “value” attributes and/or by specifying a pattern that the value must match. The value must then either be one of the enumerated tokens or match the pattern.

8.2.18. AllowedValues Class

The “AllowedValues” class is used to express constraints on the value of a data component represented by a “Count” or a “Quantity” class. The UML class is shown below:

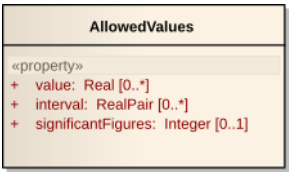


Figure 22 – AllowedValues Class

This class allows constraints to be defined either by enumerating a list of allowed values and/or a list of inclusive intervals. To be valid, the value must either be one of the enumerated values or included in one of the intervals. The numbers used in the “value” and “interval” properties shall be expressed in the same unit as the parent data component.

REQUIREMENT 35

IDENTIFIER /req/uml-simple-components/allowed-values-unit-coherent

INCLUDED IN Requirements class 2: /req/uml-simple-components

REQUIREMENT 35

STATEMENT The scale of the numbers used in the “enumeration” and “interval” properties of an instance of the “AllowedValues” class shall be expressed in the same scale as the value(s) that the constraint applies to.

If the parent data component instance is used to define a projected quantity (i.e., when the “axisID” is set), then the constraints given by this class are expressed along the same spatial reference frame axis.

The number of significant digits can also be specified with the “significantFigures” property though it is only applicable when used with a decimal representation (i.e., within the “Quantity” class). This limits the total number of digits that can be included in the number represented whether a scientific notation is used or not.

Example: Examples

All non-zero digits are considered significant. 123.45 has five significant figures: 1, 2, 3, 4 and 5.

Zeros between two non-zero digits are significant. 101.12 has five significant figures: 1, 0, 1, 1 and 2.

Leading zeros are not significant. 0.00052 has two significant figures: 5 and 2 and is equivalent to 5.2×10^{-4} and would be valid even if the number of significant figures is restricted to 2.

Trailing zeros are significant. 12.2300 has six significant figures: 1, 2, 2, 3, 0 and 0 and would thus be invalid if the number of significant figures is restricted to 4.

NOTE: *The number of significant figures and/or an interval constraint (i.e., min/max values) can help a software implementation choosing the best data type to use (i.e., ‘float’ or ‘double’, ‘short’, ‘int’ or ‘long’) to store values associated to a given data component.*

8.2.19. AllowedTimes Class

The “AllowedTimes” class is used to express constraints on the value of a data component represented by a “Time” class. The UML class is shown below:

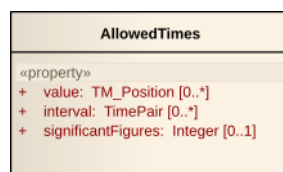


Figure 23 — AllowedTimes Class

This class is almost identical to the “AllowedValues” class and in fact all properties are used in the same way. The only difference with this class is that the “value” and “interval” properties allow the use of time data types as defined in Clause 8.2.1.

The constraints given by this class are expressed along the same time reference frame axis as the value attached to the parent data component.

8.2.20. Unions of simple component classes

Several useful groups of classes are also defined in this package. These unions can be used as attribute types and they are shown on the following diagram:

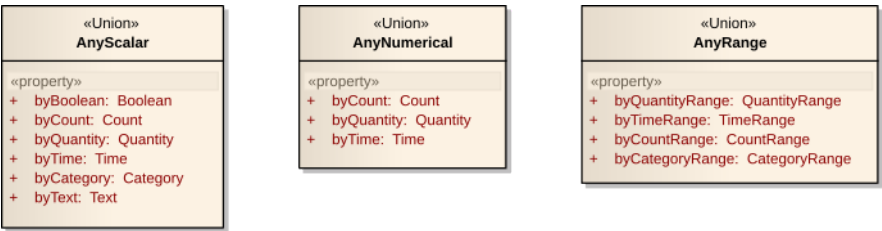


Figure 24 — Simple Component Unions

The “AnyScalar” union groups all classes representing scalar components, numerical or not. The “AnyNumerical” union includes all classes corresponding to numerical scalar representations. The “AnyRange” union regroupes all range components.

8.3. Requirements Class: Record Components Package

REQUIREMENTS CLASS 3: RECORD COMPONENTS UML PACKAGE	
IDENTIFIER	/req/uml-record-components
TARGET TYPE	Software Implementation or Encoding of the Conceptual Models
CONFORMANCE CLASS	Conformance class A.3: /conf/uml-record-components
PREREQUISITE	Requirements class 2: /req/uml-simple-components
NORMATIVE STATEMENTS	Requirement 36: /req/uml-record-components/package-fully-implemented Requirement 37: /req/uml-record-components/record-field-name-unique Requirement 38: /req/uml-record-components/vector-coord-name-unique Requirement 39: /req/uml-record-components/vector-component-no-ref-frame Requirement 40: /req/uml-record-components/vector-component-axis-defined

REQUIREMENTS CLASS 3: RECORD COMPONENTS UML PACKAGE

Requirement 41: /req/uml-record-components/vector-local-frame-valid

As detailed in the following clauses, this package defines classes modeling record style component types that can be nested to build complex structures from the simple component types introduced in Clause 8.2.

The classes defined in this package are “DataRecord” and “Vector” (other aggregates are defined in the “Choice Components” and “Block Components” packages defined in Clauses 8.4 and 8.5 respectively). The UML model is exposed below:

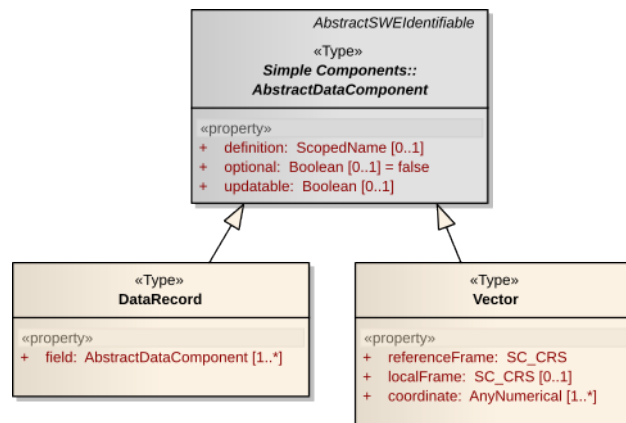


Figure 25 – Record Data Components

REQUIREMENT 36

IDENTIFIER /req/uml-record-components/package-fully-implemented

INCLUDED IN Requirements class 3: /req/uml-record-components

STATEMENT The encoding or software shall correctly implement all classes defined in the “Record Components” UML package.

As with simple component types, all data aggregates inherit attributes from the “AbstractDataComponent” class. In this case, however, these attributes provide information about the group as a whole rather than its individual components.

Example: Examples

A particular “DataRecord” might represent a standard collection of error codes coming from a GPS device.

A particular “Vector” might represent the linear or angular velocity vector of an aircraft.

In these two cases, the “definition” attribute should reference a semantic description in a registry, so that the data consumer knows what kind of data the aggregate represents. This semantic description can then be interpreted appropriately by consuming clients: for example to automatically decide how to style the data in visualization software.

8.3.1. DataRecord Class

The “DataRecord” class is modeled on the definition of ‘Record’ from ISO 11404. In this definition, a record is a composite data type composed of one to many fields, each of which having its own name and type definition. Thus it defines some logical collection of components of any type that are grouped for a given purpose.

As shown on the following figure, the “DataRecord” class in SWE Common is based on a full composite design pattern, such that each one of its “field” can be of a different type, including simple component types as well as any aggregate component type.

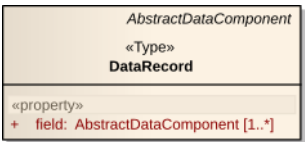


Figure 26 — DataRecord Class

The “DataRecord” class derives from the “AbstractDataComponent” class, which is necessary to enable the full composite pattern in which a “DataRecord” can be used to group scalar components, but also other records, arrays and choices recursively.

Fields

Each “field” attribute can take an instance of any concrete sub-class of “AbstractDataComponent”, which is the superset of all data component types defined in this standard. The name of each field must be unique within a given “DataRecord” instance so that it can be used as a key to uniquely identify and/or index each one of the record components.

REQUIREMENT 37	
IDENTIFIER	/req/uml-record-components/record-field-name-unique
INCLUDED IN	Requirements class 3: /req/uml-record-components
STATEMENT	Each “field” attribute in a given instance of the “DataRecord” class shall be identified by a name that is unique to this instance.

Example: Example

A “DataRecord” can group related values such as “temperature,” “pressure,” and “wind speed” into a structure called “weather measurements.” This feature is often used to organize the data and present it in a clear way to the user.

Similarly a “DataRecord” can be used to group values of several spectral bands in multi-spectral sensor data. However, using a “DataArray” may be easier to describe hyper spectral datasets with several hundreds of bands.

NOTE: The slightly different definition of record found in ISO 19103 provides for its schema to be specified in an associated “RecordType”. When used as a descriptor, the “DataRecord” implements the ISO 19103 “RecordType”. When used as a data container, it is self-describing: the descriptive information is then interleaved with the record values.

8.3.2. Vector Class

The “Vector” class is used to express multi-dimensional quantities with respect to a well defined referenced frame (usually a spatial or spatio-temporal reference frame). This is done by projecting the quantity on one or several axes that define the reference frame and assigning a value to each of the axis projections.

The “Vector” class is a special case of a record that takes a collection of coordinates that are restricted to a numerical representation. Coordinates of a “Vector” can thus only be of type “Quantity”, “Count,” or “Time.” Its UML diagram is shown below:

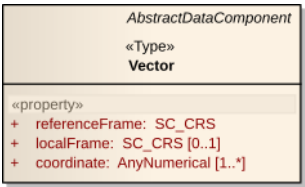


Figure 27 – Vector Class

Coordinates

Just like record fields, vector coordinates must have a unique name:

REQUIREMENT 38	
IDENTIFIER	/req/uml-record-components/vector-coord-name-unique
INCLUDED IN	Requirements class 3: /req/uml-record-components
STATEMENT	Each “coordinate” attribute in a given instance of the “Vector” class shall be identified by a name that is unique to this instance.

Reference Frame

This class contains a mandatory “referenceFrame” attribute that identifies the frame of reference with respect to which the vector quantity is expressed. The coordinates of the vector correspond to values projected on the axes of this frame.

The “referenceFrame” attribute is inherited by all components of the “Vector,” so that it shall not be redefined for each coordinate. However the “axisID” attribute shall be specified for each coordinate, in order to unambiguously indicate what axis of the reference frame it corresponds to.

REQUIREMENT 39

IDENTIFIER	/req/uml-record-components/vector-component-no-ref-frame
INCLUDED IN	Requirements class 3: /req/uml-record-components
STATEMENT	The “referenceFrame” attribute shall be omitted from all data components used to define coordinates of a “Vector” instance.

REQUIREMENT 40

IDENTIFIER	/req/uml-record-components/vector-component-axis-defined
INCLUDED IN	Requirements class 3: /req/uml-record-components
STATEMENT	The “axisID” attribute shall be specified on all data components used as children of a “Vector” instance.

Local Frame

The optional “localFrame” attribute allows identifying the frame of interest, that is to say the frame we are positioning with the coordinate values associated to this component (by opposition to the “referenceFrame” that specifies the frame with respect to which the values of the coordinates are expressed).

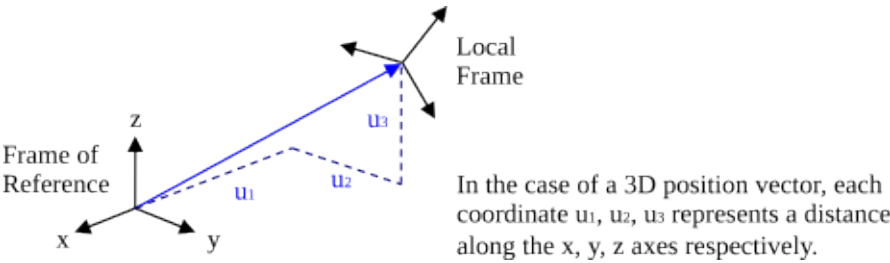
REQUIREMENT 41

IDENTIFIER	/req/uml-record-components/vector-local-frame-valid
INCLUDED IN	Requirements class 3: /req/uml-record-components
STATEMENT	The “localFrame” attribute of an instance of the “Vector” class shall have a different value than the “referenceFrame” attribute.

Correctly identifying the local and reference frame is an important feature that allows chaining several relative positions, something that is essential to correctly compute accurate position of sensor data (especially remote sensing data).

Example: Example 1

A position vector (or location vector) is used to locate the origin of a frame of interest (the local frame) relative to the origin of a frame of reference (the reference frame) through a linear translation. It is composed of three coordinates of type “Quantity”, each with a definition indicating that the coordinate represents a length expressed in the desired unit. The definition of the “Vector” itself should also indicate that it is a “location vector.”



Example 2

An orientation vector is used to indicate the rotation of the axes of a frame of interest (the local frame) relative to a frame of reference (the reference frame). It is composed of three coordinates of type “Quantity” with a definition indicating an angular property. The “Vector” definition should indicate the type of orientation vector such as “Euler Angles” or “Quaternion”. Depending on the exact definition, the order in which the coordinates are listed in the vector may matter.

NOTE: “Vector” aggregates are most commonly used to describe position, orientation, velocity, and acceleration within temporal and spatial domains, but can also be used to express relationships between any two coordinate frames.

8.4. Requirements Class: Choice Components Package

REQUIREMENTS CLASS 4: CHOICE COMPONENTS UML PACKAGE	
IDENTIFIER	/req/uml-choice-components
TARGET TYPE	Software Implementation or Encoding of the Conceptual Models
CONFORMANCE CLASS	Conformance class A.4: /conf/uml-choice-components
PREREQUISITE	Requirements class 2: /req/uml-simple-components
NORMATIVE STATEMENTS	Requirement 42: /req/uml-choice-components/package-fully-implemented Requirement 43: /req/uml-choice-components/choice-item-name-unique

As detailed in the following clauses, this package defines a class modeling a disjoint union component type. This aggregate type can be nested with other aggregate components to build complex structures.

REQUIREMENT 42	
IDENTIFIER	/req/uml-choice-components/package-fully-implemented
INCLUDED IN	Requirements class 4: /req/uml-choice-components
STATEMENT	The encoding or software shall correctly implement all classes defined in the “Choice Components” UML package.

8.4.1. DataChoice Class

The “DataChoice” class (also called Disjoint Union) is modeled on the definition of ‘Choice’ from ISO 11404. It is a composite component that allows for a choice of child components. By opposition to records that carry all their fields simultaneously, only one item at a time can be present in the data when wrapped in a “DataChoice”. The following diagram shows the “DataChoice” class as implemented in the SWE Common Data Model:

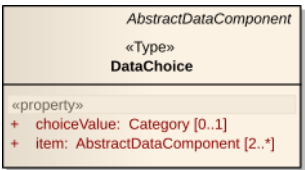


Figure 28 – DataChoice Class

This class implements a full composite pattern, so that each “item” can be any data component, including simple and aggregate types.

The “choiceValue” attribute is used to represent the token value that would be present in the data stream and that indicates the actual choice selection before the corresponding data can be given (i.e., knowing what choice item was selected ahead of time is necessary for proper decoding of encoded data streams).

Items

Each “item” attribute can take an instance of any concrete sub-class of “AbstractDataComponent,” which is the superset of all data component types defined in this standard. The name of each item shall be unique within a given “DataChoice” instance so that it can be used as a key to uniquely identify and/or index each one of the choice components.

REQUIREMENT 43

IDENTIFIER /req/uml-choice-components/choice-item-name-unique

INCLUDED IN Requirements class 4: /req/uml-choice-components

STATEMENT Each “item” attribute in a given instance of the “DataChoice” class shall be identified by a name that is unique to this instance.

The “DataChoice” component is used to describe a data structure (or a part of the structure) that can alternatively contain different types of objects. It can also be used to define the input of a service or process that allows a choice of structures as its input.

Example: Examples

NMEA 0183 compatible devices can output several types of sentences in the same data stream. Some sentences include GPS location, while some others contain heading or status data. This can be described by a “DataChoice” which items represent all the possible types of sentences output by the device.

A Sensor Planning Service (SPS) can define a choice in the tasking messages that the service can accept, thus leaving more possibilities to the users.

8.5. Requirements Class: Block Components Package

REQUIREMENTS CLASS 5: BLOCK COMPONENTS UML PACKAGE

IDENTIFIER /req/uml-block-components

TARGET TYPE Software Implementation or Encoding of the Conceptual Models

CONFORMANCE CLASS Conformance class A.5: /conf/uml-block-components

PREREQUISITES Requirements class 2: /req/uml-simple-components
Requirements class 7: /req/uml-simple-encodings

NORMATIVE STATEMENTS Requirement 44: /req/uml-block-components/package-fully-implemented
Requirement 45: /req/uml-block-components/array-component-no-value
Requirement 46: /req/uml-block-components/array-values-properly-encoded
Requirement 47: /req/uml-block-components/matrix-element-type-valid

REQUIREMENTS CLASS 5: BLOCK COMPONENTS UML PACKAGE

Requirement 48: /req/uml-block-components/datastream-array-valid

This package defines additional aggregate components for describing arrays of values that are designed to be encoded as efficient data blocks. These additional aggregate components are purposely defined in a separate requirement class because they require a more advanced implementation for handling data values as encoded blocks.

The UML models for these additional aggregate components are shown below:

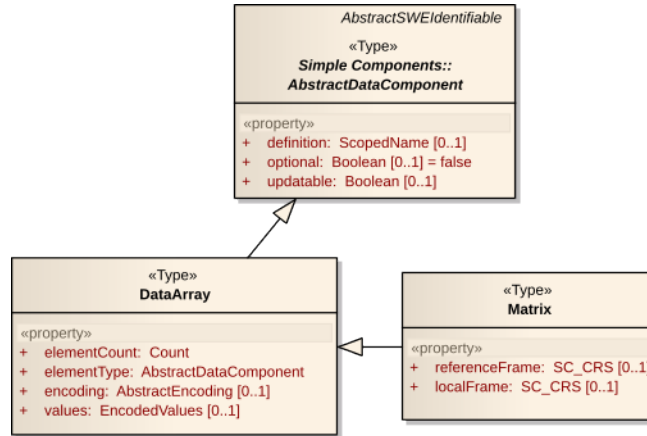


Figure 29 – Array Components

REQUIREMENT 44

IDENTIFIER /req/uml-block-components/package-fully-implemented

INCLUDED IN Requirements class 5: /req/uml-block-components

STATEMENT The encoding or software shall correctly implement all classes defined in the “Block Components” UML package.

The principle of these two classes is that the number and type of elements contained in the array is defined once, while the actual array values are listed separately without being redefined with each value. In order to achieve this, all array values are encoded as a single data block in the “values” attribute. Consequently, these classes are restricted to cases where all elements are homogeneous and thus can be described only once even though the array data may in fact contain many of them.

This package also defines the “DataStream” class that is similar in principle to the “DataArray” class but cannot be nested within other aggregate data components. It is a top level class that encapsulates the description of a full data stream.

8.5.1. DataArray Class

The “DataArray” class is modeled on the corresponding definition of ISO 11404. This definition states that an array is a collection of elements of the same type (as opposed to a record where each field can have a different type), with a defined size. This class is shown on the following UML diagram:

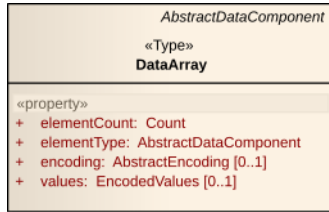


Figure 30 – DataArray Class

This class implements a full composite pattern, so that the “elementType” can be any data component, including simple and aggregate types. It can be used to group identical scalar components as well as records, choices and arrays in a recursive manner.

Element Count

The “elementCount” attribute is used to indicate the size of the array, that is to say the number of elements of the given type in the array. Note that each element is not necessarily scalar but can be a record, another array, etc.

Element Type

The content of the “elementType” attribute defines the exact structure of each element in the array. The data component used and all of its children shall not include any inline values, as these will be block encoded in the “values” attribute of the parent “DataArray.”

REQUIREMENT 45

IDENTIFIER /req/uml-block-components/array-component-no-value

INCLUDED IN Requirements class 5: /req/uml-block-components

STATEMENT Data components that are children of an instance of a block component shall be used solely as data descriptors. Their values shall be block encoded in the “values” attribute of the block component rather than included inline.

However, the “DataArray” class itself, like any other data component can be used either as a data descriptor or as a data container. To use it as a data descriptor the “encoding” and “values” attributes are not set. To use it as a data container, these attributes are both set as described below.

Encoding and Values

The “encoding” and “values” fields are there to provide array data as an efficient block which can be encoded in several ways. The different encoding methods are described in Clauses 8.7 and 8.8. The “encoding” field shall have a value if the “values” field is present, and the data shall be encoded using the specified encoding.

REQUIREMENT 46

IDENTIFIER /req/uml-block-components/array-values-properly-encoded

INCLUDED IN Requirements class 5: /req/uml-block-components

STATEMENT Whenever an instance of a block component contains values, an encoding method shall be specified by the “encoding” property and array values shall be encoded as specified by this method.

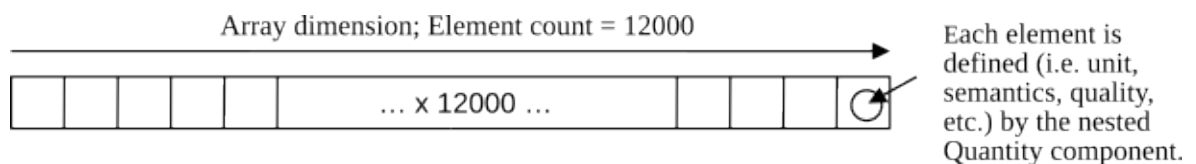
The choice of simple encodings (defined in the “Simple Encodings” package) allows encoding data as JSON or as text using a delimiter separated values (DSV, a variant of CSV) format. The “Advanced Encodings” package defines binary encodings that can be used to efficiently package large datasets.

Nested Components

By combining instances of “DataArray”, “DataRecord” and scalar components, one can obtain the complex data structures that are necessary to fully describe any kind of sensor data. In particular, the possibility of nesting a “DataRecord” or “Vector” inside a “DataArray” allows defining structures such as trajectories, profiles, multi-band images, etc.

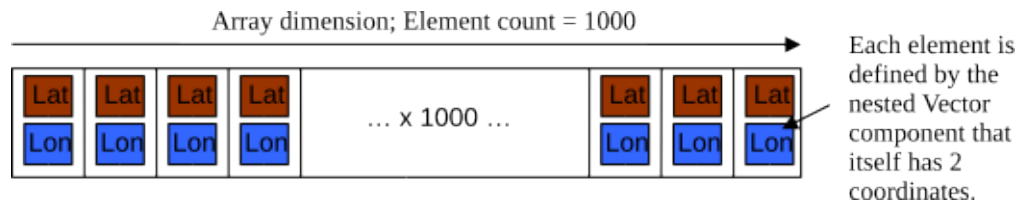
Example: Example 1

The “DataArray” class can be used to describe a simple 1D array of measurements such as radiance values obtained using a 12000 cells (1 row) CCD strip for instance. This can be done by using the “Quantity” class as the element type. In such a case, describing the dataset as a “DataRecord” would be a very repetitive task given the number of elements (12000 in this case!).



Example 2

The “DataArray” class can be used as a descriptor for a trajectory dataset by using a vector of [latitude, longitude] coordinates as its element type. Note that this can also be considered as a 1D coverage in a 2D CRS.

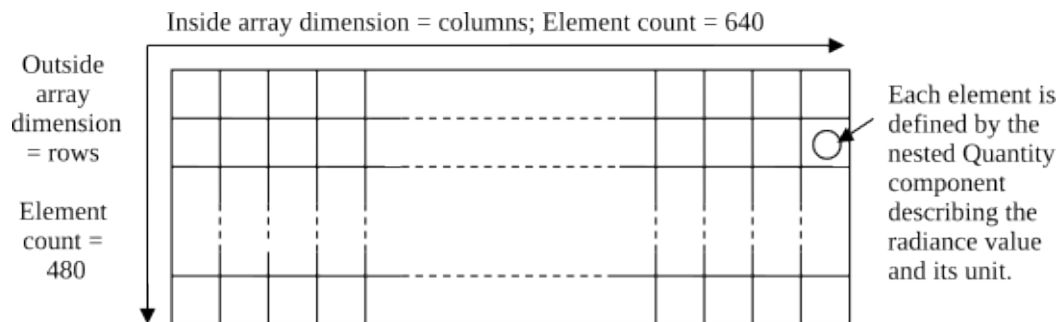


Multi-dimensional Arrays

Since the “DataArray” class alone can only represent 1-dimensional arrays, the construction of multi-dimensional arrays is done by nesting “DataArray” objects inside each other.

Example: Example

The structure of panchromatic imagery data can be described with two nested arrays, which sizes indicate the two dimensions of the image. A “Quantity” is used as the element type of the nested array in order to indicate that the repeated element of the 2D array is of type infrared radiance with a given unit.



In this example, the image is described as an array of rows, each row being an array of samples. It is also possible to describe an image as an array of columns by reversing the two dimensions. Note that this would change the order in which the data values would appear in a stream (by rows vs. by columns).

Array Size

One powerful feature of the “DataArray” model is that it allows for the element count to be either fixed or variable, thus allowing the description of data streams with variable number of repetitive elements as is often the case with many kinds of sensor.

In a fixed size array, the number of elements can be provided in the descriptor as an instance of the “Count” class with an inline value. This value is only present in the data description and not in the encoded block of array values. The definition of the “Count” instance is not required.

In a variable size array, the “elementCount” attribute either contains an instance of the “Count” class with no value or references an instance of a “Count” class in a parent or sibling data component. The value giving the actual array size is then included in the stream, before the array values themselves, so that the block can be properly decoded. One obvious implementation constraint is that the value representing the array size must be received before the array values. This is detailed further in the JSON implementation section.

Example: Examples

Argo profiling floats can measure ocean salinity and temperature profiles of variable lengths by diving at different depths and depending on the conditions. A variable size “DataArray” could be used to describe their output data as well as a dataset aggregating data from several Argo floats.

Variable size arrays can often be used to avoid unnecessary padding of fixed size array data. However for efficiency reasons (usually to enable fast random access w/o preliminary indexation), padding can also be specified in SWE Common when using the binary encoding.

Array Semantics

As with any other data component, the “name” and “description” can be used to better describe the array and more importantly the “definition” attribute can be used to formally indicate the semantics behind the array.

Example: Example

When a “DataArray” is used to package data relative to the spectral response of a sensor, the array “definition” attribute can be used to point to the formal out-of-band definition of the “spectral response” concept.

Similarly a “DataArray” used to describe the output data of an Argo float would have its “definition” attribute reference the formal definition of a “profile.”

The value of the “definition” attribute of the “Count” instance used as the “elementCount” is also especially important, since it is used to define the meaning of the array dimension. Thanks to this, it is possible to tag the dimension of an array as spatial, temporal, spectral, or any other kind. However it is not mandatory as it is on other simple components.

Example: Examples

In the CCD strip example described as a 1D array, the array index is the cell number in the strip.

In the 2D image example, the outer array index is the row number, while the inner array index is the column (or sample) number.

In a 1D array representing a time series, the array index is along the temporal dimension.

In a 2D array representing a spatial coverage, the two array indices are along spatial dimensions.

In a 3D array representing hyper-spectral imagery, the two first arrays have indices along spatial dimension while the most inner array is indexed along the spectral dimension.

This extra information can be used by software to make decisions (or at least ask the user by providing him this information) about how to represent or even interpolate the data.

8.5.2. Matrix Class

The “Matrix” extends the “DataArray” class by providing a reference frame within which the matrix elements are expressed and a local frame of interest. The UML diagram of this class is shown below:

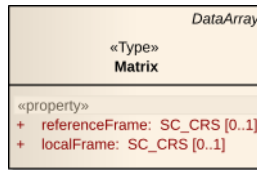


Figure 31 – Matrix Class

The “Matrix” class is usually used to represent a position matrix or a tensor quantity of second or higher order. Each matrix element is expressed along the axis of a well defined reference frame.

Element Type

The “elementType” attribute inherited from the “DataArray” class can only take a nested “Matrix” instance or a scalar numerical component. Nested matrix objects allow the full description of N-dimensional matrices.

REQUIREMENT 47

IDENTIFIER /req/uml-block-components/matrix-element-type-valid

INCLUDED IN Requirements class 5: /req/uml-block-components

STATEMENT The “elementType” attribute of an instance of the “Matrix” class can only be an instance of “Matrix” or of the classes listed in the “AnyNumerical” union.

Reference Frame

The “referenceFrame” attribute is used in the same way as with the “Vector” class to specify the frame of reference with respect to which the matrix element values are expressed. It is inherited by all child components.

Local Frame

The “localFrame” attribute is used to identify the frame of interest, that is to say the frame whose orientation or position is given with the matrix in the case where it is a position matrix. If the matrix does not specify position, “localFrame” should not be used. Whether an instance of the “Matrix” class represents a position matrix or not should be disambiguated by setting the value of its “definition” attribute.

Example: Examples

The “Matrix” class can be used to represent for instance: - A 3D 3×3 stress tensor - A 4D 4×4 homogeneous affine transformation matrix

In particular it is often used to specify the orientation of an object relative to another one, like for instance the attitude of a plane relative to the earth.

8.5.3. DataStream Class

The “DataStream” class has a structure similar than the “DataArray” class but is not a data component (i.e., it does not derive from “AbstractDataComponent”) and thus cannot be used as a child of other aggregate components. Below is its UML diagram:

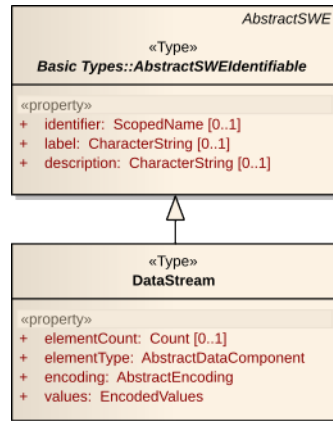


Figure 32 — DataStream Class

This class should be used as the wrapper object to define a complete data stream. It defines a data stream as containing a list of elements with an arbitrary complex structure. An important feature is that the data stream can be open ended (i.e., the number of elements is not known in advance) and is thus designed to support real time streaming of data.

Element Count

The “elementCount” attribute is optional and can be used to indicate the number of elements in the stream if it is known. This is done by instantiating an instance of the “Count” class whose “value” attribute would be set to the number of elements.

Element Type

The “elementType” attribute is used to define the structure of each element in the stream. The data component used as the element type and all of its children shall be used solely as data descriptors, meaning that they shall not include any inline values. These values will instead be block encoded in the “values” attribute of the parent “DataStream”.

Encoding and Values

The “encoding” and “values” fields are there to provide the stream values as an efficient block which can be encoded in several ways. The same encoding methods as for the “DataArray” class are available and are described in Clauses 8.7 and 8.8. The “values” attribute is optional as the DataStream class can be used as a simple descriptor.

REQUIREMENT 48

IDENTIFIER /req/uml-block-components/datastream-array-valid

INCLUDED IN Requirements class 5: /req/uml-block-components

STATEMENT Data components that are children of an instance of the "DataStream" class shall be used solely used as data descriptors. Their values shall never be included inline since they will be block encoded in the stream described by the "DataStream".

8.6. Requirements Class: Geometry Components Package

REQUIREMENTS CLASS 6: GEOMETRY COMPONENTS UML PACKAGE

IDENTIFIER /req/uml-geom-components

TARGET TYPE Software Implementation or Encoding of the Conceptual Models

CONFORMANCE CLASS Conformance class A.6: /conf/uml-geom-components

PREREQUISITE Requirements class 2: /req/uml-simple-components

NORMATIVE STATEMENTS Requirement 49: /req/uml-geom-components/package-fully-implemented
Requirement 50: /req/uml-geom-components/srs-valid
Requirement 51: /req/uml-geom-components/geom-value-valid

This package defines an additional component for representing simple feature geometries, as defined by OGC 06-103r4, within an encoded SWE Common data block or stream.

REQUIREMENT 49

IDENTIFIER /req/uml-geom-components/package-fully-implemented

INCLUDED IN Requirements class 6: /req/uml-geom-components

STATEMENT The encoding or software shall correctly implement all classes defined in the "Geometry Components" UML package.

8.6.1. Geometry Class

The “Geometry” class extends the “AbstractDataComponent” class with a value of type geometry and a constraint that can be used to limit the types of allowed geometries. This class is shown on the following UML diagram:

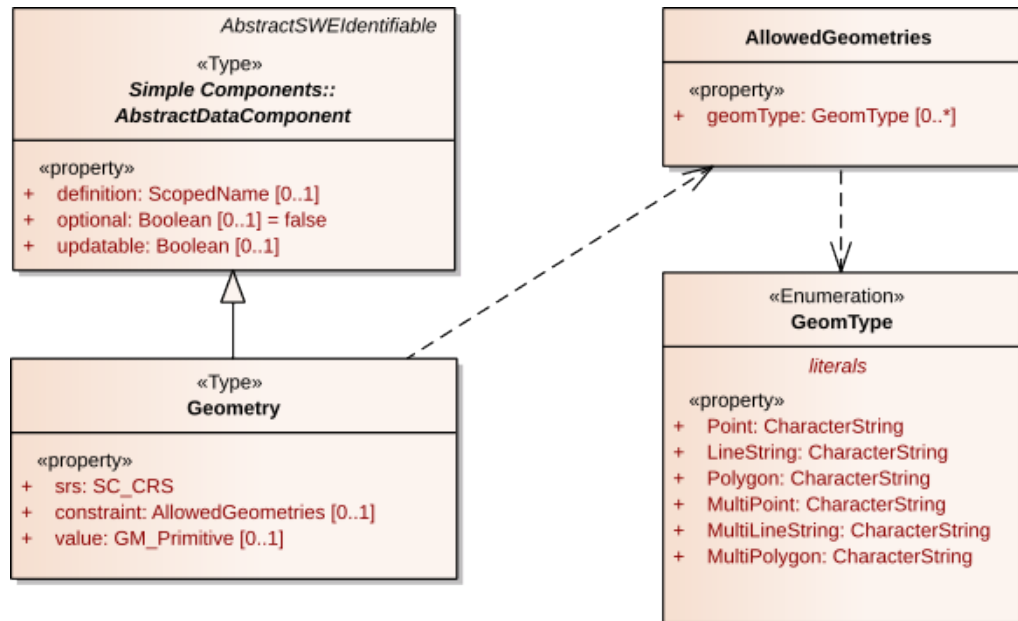


Figure 33 – Geometry Class

Coordinate Reference System

The “crs” attribute provides the URI of the coordinate reference system with respect to which the geometry coordinates are expressed. The unit of the coordinates is also provided by the coordinate reference system.

REQUIREMENT 50

IDENTIFIER /req/uml-geom-components/srs-valid

INCLUDED IN Requirements class 6: /req/uml-geom-components

STATEMENT The “srs” attribute shall reference the definition of a valid 2D or 3D spatial reference system.

Constraints

The “constraint” attribute is used to restrict the possible geometries that can be provided using this component when it is used as a descriptor. The constraint is provided using the “AllowedGeometries” class that includes a list of allowed geometry types.

Value

The value of this component must be a geometry instance, whether it's provided inline using the "value" attribute, or as part of a datastream.

REQUIREMENT 51

IDENTIFIER /req/uml-geom-components/geom-value-valid

INCLUDED IN Requirements class 6: /req/uml-geom-components

STATEMENT The "value" attribute shall be one of the concrete geometry value classes defined in OGC 06-103r4: "Point", "MultiPoint", "LineString", "MultiLineString", "Polygon", or "MultiPolygon".

NOTE: Encoding sections in this standard define how the geometry value is encoded:

- GeoJSON in the JSON implementation and JSON encoding rules
- WKT in the Text encoding rules
- WKB in the Binary encoding rules

8.7. Requirements Class: Simple Encodings Package

REQUIREMENTS CLASS 7: SIMPLE ENCODINGS UML PACKAGE

IDENTIFIER /req/uml-simple-encodings

TARGET TYPE Software Implementation or Encoding of the Conceptual Models

CONFORMANCE CLASS Conformance class A.7: /conf/uml-simple-encodings

PREREQUISITE Requirements class 1: /req/core

NORMATIVE STATEMENT Requirement 52: /req/uml-simple-encodings/package-fully-implemented

Encoding methods describe how structured array and stream data is encoded into a low level byte stream (see related concepts in Clause 7.6). Once encoded as a sequence of bytes, the data can then be transmitted using various digital means such as files on a disk or network connections.

This package includes two classes that provide definitions of simple encoding methods. They are used as descriptors of the method used to encode data component values wrapped by

aggregate classes defined in the “Block Components” package. There model is shown on the diagram below:

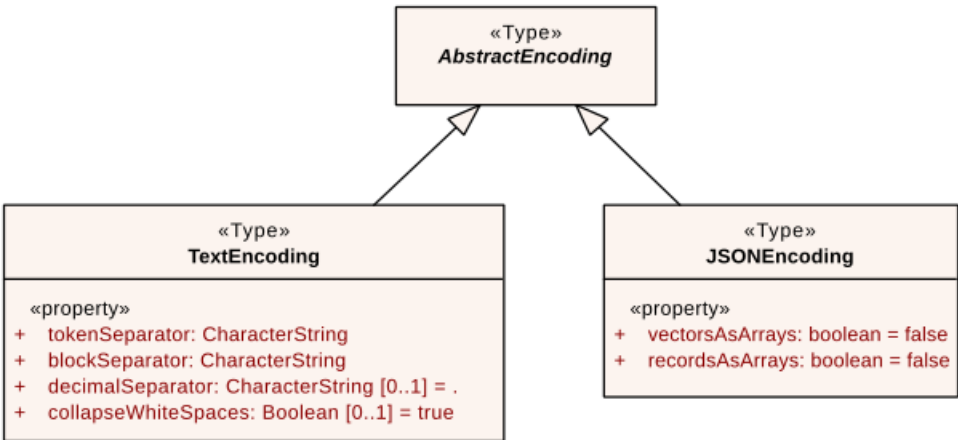


Figure 34 — Simple Encodings

Requirement 52	
IDENTIFIER	/req/uml-simple-encodings/package-fully-implemented
INCLUDED IN	Requirements class 7: /req/uml-simple-encodings
STATEMENT	The encoding or software shall correctly implement all classes defined in the “Simple Encodings” UML package.

All classes defining encoding methods derive from a common abstract class called “AbstractEncoding.” Extensions to this standard that define new encoding methods shall derive encoding classes from this abstract class.

The intent of this standard is to provide a set of core encodings covering most common needs. Each encoding has specific benefits that match the needs of different applications. Sometimes several encodings of the same dataset can be offered in order to satisfy several types of consumers and/or use cases.

In the model provided in this standard, the encoding specification is provided separately from the data component tree describing the dataset structure, thus enabling several encodings to be applied to the same data structure without changing it.

8.7.1. JSONEncoding Class

The “JSONEncoding” class defines a method allowing encoding arbitrarily complex data as JSON. The class used to specify this encoding method is shown in Figure 34.

The “recordsAsArrays” attribute specifies whether “DataRecord” values are encoded as JSON objects or JSON arrays. The “vectorsAsArrays” attribute specifies whether “Vector” values are encoded as JSON objects or JSON arrays. Both attributes are optional and default to false, meaning “DataRecord” and “Vector” values are per default encoded as JSON objects. The detailed rules are given in the implementation in Clause 10.2.

8.7.2. TextEncoding Class

The “TextEncoding” class defines a method allowing encoding arbitrarily complex data using a text based delimiter separated values (DSV) format. The class used to specify this encoding method is shown below:

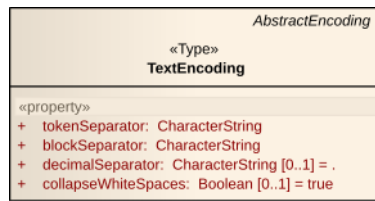


Figure 35 — TextEncoding Class

The “tokenSeparator” attribute specifies the characters to use for separating each scalar value from one another. Scalar values appear sequentially in the stream alternatively with the token separator characters, in an order unambiguously defined by the data component structure. The detailed rules are given in the implementation in Clause 10.3.

The “blockSeparator” attribute specifies characters used to mark the end of a “block”, corresponding to the complete structure defined by the data component tree (in a “DataArray”, “Matrix” or “DataStream” one block corresponds to one element, that is to say the structure defined by the “elementType” property). Stream or array data can then be composed of several blocks of the same type separated by block separator characters.

The “decimalSeparator” attribute specifies the character used as the decimal point in decimal number. This attribute is optional and the default is a period (‘.’).

Example: Example

In the case of a “DataStream” with an element type that is a “DataRecord” containing three fields – one of type “Category” and two of type “Quantity” – a data stream encoded using the Text method would look like the following:

```

STATUS_OK,24.5,1022.5¶
STATUS_OK,24.5,1022.5¶
STATUS_OK,24.5,1022.5¶
  
```

Where , (comma) is the token separator and ¶ (carriage return) is the block separator (i.e., this is the CSV format). Note that there could be many more values in a single block if the data set has a large number of fields, or if it contains an array of values.

The “collapseWhiteSpaces” attribute is a boolean flag used to specify if extra white spaces (including line feeds, tabs, spaces and carriage returns) surrounding the token and block separators should be ignored (skipped) when processing the stream.

This type of encoding is used when compactness is important but balanced by a desire of human readability. This type of encoding is easily readable (for debugging or manual usage) as well as easily imported in various spreadsheet, charting or scientific software.

The main drawback of such an encoding is the impossibility of locating an error in the stream with certitude. Secondly, if only one expected value is missing, the whole block is usually lost since the parser cannot resynchronize correctly before the next block separator. This last issue can however be solved by transmitting this type of encoded stream using error resilient protocols when needed.

8.8. Requirements Class: Advanced Encodings Package

REQUIREMENTS CLASS 8: ADVANCED ENCODINGS UML PACKAGE	
IDENTIFIER	/req/uml-advanced-encodings
TARGET TYPE	Software Implementation or Encoding of the Conceptual Models
CONFORMANCE CLASS	Conformance class A.8: /conf/uml-advanced-encodings
PREREQUISITE	Requirements class 7: /req/uml-simple-encodings
NORMATIVE STATEMENT	Requirement 53: /req/uml-advanced-encodings/package-fully-implemented

This package defines an additional encoding method for packaging sensor data as raw or base 64 binary blocks. When this package is implemented, the binary encoding method is usable, as any other encoding method, within the “DataArray” and “DataStream” classes.

REQUIREMENT 53	
IDENTIFIER	/req/uml-advanced-encodings/package-fully-implemented
INCLUDED IN	Requirements class 8: /req/uml-advanced-encodings
STATEMENT	The encoding or software shall correctly implement all classes defined in the “Advanced Encodings” UML package.

8.8.1. BinaryEncoding Class

The “BinaryEncoding” class defines a method that allows encoding complex structured data using primitive data types encoded directly at the byte level, in the same way that they are usually represented in memory.

The binary encoding method can lead to very compact streams that can be optimized for efficient parsing and fast random access. However this comes with the lack of human readability of the data and sometimes lack of compatibility with other software (i.e., software that is not SWE Common enabled).

More information is needed to fully define a binary encoding, so the model is more complex than the other encodings. It is shown below:

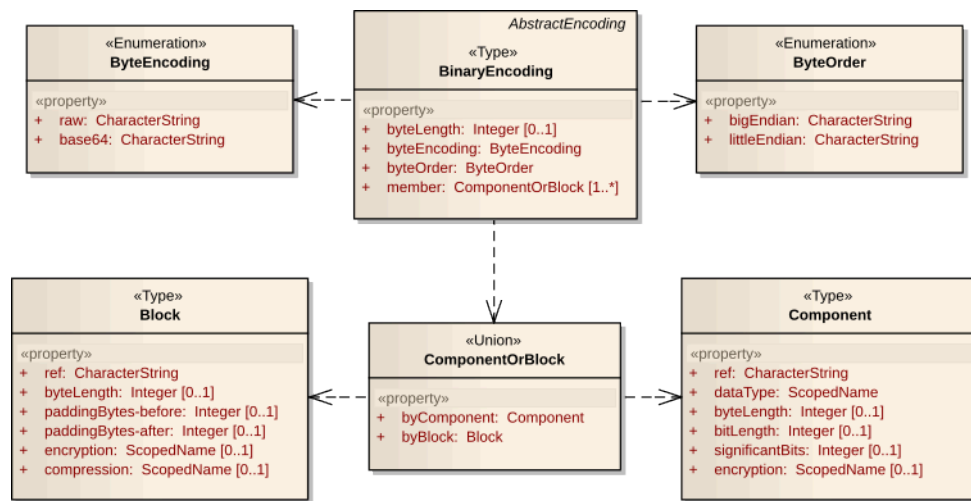


Figure 36 – BinaryEncoding Class

The main class “BinaryEncoding” specifies overall characteristics of the encoded byte stream such as the byte order (big endian or little endian) and the byte encoding (raw or base64). The two corresponding attributes, respectively “byteOrder” and “byteEncoding” are mandatory. Base64 encoding is usually chosen to insert binary content within a JSON or XML document.

The “byteLength” attribute is optional and can be used to specify the overall length of the encoded data as a total number of bytes. This should be indicated whenever possible if the data size is known in advance as it can be useful for efficient memory allocation.

The “BinaryEncoding” class also has several “member” attributes that contain detailed information about parts of the data stream. This attribute can take a choice of instance of two classes: “Component” or “Block.”

The “Component” class is used to specify binary encoding details of a given scalar component in the stream. The following information can be provided for each scalar field:

- The “ref” attribute allows identifying the data component in the dataset structure for which we’re specifying the encoding parameters. Soft-typed property names are used to uniquely identify a given component in the tree.
- The “dataType” attribute allows selecting a data type among commonly accepted ones such as ‘byte’, ‘short’, ‘int’, ‘long’, ‘double’, ‘float’, ‘string’, etc...
- The “byteLength” or “bitLength” attributes are mutually exclusive and used to further specify the length of the data type in the case where it is not a standard length (i.e., to encode integer numbers on more than 8 bytes or less than 8 bits for instance).
- The “significantBits” can be used to signal that only some of the bits of the data type are actually used to carry the value (i.e., a value may be encoded as a byte but only use 4 bits to encode a value between 0 and 15). This is mostly informational.
- The “encryption” attribute can be used to select the method with which the value is encrypted before being written to the stream.

The “Block” class is used to specify binary encoding details of a given aggregate component representing a block of values in the data stream. This is used either to specify padding before and/or after a block of data or to enable compression or encryption of all or part of a dataset.

- The “ref” attribute allows identifying the data component in the dataset structure for which we’re specifying the encoding parameters. Soft-typed property names are used to uniquely identify a given component in the tree.
- The optional “byteLength” attribute allows indicating the overall length of the encoded block to facilitate memory allocation.
- The “paddingBytes-before” and “paddingBytes-after” are used to specify the number of empty bytes (i.e., usually 0 bytes) that are inserted in the stream respectively before and after data for the referenced component. This is sometimes used to align data on N-bytes block for faster access.
- The “encryption” attribute identifies the encryption method that is used to encrypt the block of data before it is inserted in the stream.
- The “compression” attribute identifies the compression method that is used to compress the block of data before it is inserted in the stream.

This standard does not define any concrete encryption and compression methods, so that software implementations of this requirement class are not required to support any value in the “encryption” and “compression” attributes of the “Component” and “Block” classes. Extensions of this standard that define binary encryption and compression methods shall describe how the encrypted or compressed data is inserted in the SWE Common data stream.



9

JSON IMPLEMENTATION (NORMATIVE)

JSON IMPLEMENTATION (NORMATIVE)

This standard defines a normative JSON implementation of the conceptual models presented in Clause 8. The standardization target type for all requirements classes in this clause is a JSON instance document that seeks compliance with this JSON encoding model.

JSON schemas defined in this section are a direct implementation of the UML conceptual models described in Clause 8. They have been generated from these models by strictly following well-defined encoding rules. All attributes and composition/aggregation associations contained in the UML models are encoded as JSON object members.

All JSON examples given in this section are informative and are used solely for illustrating features of the normative model. Many of these examples reference semantic information by using URLs that resolve to the following online ontologies:

- The OGC online registry at <http://www.opengis.net/def/>;
- The QUDT quantity kinds ontology at <http://qudt.org/2.1/vocab/quantitykind>;
- The SWEET ontology maintained by ESIP at <http://sweetontology.net/>; and
- The MMI ontology registry and repository at <http://mmisw.org/ont/>.

Some of the JSON examples contain inline values while others don't. This is meant to illustrate that the component objects defined by the JSON implementation can be used as value objects for properties of larger metadata objects (e.g., SensorML system descriptions), but can also be used as descriptors to describe, for instance, the content of a datastream or the ranges of a coverage.

9.1. Requirements Class: Basic Types and Simple Components JSON Schemas

REQUIREMENTS CLASS 9: BASIC TYPES AND SIMPLE COMPONENTS JSON SCHEMAS	
IDENTIFIER	/req/json-simple-components
TARGET TYPE	JSON Document
CONFORMANCE CLASS	Conformance class A.9: /conf/json-simple-components
INDIRECT PREREQUISITE	Requirements class 2: /req/uml-simple-components

REQUIREMENTS CLASS 9: BASIC TYPES AND SIMPLE COMPONENTS JSON SCHEMAS

NORMATIVE STATEMENTS

Requirement 54: /req/json-simple-components/
component-types
Requirement 55: /req/json-simple-components/
schema-valid
Requirement 56: /req/json-simple-components/
special-numerical-values
Requirement 57: /req/json-simple-components/
definition-resolvable
Requirement 58: /req/json-simple-components/
inline-value-constraint-valid
Requirement 59: /req/json-simple-components/
ucum-code-and-href-represent-same-unit
Requirement 60: /req/json-simple-components/
iso8601-uom-used

Validation patterns that implement all classes defined respectively in the “Basic Types” and “Simple Components” UML packages are provided as JSON schema files at <https://schemas.opengis.net/sweCommon/3.0/json>.

The entry point schema used for validation is “[sweCommon.json](#)”.

REQUIREMENT 54

IDENTIFIER /req/json-simple-components/component-types

INCLUDED IN Requirements class 9: /req/json-simple-components

STATEMENT The standardization target SHALL implement the following data component types:
Boolean, Text, Category, Count, Quantity, Time, CategoryRange, CountRange, QuantityRange, TimeRange

REQUIREMENT 55

IDENTIFIER /req/json-simple-components/schema-valid

INCLUDED IN Requirements class 9: /req/json-simple-components

STATEMENT The JSON document instance shall be valid with respect to the JSON schema “[sweCommon.json](#)”.

9.1.1. General JSON Principles

The following rules were used when generating the JSON schemas:

- Classes are implemented as JSON Objects;
- Any property with a multiplicity greater than one is implemented as a JSON Array and its name is converted to plural form;
- Textual fields are implemented as a JSON String;
- Decimal fields are implemented as a union of JSON Number and JSON String value types (the string value allowing for special values, see Clause 9.1.2); and
- ISO8601 date/time fields are implemented as a JSON String with a union of date/time formats.

9.1.2. Special Numerical Values

JSON does not define special decimal values for ‘Not a Number’, positive infinity and negative infinity. It is thus necessary to encode them as strings.

REQUIREMENT 56

IDENTIFIER /req/json-simple-components/special-numerical-values

INCLUDED IN Requirements class 9: /req/json-simple-components

STATEMENT The special JSON Strings NaN, -Infinity and +Infinity shall be allowed as the inline or out-of-band value for Quantity and Time components (except when the Time component uses the ISO 8601 format).

NOTE: These special value strings have been chosen because they are supported natively by Javascript/ECMA Script implementations. The + unary operator can be used to transparently parse one of these strings to a Number type (see <https://262.ecma-international.org/13.0/#sec-unary-plus-operator>).

These values also correspond to infinities and NaN values defined in IEEE 754™-2008.

9.1.3. Abstract Base Classes

The three abstract base classes defined in the UML models are implemented by the following JSON schemas:

- [AbstractSweldentifiable.json](#)
- [AbstractDataComponent.json](#)
- [AbstractSimpleComponent.json](#)

REQUIREMENT 57

IDENTIFIER /req/json-simple-components/definition-resolvable

INCLUDED IN Requirements class 9: /req/json-simple-components

STATEMENT The “definition” object member defined in the “AbstractDataComponent.json” schema shall contain a URI that can be resolved to the complete human readable definition of the property that is represented by the data component.

REQUIREMENT 58

IDENTIFIER /req/json-simple-components/inline-value-constraint-valid

INCLUDED IN Requirements class 9: /req/json-simple-components

STATEMENT The inline value included in a JSON instance validating against the “AbstractSimpleComponent.json” schema shall satisfy the constraints specified by this instance.

9.1.4. Unit Reference Object

The “UnitReference” object is the partial JSON schema implementation of the “UnitOfMeasure” UML class used in Clause 8.2.8 and Clause 8.2.13, and the “UomTime” UML class used in Clause 8.2.9 and Clause 8.2.14, as defining the model for “UnitOfMeasure” and “UomTime” are out of scope for SWE Common. SWE Common only references implementations of “UnitOfMeasure”, like for instance the ones provided by UCUM.

The schema for this class is provided in [basicTypes.json](#) (see #definitions/UnitReference). Further details on the usage of “UnitReference” object can be found in Clause 9.1.9 and Clause 9.1.10.

Examples of references to a unit of measure are provided below:

```
{
  "code": "W.m-2.Sr-1.um-1"
}
```

Listing

```
{
  "href": "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian"
}
```

Listing

```
{
  "symbol": "°C",
  "code": "Cel",
  "href": "https://qudt.org/vocab/unit/DEG_C"
}
```



```
}
```

Listing

9.1.5. Boolean Object

The “Boolean” object is the JSON schema implementation of the “Boolean” UML class defined in Clause 8.2.4. The schema for this class is provided in [Boolean.json](#).

The following snippet shows an example boolean component with an inline value:

```
{
  "type": "Boolean",
  "definition": "http://sweet.jpl.nasa.gov/2.0/physDynamics.owl#Motion",
  "label": "Motion Detected",
  "description": "True when motion was detected in the room",
  "value": true
}
```

Listing

The next snippet is an example of boolean component used as data descriptor, hence with no value:

```
{
  "type": "Boolean",
  "definition": "http://mmisw.org/ont/q2o/test/timeContinuityTest",
  "label": "Time Continuity Test",
  "description": "Set to true to enable time continuity test"
}
```

Listing

9.1.6. Text Object

The “Text” object is the JSON schema implementation of the “Text” UML class defined in Clause 8.2.5. The schema for this class is provided in [Text.json](#).

Constraints on a “Text” representation are expressed using the AllowedTokens Object.

The following snippets show how the “Text” component can be used to define human readable text fields, as well as any other alpha-numerical properties:

```
{
  "type": "Text",
  "definition": "http://sensorml.com/ont/swe/property/Manufacturer",
  "label": "Manufacturer",
  "value": "Ocean Devices, Inc."
}
```

Listing

```
{
  "type": "Text",
  "definition": "http://sensorml.com/ont/x-swe/property/VehicleRegistrationNumber",
  "label": "License Plate",
}
```

```

    "value": "45ER-EJK-235"
  }

```

Listing

Constraints can also be used — typically when the component is used as a descriptor — to limit the possible text values, either by enumeration or a regular expression pattern:

```

{
  "type": "Text",
  "definition": "http://sensorml.com/ont/x-swe/property/VehicleRegistrationNumber",
  "label": "License Plate",
  "constraint": {
    "pattern": "^[0-9][A-Z]{4}-[A-Z]{3}-[0-9]{3}$"
  }
}

```

Listing

NOTE: This standard does not define any limit on the size of the text data than can be included as the value of a “Text” component, either inline or as part of a datastream. Implementations are responsible for documenting this upper limit.

9.1.7. Category Object

The “Category” object is the JSON schema implementation of the “Category” UML class defined in Clause 8.2.6. The schema for this class is provided in [Category.json](#).

Constraints on a “Category” representation are expressed using the AllowedTokens Object.

The following examples illustrate how the “Category” component is used to define various fields with categorical representations. The categorical scale is defined either via a code space, an enumeration constraint, or both (in which case the enumeration constraint defines a subset of possible values from a code space):

```

{
  "type": "Category",
  "definition": "http://sweet.jpl.nasa.gov/2.0/timeGeologic.owl#GeologicTime",
  "label": "Geological Period",
  "description": "Name of the geological period according to the nomenclature of the International Commission on Stratigraphy",
  "codeSpace": "http://sweet.jpl.nasa.gov/2.0/timeGeologic.owl#Period",
  "value": "Jurassic"
}

```

Listing

```

{
  "type": "Category",
  "definition": "http://sweet.jpl.nasa.gov/2.0/biol.owl#Species",
  "label": "Bird Species",
  "description": "Bird species according to the classification of the World Bird Database",
  "codeSpace": "http://www.birdlife.org/datazone/species/index.html"
}

```

```
}
```

Listing

9.1.8. Count Object

The “Count” object is the JSON schema implementation of the “Count” UML class defined in Clause 8.2.7. The schema for this class is provided in [Count.json](#).

Constraints on a “Count” representation are expressed using the AllowedValues Object.

The following snippet shows a “Count” component used to define the size of a row in a raster dataset:

```
{
  "type": "Count",
  "definition": "http://www.opengis.net/def/property/OGC/0/NumberOfPixels",
  "label": "Row Size",
  "description": "Number of pixels in each row of the image",
  "value": 1024
}
```

Listing

9.1.9. Quantity Object

The “Quantity” object is the JSON schema implementation of the “Quantity” UML class defined in Clause 8.2.8. The schema for this class is provided in [Quantity.json](#).

Constraints on a “Quantity” representation are expressed using the AllowedValues Object.

The unit of measure is defined using either a URI or a code expressed using the Unified Code for Units of Measure (UCUM) standard.

RECOMMENDATION 1

IDENTIFIER /rec/json-simple-components/ucum-code-used

STATEMENT Whenever it can be constructed using the UCUM specification, the unit of measure should be specified using a UCUM code as the value of the “uom/code” property. Otherwise the “uom/href” property should be used to reference an external unit definition.

REQUIREMENT 59

IDENTIFIER /req/json-simple-components/ucum-code-and-href-represent-same-unit

INCLUDED IN Requirements class 9: /req/json-simple-components

REQUIREMENT 59

STATEMENT If the unit of measurements is specified using a UCUM code as the value of the "uom/code" property and an external unit definition is referenced in the "uom/href" property, both the UCUM code and the external unit definition shall reference the same unit.

The following snippets show how "Quantity" components are used to define various (observable or controllable) properties with continuous decimal representations:

```
{
  "type": "Quantity",
  "definition": "http://qudt.org/vocab/quantitykind/Temperature",
  "label": "Outside Temperature",
  "description": "Outside temperature taken at the top of the antenna",
  "uom": { "code": "Cel" },
  "value": 21.5
}
```

Listing

```
{
  "type": "Quantity",
  "definition": "http://sensorml.com/ont/swe/property/SpectralRadiance",
  "label": "Radiance",
  "description": "Radiance measured on band1",
  "uom": { "code": "W.m-2.Sr-1.um-1" },
  "value": 2.83e-2
}
```

Listing

```
{
  "type": "Quantity",
  "definition": "http://sensorml.com/ont/swe/property/HeightAboveMSL",
  "referenceFrame": "http://www.opengis.net/def/crs/EPSG/0/5714",
  "axisID": "H",
  "label": "MSL Height",
  "description": "Height above mean sea level",
  "uom": { "code": "m" }
}
```

Listing

```
{
  "type": "Quantity",
  "definition": "https://qudt.org/vocab/quantitykind/CostPerUnitEnergy",
  "label": "Electricity Cost",
  "description": "Average cost of electricity in Europe",
  "uom": {
    "href": "https://qudt.org/vocab/unit/EUR-PER-KiloW-HR"
  }
}
```

Listing

9.1.10. Time Object

The “Time” object is the JSON schema implementation of the “Time” UML class defined in Clause 8.2.9. The schema for this class is provided in [Time.json](#).

Constraints on a “Time” representation are expressed using the AllowedTimes Object.

The unit of measure is defined using either a URI or a code expressed using the Unified Code for Units of Measure (UCUM) standard. When the temporal property is provided in the ISO 8601:2019 format, this is indicated by using a specific URI.

REQUIREMENT 60

IDENTIFIER /req/json-simple-components/iso8601-uom-used

INCLUDED IN Requirements class 9: /req/json-simple-components

STATEMENT When ISO 8601 notation is used to express the value associated to a “Time” element, the URI “<http://www.opengis.net/def/uom/ISO-8601/0/Gregorian>” shall be used as the value of the “uom/href” property.

The following snippets show how “Time” components are used to define various temporal properties, with different time scales:

ISO8601 formatted time stamp based on the UTC time standard:

```
{
  "type": "Time",
  "definition": "http://www.opengis.net/def/property/OGC-EO/0/MissionStartTime",
  "referenceFrame": "http://www.opengis.net/def/trs/BIPM/0/UTC",
  "localFrame": "urn:org:systems:001#MISSION-START-TIME",
  "label": "Flight Time",
  "description": "Time at take-off in UTC",
  "uom": {
    "href": "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian"
  },
  "value": "2009-01-26T10:21:45+01:00"
}
```

Listing

ISO8601 formatted time stamp based on the GPS time standard:

```
{
  "type": "Time",
  "definition": "http://www.opengis.net/def/property/OGC/0/SamplingTime",
  "referenceFrame": "http://www.opengis.net/def/trs/USNO/0/GPS",
  "label": "Sampling Time",
  "description": "Time at which the measurement was made",
  "uom": {
    "href": "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian"
  },
}
```

```

    "value": "2009-11-05T16:29:26Z"
  }

```

Listing

Time stamp in seconds past the Unix epoch:

```

{
  "type": "Time",
  "definition": "http://www.opengis.net/def/property/OGC/0/RunTime",
  "referenceTime": "1970-01-01T00:00:00Z",
  "label": "Model Run Time",
  "description": "Run time of the model expressed as a Unix time",
  "uom": { "code": "s" },
  "value": 1257415633
}

```

Listing

Time stamp in seconds past a custom time reference called MISSION_START_TIME:

```

{
  "type": "Time",
  "definition": "http://www.opengis.net/def/property/OGC-E0/0/ScanStartTime",
  "referenceFrame": "urn:org:systems:001#MISSION-START-TIME",
  "localFrame": "urn:org:systems:001#SCAN-START-TIME",
  "label": "Scanline Time",
  "description": "Acquisition time of the scan line",
  "uom": { "code": "s" }
}

```

Listing

9.1.11. CategoryRange Object

The “CategoryRange” object is the JSON schema implementation of the “CategoryRange” UML class defined in Clause 8.2.11. The schema for this class is provided in [CategoryRange.json](#).

“CategoryRange” objects share most properties with “Category” object, as shown on the following snippet:

```

{
  "type": "CategoryRange",
  "definition": "http://sweet.jpl.nasa.gov/2.0/timeGeologic.owl#GeologicTime",
  "label": "Approximate Dating",
  "description": "Approximate geological dating expressed as a range of geological eras",
  "codeSpace": "http://sweet.jpl.nasa.gov/2.0/timeGeologic.owl#Era",
  "value": ["Paleozoic", "Mesozoic"]
}

```

Listing

9.1.12. CountRange Object

The “CountRange” object is the JSON schema implementation of the “CountRange” UML class defined in Clause 8.2.12. The schema for this class is provided in [CountRange.json](#).

“CountRange” objects share most properties with “Count” object, as shown on the following snippet:

```
{
  "type": "CountRange",
  "definition": "http://www.opengis.net/def/property/OGC/0/ArrayIndex",
  "label": "Index Range",
  "value": [0, 3000]
}
```

Listing

9.1.13. QuantityRange Object

The “QuantityRange” object is the JSON schema implementation of the “QuantityRange” UML class defined in Clause 8.2.13. The schema for this class is provided in [QuantityRange.json](#).

“QuantityRange” objects share most properties with the “Quantity” object, as shown on the following snippet:

```
{
  "type": "QuantityRange",
  "definition": "http://mmisw.org/ont/mmi/device/OperationalRange",
  "label": "Operational Range",
  "description": "Operational temperature range of the cryogenic thermometer",
  "uom": { "code": "K" },
  "value": [10, 300]
}
```

Listing

9.1.14. TimeRange Object

The “TimeRange” object is the JSON schema implementation of the “TimeRange” UML class defined in Clause 8.2.14. The schema for this class is provided in [TimeRange.json](#).

“TimeRange” objects share most properties with the “Time” object, as shown on the following snippet:

```
{
  "type": "TimeRange",
  "definition": "http://www.opengis.net/def/property/E0/0/SurveyPeriod",
  "referenceFrame": "http://www.opengis.net/def/trs/BIPM/0/UTC",
  "label": "Survey Period",
  "uom": {
    "href": "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian"
  },
  "value": ["2008-01-05T11:02:54Z", "2009-11-05T16:29:26Z"]
}
```

```
}
```

Listing

9.1.15. NilValues Object

The “NilValues” object is the JSON schema implementation of the “NilValues” UML class defined in Clause 8.2.16. Schema patterns for this class are provided in [basicTypes.json](#). Several sub-patterns are defined for the decimal, integer and text cases.

Examples of NIL values definitions are provided below, in the case of numerical, countable and textual representations:

```
{
  "type": "Quantity",
  "definition": "http://sweet.jpl.nasa.gov/2.0/physRadiation.
owl#IonizingRadiation",
  "label": "Radiation Dose",
  "description": "Radiation dose measured by Gamma detector",
  "uom": { "code": "uR" },
  "nilValues": [
    { "reason": "http://www.opengis.net/def/nil/OGC/0/BelowDetectionRange",
      "value": "-Infinity" },
    { "reason": "http://www.opengis.net/def/nil/OGC/0/AboveDetectionRange",
      "value": "Infinity" }
  ]
}
```

Listing

```
{
  "type": "Count",
  "definition": "http://sweet.jpl.nasa.gov/2.0/physRadiation.owl#Radiance",
  "label": "Band 1",
  "nilValues": [
    { "reason": "http://www.opengis.net/def/nil/OGC/0/BelowDetectionRange",
      "value": 0 },
    { "reason": "http://www.opengis.net/def/nil/OGC/0/AboveDetectionRange",
      "value": 255 }
  ]
}
```

Listing

```
{
  "type": "Text",
  "definition": "http://sensorml.com/ont/x-swe/property/
VehicleRegistrationNumber",
  "label": "License Plate",
  "nilValues": [
    { "reason": "http://www.opengis.net/def/nil/OGC/0/Missing", "value":
      "Missing" },
    { "reason": "http://www.opengis.net/def/nil/OGC/0/Unknown", "value":
      "Unknown" }
  ]
}
```

Listing

9.1.16. AllowedTokens Object

The “AllowedTokens” object is the JSON schema implementation of the “AllowedTokens” UML class defined in Clause 8.2.17. The schema for this class is provided in [basicTypes.json](#) (see [#definitions/AllowedTokens](#)).

Examples of constraints for textual or categorical properties are provided below:

```
{
  "type": "Text",
  "definition": "http://sensorml.com/ont/swe/property/ModelNumber",
  "label": "Model Number",
  "constraint": {
    "pattern": "^[0-9][A-Z]{3}[0-9]{2}S1$"
  }
}
```

Listing

```
{
  "type": "Category",
  "definition": "http://www.opengis.net/def/property/OGC/0/SensorStatus",
  "label": "Sensor Status",
  "description": "Current connection status of the sensor",
  "constraint": {
    "values": [ "Off", "Stand-by", "Ready", "Busy" ]
  }
}
```

Listing

9.1.17. AllowedValues Object

The “AllowedValues” object is the JSON schema implementation of the “AllowedValues” UML class defined in Clause 8.2.18. The schema for this class is provided in [basicTypes.json](#) (see [#definitions/AllowedValues](#)).

Examples of constraints for various numerical properties are provided below:

```
{
  "type": "Quantity",
  "definition": "http://qudt.org/vocab/quantitykind/Angle",
  "label": "Planar Angle",
  "uom": { "code": "deg" },
  "constraint": {
    "intervals": [[-180, 180]]
  }
}
```

Listing

```
{
  "type": "Count",
  "definition": "http://www.opengis.net/def/property/OGC/0/NumberOfPixels",
  "label": "Image Width",
  "constraint": {
```

```

    "values": [256, 512, 1024]
  }
}

```

Listing

```

{
  "type": "Quantity",
  "definition": "http://sensorml.com/ont/swe/property/GeodeticLatitude",
  "label": "Latitude",
  "uom": { "code": "deg" },
  "constraint": {
    "intervals": [[-90, 90]],
    "significantFigures": 6
  }
}

```

Listing

Numerical constraints can also be unbounded:

```

{
  "type": "Quantity",
  "definition": "http://qudt.org/vocab/quantitykind/RadialDistance",
  "label": "Radial Distance",
  "description": "Radial distance is always positive",
  "uom": { "code": "m" },
  "constraint": {
    "intervals": [[0, "+Infinity"]]
  }
}

```

Listing

9.1.18. AllowedTimes Object

The “AllowedTimes” object is the JSON schema implementation of the “AllowedTimes” UML class defined in Clause 8.2.19. The schema for this class is provided in [basicTypes.json](#) (see [#definitions/AllowedTimes](#)).

Examples of constraints for various temporal properties, expressed as ISO-8601 or decimal values, are provided below:

```

{
  "type": "Time",
  "definition": "http://www.opengis.net/def/property/OGC/0/SamplingTime",
  "referenceFrame": "http://www.opengis.net/def/trs/USNO/0/GPS",
  "label": "Acquisition Time",
  "uom": {
    "href": "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian"
  },
  "constraint": {
    "intervals": [["2009-01-01T00:00:00Z", "+Infinity"]]
  }
}

```

Listing

```

{
  "type": "Time",

```

```

    "definition": "http://www.opengis.net/def/property/OGC/0/SamplingTime",
    "referenceFrame": "urn:org:systems:001#SCAN-START-TIME",
    "label": "Lidar Pulse Time",
    "description": "Time stamp of LiDAR pulse relative to start of scan",
    "uom": { "code": "ms" },
    "constraint": {
      "intervals": [[0, 1e6]]
    }
  }
}

```

Listing

9.2. Requirements Class: Record Components JSON Schema

REQUIREMENTS CLASS 10: RECORD COMPONENTS JSON SCHEMA	
IDENTIFIER	/req/json-record-components
TARGET TYPE	JSON Document
CONFORMANCE CLASS	Conformance class A.10: /conf/json-record-components
PREREQUISITE	Requirements class 9: /req/json-simple-components
INDIRECT PREREQUISITE	Requirements class 3: /req/uml-record-components
NORMATIVE STATEMENT	Requirement 61: /req/json-record-components/component-types

REQUIREMENT 61	
IDENTIFIER	/req/json-record-components/component-types
INCLUDED IN	Requirements class 10: /req/json-record-components
STATEMENT	The standardization target SHALL implement the following data component types: <i>DataRecord</i> , <i>Vector</i>

9.2.1. DataRecord Object

The “DataRecord” object is the JSON schema implementation of the “DataRecord” UML class defined in Clause 8.3.1. The schema for this class is provided in [DataRecord.json](#).

The example below describes a record composed of weather data fields. In this case the “DataRecord” element is used as a descriptor for records of data that are provided as part of a datastream:

```
{
  "type": "DataRecord",
  "label": "Weather Data Record",
  "fields": [
    {
      "name": "time",
      "type": "Time",
      "definition": "http://www.opengis.net/def/property/OGC/0/SamplingTime",
      "referenceFrame": "http://www.opengis.net/def/trs/BIPM/0/UTC",
      "label": "Sampling Time",
      "uom": {
        "href": "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian"
      }
    },
    {
      "name": "temperature",
      "type": "Quantity",
      "definition": "http://mmisw.org/ont/cf/parameter/air_temperature",
      "label": "Air Temperature",
      "uom": { "code": "Cel" }
    },
    {
      "name": "pressure",
      "type": "Quantity",
      "definition": "http://mmisw.org/ont/cf/parameter/air_pressure_at_mean_sea_level",
      "label": "Air Pressure",
      "uom": { "code": "mbar" }
    },
    {
      "name": "windSpeed",
      "type": "Quantity",
      "definition": "http://mmisw.org/ont/cf/parameter/wind_speed",
      "label": "Wind Speed",
      "uom": { "code": "km/h" }
    },
    {
      "name": "windDirection",
      "type": "Quantity",
      "definition": "http://mmisw.org/ont/cf/parameter/wind_to_direction",
      "label": "Wind Direction",
      "uom": { "code": "deg" }
    }
  ]
}
```

Listing

```
{
  "type": "DataRecord",
```

```

"definition": "urn:x-ogc:def:property:CSM::RadialDistortionCoefficients",
"label": "Radial Distortion Coefficients",
"fields": [
  {
    "name": "k1",
    "type": "Quantity",
    "definition": "urn:x-ogc:def:property:CSM::DISTOR_RAD1",
    "label": "Coef k1",
    "uom": { "code": "mm-2" },
    "value": 1.92709e-5
  },
  {
    "name": "k2",
    "type": "Quantity",
    "definition": "urn:x-ogc:def:property:CSM::DISTOR_RAD2",
    "label": "Coef k2",
    "uom": { "code": "mm-2" },
    "value": -5.14206e-10
  },
  {
    "name": "k3",
    "type": "Quantity",
    "definition": "urn:x-ogc:def:property:CSM::DISTOR_RAD3",
    "label": "Coef k3",
    "uom": { "code": "mm-2" },
    "value": -3.33356e-12
  }
]
}

```

Listing

9.2.2. Vector Object

The “Vector” object is the JSON schema implementation of the “Vector” UML class defined in Clause 8.3.2. The schema for this class is provided in [Vector.json](#).

```

{
  "type": "Vector",
  "definition": "http://www.opengis.net/def/property/OGC/0/PlatformLocation",
  "referenceFrame": "http://www.opengis.net/def/crs/EPSG/0/4326",
  "label": "Platform Location",
  "coordinates": [
    {
      "name": "lat",
      "type": "Quantity",
      "definition": "http://sensorml.com/ont/swe/property/GeodeticLatitude",
      "label": "Latitude",
      "axisID": "Lat",
      "uom": { "code": "deg" },
      "value": 45.36
    },
    {
      "name": "lon",
      "type": "Quantity",
      "definition": "http://sensorml.com/ont/swe/property/Longitude",
      "label": "Longitude",
      "axisID": "Lon",
      "uom": { "code": "deg" },
      "value": 5.2
    }
  ]
}

```

```

    }
  ]
}

```

Listing

```

{
  "type": "Vector",
  "definition": "http://qudt.org/vocab/quantitykind/LinearVelocity",
  "referenceFrame": "http://www.opengis.net/def/crs/OGC/0/ECI_J2000",
  "label": "Platform Velocity",
  "coordinates": [
    {
      "name": "vx",
      "type": "Quantity",
      "definition": "http://qudt.org/vocab/quantitykind/Speed",
      "label": "Velocity X",
      "uom": { "code": "m/s" }
    },
    {
      "name": "vy",
      "type": "Quantity",
      "definition": "http://qudt.org/vocab/quantitykind/Speed",
      "label": "Velocity Y",
      "uom": { "code": "m/s" }
    },
    {
      "name": "vz",
      "type": "Quantity",
      "definition": "http://qudt.org/vocab/quantitykind/Speed",
      "label": "Velocity Z",
      "uom": { "code": "m/s" }
    }
  ]
}

```

Listing

```

{
  "type": "Vector",
  "definition": "http://sensorml.com/ont/swe/property/RotationQuaternion",
  "referenceFrame": "http://www.opengis.net/def/crs/OGC/0/ECI_J2000",
  "localFrame": "urn:org:systems:001#PLATFORM_FRAME",
  "label": "Platform Orientation",
  "coordinates": [
    {
      "name": "qx",
      "type": "Quantity",
      "definition": "http://sensorml.com/ont/swe/property/Coordinate",
      "label": "QX",
      "uom": { "code": "1" }
    },
    {
      "name": "qy",
      "type": "Quantity",
      "definition": "http://sensorml.com/ont/swe/property/Coordinate",
      "label": "QY",
      "uom": { "code": "1" }
    },
    {
      "name": "qz",
      "type": "Quantity",
      "definition": "http://sensorml.com/ont/swe/property/Coordinate",

```

```

    "label": "QZ",
    "uom": { "code": "1" }
  },
  {
    "name": "qw",
    "type": "Quantity",
    "definition": "http://sensorml.com/ont/swe/property/Coordinate",
    "label": "QW",
    "uom": { "code": "1" }
  }
]
}

```

Listing

9.3. Requirements Class: Choice Components JSON Schema

REQUIREMENTS CLASS 11: CHOICE COMPONENTS JSON SCHEMA

IDENTIFIER	/req/json-choice-components
TARGET TYPE	JSON Document
CONFORMANCE CLASS	Conformance class A.11: /conf/json-choice-components
PREREQUISITE	Requirements class 9: /req/json-simple-components
INDIRECT PREREQUISITE	Requirements class 4: /req/uml-choice-components
NORMATIVE STATEMENT	Requirement 62: /req/json-choice-components/component-types

REQUIREMENT 62

IDENTIFIER	/req/json-choice-components/component-types
INCLUDED IN	Requirements class 11: /req/json-choice-components
STATEMENT	The standardization target SHALL implement the following data component types: <i>DataChoice</i>

9.3.1. DataChoice Object

The “DataChoice” object is the JSON schema implementation of the “DataChoice” UML class defined in Clause 8.4.1. The schema for this class is provided in [DataChoice.json](#).

```
{
  "type": "DataChoice",
  "label": "Weather Data Message",
  "items": [
    {
      "name": "TEMP",
      "type": "DataRecord",
      "label": "Temperature Measurement",
      "fields": [
        {
          "name": "time",
          "type": "Time",
          "definition": "http://www.opengis.net/def/property/OGC/0/SamplingTime",
          "referenceFrame": "http://www.opengis.net/def/trs/BIPM/0/UTC",
          "label": "Sampling Time",
          "uom": {
            "href": "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian"
          }
        },
        {
          "name": "temp",
          "type": "Quantity",
          "definition": "http://mmisw.org/ont/cf/parameter/air_temperature",
          "label": "Air Temperature",
          "uom": { "code": "Cel" }
        }
      ]
    },
    {
      "name": "PRESS",
      "type": "DataRecord",
      "label": "Pressure Measurement",
      "fields": [
        {
          "name": "time",
          "type": "Time",
          "definition": "http://www.opengis.net/def/property/OGC/0/SamplingTime",
          "referenceFrame": "http://www.opengis.net/def/trs/BIPM/0/UTC",
          "label": "Sampling Time",
          "uom": {
            "href": "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian"
          }
        },
        {
          "name": "press",
          "type": "Quantity",
          "definition": "http://mmisw.org/ont/cf/parameter/air_pressure_at_mean_sea_level",
          "label": "Air Pressure",
          "uom": { "code": "HPa" }
        }
      ]
    }
  ]
}
```



```
}
```

Listing

9.4. Requirements Class: Block Components JSON Schema

REQUIREMENTS CLASS 12: BLOCK COMPONENTS JSON SCHEMA	
IDENTIFIER	/req/json-block-components
TARGET TYPE	JSON Document
CONFORMANCE CLASS	Conformance class A.12: /conf/json-block-components
PREREQUISITE	Requirements class 9: /req/json-simple-components
INDIRECT PREREQUISITE	Requirements class 5: /req/uml-block-components
NORMATIVE STATEMENTS	Requirement 63: /req/json-block-components/component-types Requirement 64: /req/json-block-components/encoded-values-valid Requirement 65: /req/json-block-components/referenced-encoded-values-valid

REQUIREMENT 63	
IDENTIFIER	/req/json-block-components/component-types
INCLUDED IN	Requirements class 12: /req/json-block-components
STATEMENT	The standardization target SHALL implement the following data component types: <i>dataArray</i> , <i>Matrix</i> , <i>DataStream</i>

9.4.1. dataArray Object

The “dataArray” element is the JSON schema implementation of the “dataArray” UML class defined in Clause 8.5.1. The schema for this class is provided in [dataArray.json](#).

```
{
```

```

"type": "DataArray",
"label": "Calibration Table",
"elementType": {
  "name": "point",
  "type": "DataRecord",
  "label": "Data Point",
  "fields": [
    {
      "name": "t",
      "type": "Quantity",
      "definition": "https://qudt.org/vocab/quantitykind/Temperature",
      "label": "Temperature",
      "uom": { "code": "Cel" }
    },
    {
      "name": "r",
      "type": "Quantity",
      "definition": "https://qudt.org/vocab/quantitykind/Resistance",
      "label": "Resistance",
      "uom": { "code": "KOhm" }
    }
  ]
},
"values": [
  {"t": 12, "r": 3.03},
  {"t": 30.1, "r": 1.68},
  {"t": 40.0, "r": 1.16},
  {"t": 50.1, "r": 0.85},
  {"t": 59.8, "r": 0.62}
]
}

```

Listing

When provided inline, “DataArray” values are encoded using the method defined in Clause 9.4.4.

The following example shows how to define a 1D variable size array whose data is provided separately.

```

{
  "type": "DataArray",
  "definition": "http://sensorml.com/ont/swe/property/Trajectory",
  "label": "Mobile Trajectory",
  "elementCount": {
    "definition": "http://www.opengis.net/def/property/OGC/0/NumberOfPoints",
    "label": "Implicit Size"
  },
  "elementType": {
    "name": "point",
    "type": "Vector",
    "definition": "http://www.opengis.net/def/property/OGC/0/PlatformLocation",
    "referenceFrame": "http://www.opengis.net/def/crs/EPSG/0/4326",
    "label": "Location Point",
    "coordinates": [
      {
        "name": "lat",
        "type": "Quantity",
        "definition": "http://sensorml.com/ont/swe/property/GeodeticLatitude",
        "label": "Latitude",
        "axisID": "Lat",
        "uom": { "code": "deg" }
      }
    ]
  },
}

```

```

    {
      "name": "lon",
      "type": "Quantity",
      "definition": "http://sensorml.com/ont/swe/property/Longitude",
      "label": "Longitude",
      "axisID": "Lon",
      "uom": { "code": "deg" }
    }
  ]
}

```

Listing

“dataArray” components can also be nested to form multi-dimensional arrays, as shown in the following example of a 2D array:

```

{
  "type": "DataArray",
  "definition": "http://sensorml.com/ont/swe/property/RasterImage",
  "label": "Satellite Image",
  "elementCount": {
    "definition": "http://www.opengis.net/def/property/OGC/0/NumberOfRows",
    "value": 3000
  },
  "elementType": {
    "name": "row",
    "type": "DataArray",
    "definition": "http://sensorml.com/ont/swe/property/RasterImage",
    "elementCount": {
      "definition": "http://www.opengis.net/def/property/OGC/0/NumberOfSamples",
      "value": 3000
    },
    "elementType": {
      "name": "pixel",
      "type": "DataRecord",
      "definition": "http://sensorml.com/ont/swe/property/GridCell",
      "fields": [
        {
          "name": "band1",
          "type": "Quantity",
          "definition": "http://qudt.org/vocab/quantitykind/Radiance",
          "label": "Radiance",
          "description": "Radiance measured on band1",
          "uom": { "code": "W.m-2.Sr-1" }
        },
        {
          "name": "band2",
          "type": "Quantity",
          "definition": "http://qudt.org/vocab/quantitykind/Radiance",
          "label": "Radiance",
          "description": "Radiance measured on band2",
          "uom": { "code": "W.m-2.Sr-1" }
        },
        {
          "name": "band3",
          "type": "Quantity",
          "definition": "http://qudt.org/vocab/quantitykind/Radiance",
          "label": "Radiance",
          "description": "Radiance measured on band3",
          "uom": { "code": "W.m-2.Sr-1" }
        }
      ]
    }
  }
}

```

```

    }
  }
}

```

Listing

9.4.2. Matrix Object

The “Matrix” object is the JSON schema implementation of the “Matrix” UML class defined in Clause 8.5.2. The schema for this class is provided in [Matrix.json](#).

```

{
  "type": "Matrix",
  "definition": "http://sensorml.com/ont/swe/property/RotationMatrix",
  "referenceFrame": "http://www.opengis.net/def/crs/OGC/0/ECI_J2000",
  "label": "3D Orientation Matrix",
  "elementType": {
    "name": "row",
    "type": "Matrix",
    "elementType": {
      "name": "coef",
      "type": "Quantity",
      "definition": "http://sensorml.com/ont/swe/property/Coordinate",
      "label": "Matrix Coef",
      "uom": { "code": "1" }
    }
  },
  "values": [
    [0.36,0.48,-0.8],
    [-0.8,0.6,0],
    [0.48,0.64,0.6]
  ]
}

```

Listing

When provided inline, “Matrix” values are encoded using the method defined in Clause 9.4.4.

9.4.3. DataStream Object

The “DataStream” object is the JSON schema implementation of the “DataStream” UML class defined in Clause 8.5.3. The schema for this class is provided in [DataStream.json](#).

```

{
  "type": "DataStream",
  "label": "Aircraft Navigation",
  "elementType": {
    "name": "navData",
    "type": "DataRecord",
    "fields": [
      {
        "type": "Time",
        "definition": "http://www.opengis.net/def/property/OGC/0/SamplingTime",
        "referenceFrame": "http://www.opengis.net/def/trs/USNO/0/GPS",
        "referenceTime": "1970-01-01T00:00:00Z",
        "label": "Sampling Time",
        "uom": { "code": "s" }
      }
    ]
  }
}

```

```

{
  "name": "location",
  "type": "Vector",
  "definition": "http://www.opengis.net/def/property/OGC/0/PlatformLocation",
  "referenceFrame": "http://www.opengis.net/def/crs/EPSG/0/4979",
  "label": "Platform Location",
  "coordinates": [
    {
      "name": "lat",
      "type": "Quantity",
      "definition": "http://sensorml.com/ont/swe/property/GeodeticLatitude",
      "label": "Latitude",
      "axisID": "Lat",
      "uom": { "code": "deg" }
    },
    {
      "name": "lon",
      "type": "Quantity",
      "definition": "http://sensorml.com/ont/swe/property/Longitude",
      "label": "Longitude",
      "axisID": "Lon",
      "uom": { "code": "deg" }
    },
    {
      "name": "alt",
      "type": "Quantity",
      "definition": "http://sensorml.com/ont/swe/property/HeightAboveEllipsoid",
      "label": "Altitude",
      "axisID": "h",
      "uom": { "code": "m" }
    }
  ]
},
{
  "name": "attitude",
  "type": "Vector",
  "definition": "http://www.opengis.net/def/property/OGC/0/PlatformOrientation",
  "referenceFrame": "http://www.opengis.net/def/cs/OGC/0/ENU",
  "label": "Platform Attitude",
  "coordinates": [
    {
      "name": "heading",
      "type": "Quantity",
      "definition": "http://sensorml.com/ont/swe/property/TrueHeading",
      "label": "Heading",
      "axisID": "Z",
      "uom": { "code": "deg" }
    },
    {
      "name": "pitch",
      "type": "Quantity",
      "definition": "http://sensorml.com/ont/swe/property/PitchAngle",
      "label": "Pitch",
      "axisID": "X",
      "uom": { "code": "deg" }
    },
    {
      "name": "roll",
      "type": "Quantity",

```

```

        "definition": "http://sensorml.com/ont/swe/property/RollAngle",
        "label": "Roll",
        "axisID": "Y",
        "uom": { "code": "deg" }
      }
    ]
  },
  "encoding": {
    "type": "TextEncoding",
    "tokenSeparator": ",",
    "blockSeparator": "\n",
    "decimalSeparator": "."
  }
}

```

Listing

When provided inline, “DataStream” values are encoded using the method defined in Clause 9.4.4.

9.4.4. Inline Value Blocks

Inline values for “DataArray”, “Matrix” and “DataStream” components shall always be encoded using the JSON encoding rules when provided within a JSON document. No other method is allowed within a JSON document compliant with this standard. Inline block component values shall always be wrapped using a JSON Array.

REQUIREMENT 64

IDENTIFIER	/req/json-block-components/encoded-values-valid
INCLUDED IN	Requirements class 12: /req/json-block-components
STATEMENT	Inline values of all block components SHALL be encoded using the JSON Encoding Rules.

However, when values are provided separately from the component description (e.g., when datastream values are provided separately), any encoding methods defined in Clause 10 can be used. The out-of-band values can be referenced using an URL, for which at least the schemes http, https and data (as specified in IETF RFC 2397) shall be supported.

REQUIREMENT 65

IDENTIFIER	/req/json-block-components/referenced-encoded-values-valid
INCLUDED IN	Requirements class 12: /req/json-block-components

REQUIREMENT 65

STATEMENT Implementations SHALL support at least the schemes http, https and data for referenced values.

The data schema can be used to include all kind of data inline:

```
{
  "$schema": "file:///home/autermann/Source/ogcapi-connected-systems/swecommon/
schemas/json/DataArray.json",
  "type": "DataArray",
  "definition": "http://sensorml.com/ont/swe/property/RasterImage",
  "label": "Satellite Image",
  "elementCount": {
    "definition": "http://www.opengis.net/def/property/OGC/0/NumberOfRows",
    "value": 4
  },
  "elementType": {
    "name": "row",
    "type": "DataArray",
    "definition": "http://sensorml.com/ont/swe/property/RasterImage",
    "elementCount": {
      "definition": "http://www.opengis.net/def/property/OGC/0/NumberOfSamples",
      "value": 4
    },
    "elementType": {
      "name": "pixel",
      "type": "DataRecord",
      "definition": "http://sensorml.com/ont/swe/property/GridCell",
      "fields": [
        {
          "name": "band1",
          "type": "Quantity",
          "definition": "http://qudt.org/vocab/quantitykind/Radiance",
          "label": "Radiance",
          "description": "Radiance measured on band1",
          "uom": { "code": "W.m-2.Sr-1" }
        },
        {
          "name": "band2",
          "type": "Quantity",
          "definition": "http://qudt.org/vocab/quantitykind/Radiance",
          "label": "Radiance",
          "description": "Radiance measured on band2",
          "uom": { "code": "W.m-2.Sr-1" }
        },
        {
          "name": "band3",
          "type": "Quantity",
          "definition": "http://qudt.org/vocab/quantitykind/Radiance",
          "label": "Radiance",
          "description": "Radiance measured on band3",
          "uom": { "code": "W.m-2.Sr-1" }
        }
      ]
    }
  }
},
"encoding": {
  "type": "BinaryEncoding",
  "byteEncoding": "raw",
  "byteOrder": "bigEndian",
```

```

    "members": [
      {
        "type": "Component",
        "ref": "row/pixel/band1",
        "dataType": "http://www.opengis.net/def/dataType/OGC/0/unsignedByte"
      },
      {
        "type": "Component",
        "ref": "row/pixel/band2",
        "dataType": "http://www.opengis.net/def/dataType/OGC/0/unsignedByte"
      },
      {
        "type": "Component",
        "ref": "row/pixel/band3",
        "dataType": "http://www.opengis.net/def/dataType/OGC/0/unsignedByte"
      }
    ]
  },
  "values": {"href": "data:application/octet-stream;base64,MptSyfqPYAB5A9aV3j1uYw9EywICwMDZVcmnRlpS1NI1crn8K7NUe/X0I8r4IVq9"}
}

```

Listing

9.5. Requirements Class: Geometry Components JSON Schema

REQUIREMENTS CLASS 13: GEOMETRY COMPONENTS JSON SCHEMA	
IDENTIFIER	/req/json-geom-components
TARGET TYPE	JSON Document
CONFORMANCE CLASS	Conformance class A.13: /conf/json-geom-components
PREREQUISITE	Requirements class 9: /req/json-simple-components
INDIRECT PREREQUISITE	Requirements class 6: /req/uml-geom-components
NORMATIVE STATEMENT	Requirement 66: /req/json-geom-components/component-types

REQUIREMENT 66	
IDENTIFIER	/req/json-geom-components/component-types

REQUIREMENT 66

INCLUDED IN Requirements class 13: /req/json-geom-components

STATEMENT The standardization target SHALL implement the following data component types: *Geometry*

9.5.1. Geometry Object

The “Geometry” object is the JSON schema implementation of the “Geometry” UML class defined in Clause 8.6.1. The schema for this class is provided in [Geometry.json](#).

```
{
  "type": "Geometry",
  "definition": "http://sensorml.com/ont/swe/property/TargetLocation",
  "srs": "http://www.opengis.net/def/crs/EPSG/0/4326",
  "label": "Target Location",
  "description": "A point geometry",
  "value": {
    "type": "Point",
    "coordinates": [12.34, 56.36]
  }
}

{
  "type": "Geometry",
  "definition": "http://sensorml.com/ont/swe/property/Trajectory",
  "srs": "http://www.opengis.net/def/crs/EPSG/0/4326",
  "label": "Desired Trajectory",
  "description": "Desired UxS trajectory defined as a line string",
  "value": {
    "type": "LineString",
    "coordinates": [[12.34, 56.36], [12.45, 56.37], [12.45, 56.39], [12.34,
56.36]]
  }
}

{
  "type": "Geometry",
  "definition": "http://sensorml.com/ont/x-swe/property/SurveillanceArea",
  "srs": "http://www.opengis.net/def/crs/EPSG/0/4326",
  "label": "Surveillance Area",
  "description": "Desired UxS surveillance area defined as a polygon",
  "value": {
    "type": "Polygon",
    "coordinates": [
      [[12.34, 56.36], [12.45, 56.37], [12.45, 56.39], [12.34, 56.36]]
    ]
  }
}
```

Listing

9.6. Requirements Class: Simple Encodings JSON Schema

REQUIREMENTS CLASS 14: SIMPLE ENCODINGS JSON SCHEMA	
IDENTIFIER	/req/json-simple-encodings
TARGET TYPE	JSON Document
CONFORMANCE CLASS	Conformance class A.14: /conf/json-simple-encodings
PREREQUISITES	Requirements class 18: /req/text-encoding-rules Requirements class 17: /req/json-encoding-rules
INDIRECT PREREQUISITE	Requirements class 14: /req/json-simple-encodings
NORMATIVE STATEMENTS	Requirement 67: /req/json-simple-encodings/ encoding-types Requirement 68: /req/json-simple-encodings/json- encoding-rules-applied Requirement 69: /req/json-simple-encodings/text- encoding-rules-applied

REQUIREMENT 67	
IDENTIFIER	/req/json-simple-encodings/encoding-types
INCLUDED IN	Requirements class 14: /req/json-simple-encodings
STATEMENT	The standardization target SHALL support the following encoding types: <i>JSONEncoding</i> , <i>Text Encoding</i>

Validation patterns that implement classes defined in the “Simple Encodings” UML packages are provided in the JSON schema [encodings.json](#).

When datastream or data array values are provided out-of-band (i.e., not inline in the “DataArray”, “Matrix” or “DataStream” description), a different encoding than JSON can be selected. This is specified by using one of the following classes.

9.6.1. JSONEncoding Object

REQUIREMENT 68

IDENTIFIER /req/json-simple-encodings/json-encoding-rules-applied

INCLUDED IN Requirements class 14: /req/json-simple-encodings

STATEMENT The encoded values block described by a “JSONEncoding” object shall pass the “JSON Encoding Rules” conformance test class.

The “JSONEncoding” object is the JSON schema implementation of the “JSONEncoding” UML class defined in Clause 8.7.1. The schema for this class is provided in [encodings.json#/\\$defs/JSONEncoding](#).

```
{
  "type": "JSONEncoding",
  "recordsAsArrays": false,
  "vectorsAsArrays": false
}
```

Listing

The JSON encoding method is the default method when the data component tree is itself encoded in JSON.

9.6.2. TextEncoding Object

REQUIREMENT 69

IDENTIFIER /req/json-simple-encodings/text-encoding-rules-applied

INCLUDED IN Requirements class 14: /req/json-simple-encodings

STATEMENT The encoded values block described by a “TextEncoding” object shall pass the “Text Encoding Rules” conformance test class.

The “TextEncoding” object is the JSON schema implementation of the “TextEncoding” UML class defined in Clause 8.7.2. The schema for this class is provided in [encodings.json#/\\$defs/TextEncoding](#).

```
{
  "type": "TextEncoding",
  "tokenSeparator": ",",
  "blockSeparator": "\n",
  "decimalSeparator": "."
}
```

Listing

9.7. Requirements Class: Advanced Encodings JSON Schema

REQUIREMENTS CLASS 15: ADVANCED ENCODINGS JSON SCHEMA	
IDENTIFIER	/req/json-advanced-encodings
TARGET TYPE	JSON Document
CONFORMANCE CLASS	Conformance class A.15: /conf/json-advanced-encodings
PREREQUISITES	Requirements class 14: /req/json-simple-encodings Requirements class 19: /req/binary-encoding-rules
INDIRECT PREREQUISITE	Requirements class 8: /req/uml-advanced-encodings
NORMATIVE STATEMENTS	Requirement 70: /req/json-advanced-encodings/encoding-types Requirement 71: /req/json-advanced-encodings/binary-encoding-rules-applied Requirement 72: /req/json-advanced-encodings/ref-syntax-valid Requirement 73: /req/json-advanced-encodings/scalar-ref-component-valid Requirement 74: /req/json-advanced-encodings/datatype-valid Requirement 75: /req/json-advanced-encodings/datatype-compatible Requirement 76: /req/json-advanced-encodings/no-datatype-length Requirement 77: /req/json-advanced-encodings/block-ref-component-valid

REQUIREMENT 70	
IDENTIFIER	/req/json-advanced-encodings/encoding-types
INCLUDED IN	Requirements class 15: /req/json-advanced-encodings
STATEMENT	The standardization target SHALL support the following encoding types: <i>BinaryEncoding</i>

This requirement class defines an additional encoding method that can be used to encode data values as raw or base64 binary blocks.

9.7.1. BinaryEncoding Object

REQUIREMENT 71

IDENTIFIER /req/json-advanced-encodings/binary-encoding-rules-applied

INCLUDED IN Requirements class 15: /req/json-advanced-encodings

STATEMENT The encoded values block described by a “BinaryEncoding” element shall pass the “Binary Encoding Rules” conformance test class.

The “BinaryEncoding” object is the JSON schema implementation of the “BinaryEncoding” UML class defined in Clause 8.8.1. The schema for this class is provided in [encodings.json#/\\$defs/BinaryEncoding](#).

An example instance is provided below:

```
{
  "type": "BinaryEncoding",
  "byteOrder": "bigEndian",
  "byteEncoding": "raw",
  "members": [
    {
      "type": "Component",
      "ref": "/time",
      "dataType": "http://www.opengis.net/def/dataType/OGC/0/double"
    },
    {
      "type": "Block",
      "ref": "/img",
      "compression": "H264"
    }
  ]
}
```

Listing

9.7.1.1. Binary Component Object

The “Component” object implements the UML class with the same name. It is used to specify encoding parameters of scalar components. It allows for the detailed specification of the encoding parameters associated to components of the data description tree as discussed in Clause 8.8.1.

The “ref” attribute takes a value of a particular syntax that allows pointing to any data component. The syntax is a ‘/’ separated list of component names, starting with the name of the

root component and listed hierarchically. Each of these component names shall match the value of the “name” attribute defined in the data definition tree.

Requirement 72	
IDENTIFIER	/req/json-advanced-encodings/ref-syntax-valid
INCLUDED IN	Requirements class 15: /req/json-advanced-encodings
STATEMENT	The “ref” attribute of the “Component” or “Block” object SHALL contain a hierarchical ‘/’ separated list of data component names.

The “ref” attribute used on the “Component” element shall point exclusively to a scalar component.

Requirement 73	
IDENTIFIER	/req/json-advanced-encodings/scalar-ref-component-valid
INCLUDED IN	Requirements class 15: /req/json-advanced-encodings
STATEMENT	The “ref” attribute of a “Component” object SHALL reference a scalar or range component.

This standard defines the list of data types that are allowed for scalar values when encoded with the binary encoding method. The corresponding URIs listed below shall be used as the value of the datatype attribute of an instance of the “Component” element.

Requirement 74	
IDENTIFIER	/req/json-advanced-encodings/datatype-valid
INCLUDED IN	Requirements class 15: /req/json-advanced-encodings
STATEMENT	The value of the “dataType” property of the “Component” object SHALL be one of the URIs listed in Table 2.

These data types are specified in the normative table below:

Table 2 — Allowed Binary Data Types

Common Name	URI to use in “data Type” attribute	Description
-------------	-------------------------------------	-------------

Signed Byte	http://www.opengis.net/def/dataType/OGC/0/signedByte	8-bits signed binary integer. Range: -128 to +127
Unsigned Byte	http://www.opengis.net/def/dataType/OGC/0/unsignedByte	8-bits unsigned binary integer. Range: 0 to +255
Signed Short	http://www.opengis.net/def/dataType/OGC/0/signedShort	16-bits signed binary integer. Range: -32,768 to +32,767
Unsigned Short	http://www.opengis.net/def/dataType/OGC/0/unsignedShort	16-bits unsigned binary integer. Range: 0 to +65,535
Signed Int	http://www.opengis.net/def/dataType/OGC/0/signedInt	32-bits signed binary integer. Range: -2,147,483,648 to +2,147,483,647
Unsigned Int	http://www.opengis.net/def/dataType/OGC/0/unsignedInt	32-bits unsigned binary integer. Range: 0 to +4,294,967,295
Signed Long	http://www.opengis.net/def/dataType/OGC/0/signedLong	64-bits signed binary integer. Range: -2^{63} to $+2^{63} - 1$
Unsigned Long	http://www.opengis.net/def/dataType/OGC/0/unsignedLong	64-bits unsigned binary integer. Range: 0 to $+2^{64} - 1$
Half Precision Float	http://www.opengis.net/def/dataType/OGC/0/float16	16-bits single precision floating point number as defined in IEEE 754.
Float	http://www.opengis.net/def/dataType/OGC/0/float32	32-bits single precision floating point number as defined in IEEE 754.
Double	http://www.opengis.net/def/dataType/OGC/0/double or http://www.opengis.net/def/dataType/OGC/0/float64	64-bits double precision floating point number as defined in IEEE 754.
Long Double	http://www.opengis.net/def/dataType/OGC/0/float128	128-bits quadruple precision floating point number as defined in IEEE 754.

UTF-8 String (Variable Length)	http://www.opengis.net/def/dataType/OGC/0/string-utf-8 "byteLength" attribute is not set.	Variable length string composed of a 2-bytes unsigned short value indicating its length followed by a sequence of UTF-8 encoded characters as specified by the Unicode Standard (2.5).
UTF-8 String* (Fixed Length)	http://www.opengis.net/def/dataType/OGC/0/string-utf-8 "byteLength" attribute is set.	Fixed length string composed of a sequence of UTF-8 encoded characters as specified by the Unicode Standard (2.5), and padded with 0 characters.

The data type should be chosen so that its range allows the encoding of all possible values for a field (i.e., compatible with the field representation and constraints) including NIL values. This means that certain combinations of data type and components are not allowed. If a scalar component does not specify any constraint, any data type compatible with its representation can be used and it is the responsibility of the implementation to ensure that all future values for the component will "fit" in the data type.

REQUIREMENT 75

IDENTIFIER /req/json-advanced-encodings/datatype-compatible

INCLUDED IN Requirements class 15: /req/json-advanced-encodings

STATEMENT The chosen data type SHALL be compatible with the scalar component representation, constraints and NIL values.

Only data types marked with an asterisk allow the usage of the "byteLength" or "bitLength" attribute to customize their size. Usage of these attributes is forbidden on all other data types since their size is fixed and already specified in this standard (in the case of a variable length string, the size is included in the stream).

REQUIREMENT 76

IDENTIFIER /req/json-advanced-encodings/no-datatype-length

INCLUDED IN Requirements class 15: /req/json-advanced-encodings

STATEMENT The "bitLength" and "byteLength" properties SHALL not be set when a fixed size data type is used.

The value of the "byteEncoding" attribute allows the selection of either the 'raw' or 'base64' encoding methods. When 'base64' is selected each byte is converted to its base 64 representation before it is included in the encoded block, making it possible to include the values directly inline in the JSON instance.

9.7.1.2. Binary Block Object

The “Block” element implements the UML class with the same name. It is used to specify padding, encryption and/or compression of a block of data corresponding to an aggregate component.

The “ref” attribute shall point to an aggregate component in the data description and set one or more of the “compression,” “encryption,” or “padding” attributes.

REQUIREMENT 77

IDENTIFIER	/req/json-advanced-encodings/block-ref-component-valid
INCLUDED IN	Requirements class 15: /req/json-advanced-encodings
STATEMENT	The “ref” attribute of the “Block” object SHALL reference an aggregate component.

When padding is specified, padding bytes with a value of zero are inserted before (when “paddingBytesBefore” is set) and/or after (when “paddingBytesAfter” is set) the whole block of values corresponding to the aggregate components. Decoders should skip these bytes completely.

This standard does not specify specific compression or encryption methods. Future extensions can define single or groups of methods to target specific application domains. Compression methods can be specific such as the ones for video (e.g., MPEG-2, MPEG-4, etc.) or imagery (e.g., JPEG, JPEG2000, etc.) or generic so that they are applicable for any kind of data (e.g., GZIP, BZIP, etc.). They can be lossy or lossless. When a compression method results in variable length data blocks, the method should also define how the block length is specified.



10

DATA BLOCKS AND STREAMS ENCODING RULES (NORMATIVE)

DATA BLOCKS AND STREAMS ENCODING RULES (NORMATIVE)

10.1. Requirements Class: General Encoding Rules

REQUIREMENTS CLASS 16: GENERAL ENCODING RULES	
IDENTIFIER	/req/general-encoding-rules
TARGET TYPE	Encoded Values Instance
CONFORMANCE CLASS	Conformance class A.16: /conf/general-encoding-rules
INDIRECT PREREQUISITE	Requirements class 7: /req/uml-simple-encodings
NORMATIVE STATEMENTS	Requirement 78: /req/general-encoding-rules/record-encoding-rule Requirement 79: /req/general-encoding-rules/choice-encoding-rule Requirement 80: /req/general-encoding-rules/array-encoding-rule Requirement 81: /req/general-encoding-rules/array-size-encoding-rule

All encodings defined in this standard follow general principles so that it is possible to implement them in a similar way.

The way values are encoded is linked to the data structure specified using a hierarchy of data components. The values are included sequentially in the data stream by recursively processing all data components composing the dataset definition tree.

10.1.1. Rules for Scalar Components

The value of each scalar component is encoded as a single scalar value. The actual binary representation of this scalar value depends on the encoding method. For example, in “TextEncoding,” a numerical value is represented by its string representation that usually span several bytes (e.g., ‘1.2345’ spans 6 bytes), why with the “BinaryEncoding” encode a similar value would likely be encoded as an IEEE 754 single precision floating-point format.

The value of a “Time” component is encoded either as a decimal value or as a string in the case where a calendar representation or indeterminate value is used.

When the value of a scalar component is NIL, the appropriate nil value is used in the stream and replaces the actual measurement value. This is always possible because nil values are required to be expressed with a data type that is compatible with the representation of the corresponding field.

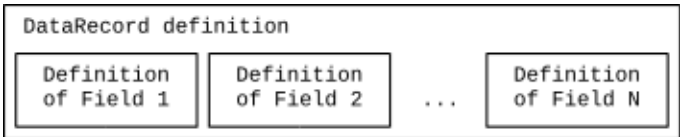
10.1.2. Rules for Range Components

The values of range components are encoded as a sequence of two successive values, first the lower bound of the range, then the upper bound. Each of these values is encoded exactly like the values of scalar components.

10.1.3. Rules for DataRecord and Vector

Both “DataRecord” and “Vector” components are aggregates consisting of an ordered sequence of child components. The values contained in these aggregates are encoded by successively encoding each child component in the order in which they are listed in the record or vector descriptor and including the resulting values sequentially in the stream.

The definition of a “DataRecord” (or “Vector”) structure composed of N fields (or coordinates for vectors) can be represented in the following way:

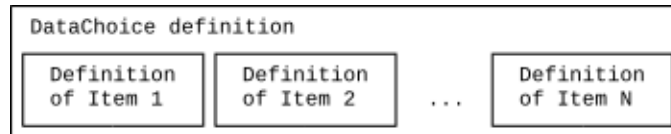


The data block corresponding to such a structure would sequentially include all values for field 1, then all values for field 2, etc. until the last field is reached. Each field may consist of a single value if it is a scalar but may also consist of multiple values if it is itself an aggregate or a range component.

REQUIREMENT 78	
IDENTIFIER	/req/general-encoding-rules/record-encoding-rule
INCLUDED IN	Requirements class 16: /req/general-encoding-rules
STATEMENT	“DataRecord” fields or “Vector” coordinates shall be encoded sequentially in a data block in the order in which these fields or coordinates are listed in the data descriptor.

10.1.4. Rules for DataChoice

The “DataChoice” is an aggregate consisting of a choice of several child components called items. When values of a data choice are encoded, the resulting data block consists of two things: A token identifying the selecting item and the item values themselves. Only values of a single item can be encoded in each instance of a choice.



The data block corresponding to such a structure would then sequentially include the item identifier (i.e., the choice value) and then the value(s) for the selected item. The item may consist of a single value if it is a scalar or multiple values if it is itself an aggregate or a range component.

REQUIREMENT 79

IDENTIFIER /req/general-encoding-rules/choice-encoding-rule

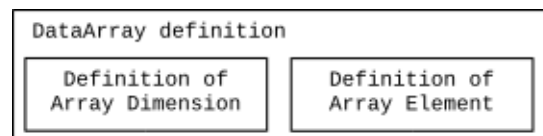
INCLUDED IN Requirements class 16: /req/general-encoding-rules

STATEMENT Encoded values for the selected item of a “DataChoice” shall be provided along with information that unambiguously identifies the selected item.

10.1.5. Rules for DataArray and Matrix

The “DataArray” is an aggregate consisting of a number of repeated elements, all of the same type as defined by the element type. Values contained by a “DataArray” are encoded by sequentially including the values of each element.

The definition of a “DataArray” (“Matrix”) structure composed of the array dimension and size and the element type definition. This can be represented in the following way:



The data block corresponding to such a structure would sequentially include the number representing the array size (only if it is variable) followed by one or more values corresponding to each array element. The number of values encoded for each element depends only on the array element definition, and the total number of values also depends on the array size.

REQUIREMENT 80

IDENTIFIER /req/general-encoding-rules/array-encoding-rule

INCLUDED IN Requirements class 16: /req/general-encoding-rules

STATEMENT “DataArray” elements shall be encoded sequentially in a data block in the order of their index in the array (i.e., from low to high index).

REQUIREMENT 81

IDENTIFIER /req/general-encoding-rules/array-size-encoding-rule

INCLUDED IN Requirements class 16: /req/general-encoding-rules

STATEMENT Encoded data for a variable size “DataArray” shall include a number specifying the array size whatever the encoding method used.

10.2. Requirements Class: JSON Encoding Rules

REQUIREMENTS CLASS 17: JSON ENCODING RULES

IDENTIFIER /req/json-encoding-rules

TARGET TYPE Encoded Values Instance

CONFORMANCE CLASS Conformance class A.17: /conf/json-encoding-rules

PREREQUISITE Requirements class 16: /req/general-encoding-rules

INDIRECT PREREQUISITE Requirements class 7: /req/uml-simple-encodings

NORMATIVE STATEMENTS

- Requirement 82: /req/json-encoding-rules/json-valid
- Requirement 83: /req/json-encoding-rules/scalar-value-valid
- Requirement 84: /req/json-encoding-rules/range-value-valid
- Requirement 85: /req/json-encoding-rules/record-object-valid
- Requirement 86: /req/json-encoding-rules/vector-object-valid

REQUIREMENTS CLASS 17: JSON ENCODING RULES

Requirement 87: /req/json-encoding-rules/choice-object-valid
Requirement 88: /req/json-encoding-rules/array-values-valid
Requirement 89: /req/json-encoding-rules/geometry-valid

The “JSON Encoding” method encodes field values by their JSON representation.

REQUIREMENT 82

IDENTIFIER /req/json-encoding-rules/json-valid

INCLUDED IN Requirements class 17: /req/json-encoding-rules

STATEMENT Data blocks and datastreams encoded using the JSON Encoding rules shall be valid JSON documents as defined by IETF RFC 8259.

The encoding rules defined in this document refer to JSON data types defined by IETF RFC 8259. Their definitions are recalled below:

JSON Object: An object structure is represented as a pair of curly brackets surrounding zero or more name/value pairs (or members). Members are separated by commas. Each member must have a distinct name.

JSON Array: An array structure is represented as square brackets surrounding zero or more values (or elements). Elements are separated by commas.

JSON Number: A decimal or integer number represented in base 10, with a sign and optional exponent.

JSON String: A string of Unicode characters that begins and ends with quotation marks.

10.2.1. Rules for Scalar Components

Scalar components are encoded as specified in Table 3. Special numerical values allowed for “Quantity” and “Time” components are defined in Clause 9.1.2.

REQUIREMENT 83

IDENTIFIER /req/json-encoding-rules/scalar-value-valid

INCLUDED IN Requirements class 17: /req/json-encoding-rules

REQUIREMENT 83

STATEMENT The value of a scalar component shall be represented using a JSON Number, a JSON String, or a boolean literal value, as defined in Table 3.

Table 3 — Simple Component to JSON Value Types Mapping

COMPONENT TYPE	JSON VALUE TYPE	EXAMPLES
Boolean	Boolean literal	true false
Text	JSON String	"word" "a full sentence" "BYC-589-AA"
Category	JSON String	"ON" "Paleozoic" "diesel"
Count	JSON Number	12 0
Quantity	JSON Number, or JSON String with special numerical value.	12 23.1 "NaN" "-Infinity" "+Infinity"
Time	JSON String with a ISO8601 date/time string, or JSON Number, or JSON String with special numerical value.	"2023-03-15T12:45:56Z" -23.1 12 "NaN" "-Infinity" "+Infinity"

10.2.2. Rules for Range Components

A range component is encoded using a JSON array of two values.

REQUIREMENT 84

IDENTIFIER /req/json-encoding-rules/range-value-valid

INCLUDED IN Requirements class 17: /req/json-encoding-rules

A Values of range components shall be wrapped in a JSON Array with exactly 2 scalar values.

REQUIREMENT 84

B	Each value is encoded in the same manner as the corresponding scalar component as defined in Table 3.
---	---

Table 4 — Range Component to JSON Mapping

COMPONENT TYPE	EXAMPLES
CategoryRange	["Cenozoic", "Paleozoic"]
CountRange	[0, 12]
QuantityRange	[-12, 35] [-180.0, 180.0] ["-Infinity", 0.0] [10.0, "+Infinity"] ["NaN", "NaN"]
TimeRange	["2023-01-01T00:00:00Z", "2023-03-15T12:45:56Z"] ["2023-01-01T00:00:00Z", "+Infinity"] ["-Infinity", "2023-01-01T00:00:00Z"] ["2023-01-01T00:00:00Z", "+Infinity"] ["NaN", "NaN"]

10.2.3. Rules for DataRecord and Vector

“DataRecord” and “Vector” components are encoded using a JSON Object whose members are named like the record fields per default. The attributes `vectorAsArrays` and `recordsAsArrays` of the corresponding JSON Encoding can be used to switch to a more compact encoding using JSON arrays.

REQUIREMENT 85

IDENTIFIER /req/json-encoding-rules/record-object-valid

INCLUDED IN Requirements class 17: /req/json-encoding-rules

A “DataRecord” values shall be wrapped either in a JSON Object or in a JSON Array. If the attribute ‘recordsAsArrays’ of the corresponding ‘JSONEncoding’ is true, all “DataRecord” values shall be encoded as JSON objects, else as JSON arrays. If the ‘recordsAsArrays’ or the corresponding ‘JSONEncoding’ is omitted, “DataRecord” values shall be encoded as JSON objects by default.

B If “DataRecord” values are encoded as JSON objects, the name of the JSON object members shall be the same as the “DataRecord” field names. The value of each JSON object member shall be chosen by following the encoding rules of the data component used as the record field with the same name.

REQUIREMENT 85

If a record field is marked as 'optional', the corresponding JSON object member can be omitted or its JSON value can be set to null.

C

If "DataRecord" values are encoded as JSON arrays, the order of JSON array items shall be the same as the "DataRecord" fields. The value of each JSON array item shall be chosen by following the encoding rules of the data component used as the record field at the same position. If a record field is marked as 'optional', the corresponding JSON array item can be set null, but cannot be omitted.

REQUIREMENT 86

IDENTIFIER /req/json-encoding-rules/vector-object-valid

**INCLUDED
IN**

Requirements class 17: /req/json-encoding-rules

A

"Vector" values shall be wrapped either in a JSON Object or in a JSON Array. If the attribute 'vectorsAsArrays' of the corresponding 'JSONEncoding' is true, all "Vector" values shall be encoded as JSON objects, else as JSON arrays. If the 'vectorsAsArrays' or the corresponding 'JSONEncoding' is omitted, "Vector" values shall be encoded as JSON objects by default.

B

If "Vector" values are encoded as JSON objects, the name of the JSON object members shall be the same as the "Vector" coordinate names. The value of each JSON object member shall be chosen by following the encoding rules of the data component used as the vector coordinate field with the same name.

C

If "Vector" values are encoded as JSON arrays, the order of JSON array items shall be the same as the "Vector" coordinate fields. The value of each JSON array item shall be chosen by following the encoding rules of the data component used as the vector coordinate field at the same position.

See the following examples:

- DataArray with inline values (curve)
- Datastream with records (weather data)
- Datastream with records and optional fields (navigation data)
- Datastream with records and vector fields encoded as arrays (navigation data)

10.2.4. Rules for DataChoice

Values of "DataChoice" components are encoded using a JSON Object with a single member whose name is the name of the selected choice item.

REQUIREMENT 87

IDENTIFIER /req/json-encoding-rules/choice-object-valid

INCLUDED IN Requirements class 17: /req/json-encoding-rules

- | | |
|----------|--|
| A | "DataChoice" values shall be encapsulated in a JSON Object. |
| B | The JSON object shall contain a single member whose name is the same as the selected choice item. |
| C | The JSON value of this unique member shall be chosen according to the encoding rules of the data component corresponding to the selected item. |

See example: Datastream with choice (navigation data)

10.2.5. Rules for DataArray and Matrix

Values of "DataArray" and "Matrix" components are encoded using a JSON Array, containing as many elements as there are elements in the Array component.

REQUIREMENT 88

IDENTIFIER /req/json-encoding-rules/array-values-valid

INCLUDED IN Requirements class 17: /req/json-encoding-rules

- | | |
|----------|---|
| A | "DataArray" and "Matrix" values shall be encapsulated in a JSON Array. |
| B | Each array element shall be encoded using the rules corresponding to the data component used as the array element type. |

See the following examples:

- Fixed size 2D array (stress matrix)
- Datastream of variable size 1D arrays (profile series)

10.2.6. Rules for Geometry

The value of a "Geometry" component is encoded using a GeoJSON Geometry object.

REQUIREMENT 89

IDENTIFIER /req/json-encoding-rules/geometry-valid

INCLUDED IN Requirements class 17: /req/json-encoding-rules

A The value of a “Geometry” component shall be encoded as a GeoJSON Geometry Object, following rules defined by IETF RFC 7946.

B The allowed GeoJSON geometry types shall be restricted to: Point, LineString, Polygon, MultiPoint, MultiLineString, and MultiPolygon

C The number of dimensions of the GeoJSON geometry shall match the number of dimensions of the coordinate reference system identified by the “srs” attribute of the component.

See example: Datastream with geometry (feature detection)

10.2.7. Media Types

When array or datastream values are encoded with the JSON encoding method and provided standalone (i.e., outside of any wrapper format), one of the following media type identifiers shall be used.

1. One of **application/json** or **application/vnd.ogc.swe+json** shall be used as the content-type for files and HTTP responses.
2. **application/vnd.ogc.swe+json** shall be used for format negotiation between server and client (e.g., when the format is advertised by an API or web service). In particular, this media type shall be used in HTTP Accept and Link headers and in any server response used to advertise support or link to a resource encoded with this format.

10.3. Requirements Class: Text Encoding Rules

REQUIREMENTS CLASS 18: TEXT ENCODING RULES

IDENTIFIER /req/text-encoding-rules

TARGET TYPE Encoded Values Instance

CONFORMANCE CLASS Conformance class A.18: /conf/text-encoding-rules

REQUIREMENTS CLASS 18: TEXT ENCODING RULES

PREREQUISITE	Requirements class 16: /req/general-encoding-rules
INDIRECT PREREQUISITE	Requirements class 7: /req/uml-simple-encodings
NORMATIVE STATEMENTS	Requirement 90: /req/text-encoding-rules/abnf-syntax-valid Requirement 91: /req/text-encoding-rules/separators-valid Requirement 92: /req/text-encoding-rules/optional-field-marker-present Requirement 93: /req/text-encoding-rules/choice-selection-marker-valid Requirement 94: /req/text-encoding-rules/geometry-valid

The “TextEncoding” method encodes field values (especially numbers) by their text representation. Special characters provide a way to separate successive values and successive blocks. The ABNF syntax defined in IETF RFC 5234 is used to formalize the encoding rules, and thus all ABNF snippets provided in this section are normative.

REQUIREMENT 90

IDENTIFIER	/req/text-encoding-rules/abnf-syntax-valid
INCLUDED IN	Requirements class 18: /req/text-encoding-rules
STATEMENT	The encoded values block shall be formatted as defined by the ABNF grammar defined in this clause.

10.3.1. Separators

Token separators are used between single values and the block separator is used at the end of each block. The block corresponds to one element of the “DataArray” or “DataStream” carrying the “values” element in which the values are encoded. There are no special separators to delimitate nested records, arrays and choices.

Separators shall be chosen so that nothing in the dataset contains the exact same character sequence as the one chosen for token or block separator.

REQUIREMENT 91

IDENTIFIER	/req/text-encoding-rules/separators-valid
------------	---

REQUIREMENT 91

INCLUDED IN

Requirements class 18: /req/text-encoding-rules

STATEMENT

Block and token separators used in the “TextEncoding” method shall be chosen as a sequence of characters that never occur in the data values themselves.

When the attribute “collapseWhiteSpaces” is set to true (its default value), all white space characters surrounding the token and block separators shall be ignored. The BNF grammar for separators is given below:

```
white-space = %d9 / %d10 / %d13 / %d32 ; TAB, LF, CR or SPACE
```

```
token-separator-chars = < Value of the tokenSeparator attribute >
```

```
block-separator-chars = < Value of the blockSeparator attribute >
```

```
token-separator = [white-space] token-separator-chars [white-space]
```

```
block-separator = [white-space] block-separator-chars [white-space]
```

Listing

White spaces around separators are in fact only allowed when the “collapseWhiteSpaces” attribute is set to ‘true’ (which is the default).

10.3.2. Rules for Scalar Components

The value for a scalar component is encoded as its text representation, following XML schema datatypes conventions.

```
scalar-value = xs:bool / xs:string / xs:double / xs:int / xs:date / xs:dateTime
```

Listing

Nil values are included in the stream just like normal scalar values. Since their data type has to match the field data type, there is no special treatment necessary for a decoder or encoder. It is the responsibility of the application to match the data value against the list of registered nil values for a given field in order to detect if it is associated to a nil reason or if it is an actual measurement value.

10.3.3. Rules for Range Components

Range components are encoded as a sequence of two tokens (each one representing a scalar value) separated by a token separator:

```
min-value = scalar-value
```

```
max-value = scalar-value
```

range-values = min-value token-separator max-value

Listing

10.3.4. Rules for DataRecord and Vector

Values of fields of a “DataRecord” are recursively encoded following rules associated to the type of component used for the field’s description (i.e., scalar, record, array, etc.) and separated by token separators as expressed by the following grammar:

field-count = < Number of fields in the record minus one. Greater or equal to 0 >

any-field-value = scalar-value / range-values / record-values / choice-values / array-values

mandatory-field-value = any-field-value

optional-field-value = (“Y” token-separator any-field-value) / “N”

field-value = mandatory-field-value / optional-field-value

record-values = field-value <field-count>*(token-separator field-value)

Listing

When a field is marked as optional in the definition, the token ‘Y’ or ‘N’ shall be inserted in the data block. When the field value is omitted, the token ‘N’ is inserted alone. When it is included, the token ‘Y’ is inserted followed by the actual field value.

REQUIREMENT 92

IDENTIFIER /req/text-encoding-rules/optional-field-marker-present

INCLUDED IN Requirements class 18: /req/text-encoding-rules

STATEMENT The ‘Y’ or ‘N’ token shall be inserted in a text encoded data block for all fields that have the “optional” attribute set to ‘true’.

Coordinate values of “Vector” components are encoded with a similar syntax, but a coordinate value can only be scalar and cannot be omitted:

coord-count = < Number of coordinates in the vector minus one. Greater or equal to 0 >

vector-values = scalar-value <coord-count>*(token-separator scalar-value)

Listing

See the following examples:

- DataArray with inline values (curve)

- Datastream with records (weather data)
- Datastream with records and optional fields (navigation data)

10.3.5. Rules for DataChoice

A “DataChoice” is encoded with the text method by providing the name of the selected item before the item values themselves. The name used shall correspond to the “name” attribute of the “item” property element that describes the structure of the selected item.

```
selected-item-name = < Value of the “name” attribute of the item selected >
selected-item-values = scalar-value / range-values / record-values / choice-
values / array-values
choice-values = selected-item-name token-separator selected-item-values
```

Listing

REQUIREMENT 93

IDENTIFIER /req/text-encoding-rules/choice-selection-marker-valid

INCLUDED IN Requirements class 18: /req/text-encoding-rules

STATEMENT The selected-item-name token shall correspond to the value of the “name” attribute of the “item” property element that represents the selected item.

See example: Datastream with choice (navigation data).

10.3.6. Rules for DataArray and Matrix

Values of each “DataArray” or “Matrix” element are recursively encoded following rules associated to the type of component used for the element type (i.e., scalar, record, array, etc.). Groups of values (or single value in the case of a scalar element type) corresponding to each element are sequentially appended to the data block and separated by token or block separators, depending on the context: When the “DataArray” is the root of the component tree that is being encoded, its elements are separated by block separators, otherwise its elements are separated by token separators.

A “DataArray” or “Matrix” can have a fixed or variable size, which leads to two slightly different syntaxes for encoding values: array-separator = token-separator / block-separator ; block-separator is only used when the array is the root of the component tree whose values are being encoded.

```
array-values = fixed-size-array-values / variable-size-array-values
```

Listing

Fixed size arrays have a size of at least one, and are encoded as defined below:

`fixed-element-count` = < Number of elements in a fixed size array minus one. Greater or equal to 0 since fixed size is always at least one >

`element-values` = `scalar-value` / `range-values` / `record-values` / `choice-values` / `array-values`

`fixed-size-array-values` = `element-values` <`fixed-element-count`>*(`array-separator` `element-values`)

Listing

When a “DataArray” (“Matrix”) is defined as variable size, its size can be 0 and the array size is included as a token in the data block, before the actual array elements values are listed:

`variable-element-count` = < Number of elements in a variable size array. Greater or equal to 0 since variable size can be 0 for an empty array >

`variable-size-array-values` = `variable-element-count` <`variable-element-count`>*(`array-separator` `element-values`)

Listing

See the following examples:

- DataArray with inline values (curve)
- Fixed size 2D array (stress matrix)
- Datastream of variable size 1D arrays (profile series)

10.3.7. Rules for DataStream

Values of “DataStream” elements are encoded as a sequence of tokens in a way similar to how “DataArray” values are encoded. Groups of encoded values corresponding to one element of a “DataStream” are always separated by block separators, while all values within these groups are separated by token separators:

`stream-element-count` = < Number of elements in a data stream minus one. Greater or equal to 0 since the number of elements in a data stream is always at least one >

`stream-values` = `element-values` <`stream-element-count`>*(`block-separator` `element-values`);

Listing

Examples of “DataStream” with “TextEncoding” have already been given in previous sections.

10.3.8. Rules for Geometry

The value of a “Geometry” component is encoded using the WKT format defined in the Simple Feature Access Standard (OGC 06-103r4).

REQUIREMENT 94

IDENTIFIER /req/text-encoding-rules/geometry-valid

INCLUDED IN Requirements class 18: /req/text-encoding-rules

A The value of a “Geometry” component shall be encoded using the WKT format defined in OGC 06-103r4, clause 7.

B The WKT representation shall be either a “Two-Dimension Geometry WKT” (clause 7.2.2 of OGC 06-103r4) or a “Three-Dimension Geometry WKT” (clause 7.2.3 of OGC 06-103r4). The ‘M’ coordinate shall not be used.

C The number of dimensions of the WKT geometry shall match the number of dimensions of the coordinate reference system identified by the “srs” attribute of the component.

D When a geometry value is included in a text-encoded datastream, the token separator shall not be the comma character (ASCII code 44) to avoid conflicting with commas used inside the WKT representation.

See example: Datastream with geometry (feature detection)

10.3.9. Media Types

When array or datastream values are encoded with the Text encoding method and provided standalone (i.e., outside of any wrapper format such as a JSON or XML document), one of the following media type identifiers shall be used.

1. One of **text/plain**, **text/csv**, or **application/vnd.ogc.swe+text** shall be used as the content-type for files and HTTP responses.
 - **text/csv** can be used only when the token separator is set to a single comma ‘,’ and the block separator is set to ‘CRLF’.
 - **text/plain** and **application/vnd.ogc.swe+text** can be used for any combination of separators.
2. **application/vnd.ogc.swe+text** shall be used for format negotiation between server and client (e.g., when the format is advertised by an API or web service). In particular, this media type shall be used in HTTP Accept and Link headers and in

any server response used to advertise support or link to a resource encoded with this format.

NOTE: It is recommended that the character set code be correctly appended to these media types if it differs from US-ASCII or UTF-8.

10.4. Requirements Class: Binary Encoding Rules

REQUIREMENTS CLASS 19: BINARY ENCODING RULES	
IDENTIFIER	/req/binary-encoding-rules
TARGET TYPE	Encoded Values Instance
CONFORMANCE CLASS	Conformance class A.19: /conf/binary-encoding-rules
PREREQUISITE	Requirements class 16: /req/general-encoding-rules
INDIRECT PREREQUISITE	Requirements class 8: /req/uml-advanced-encodings
NORMATIVE STATEMENTS	Requirement 19-6: /req/binary-encodings-rules/geometry-valid Requirement 95: /req/binary-encoding-rules/abnf-syntax-valid Requirement 96: /req/binary-encoding-rules/type-encoding-valid Requirement 97: /req/binary-encoding-rules/base64-translation-applied Requirement 98: /req/binary-encoding-rules/optional-field-marker-present Requirement 99: /req/binary-encoding-rules/choice-selection-marker-valid

The “BinaryEncoding” method encodes field values by their binary representation. The ABNF syntax defined in IETF RFC 5234 is used to formalize the encoding rules, and thus all ABNF snippets provided in this section are normative.

REQUIREMENT 95	
IDENTIFIER	/req/binary-encoding-rules/abnf-syntax-valid
INCLUDED IN	Requirements class 19: /req/binary-encoding-rules

REQUIREMENT 95

STATEMENT The encoded values block shall be formatted as defined by the ABNF grammar defined in this clause.

The encoding rules are similar to those of the “TextEncoding” method except that numerical values are encoded directly as their binary representation and that no separators are used. Separators are not needed because data types have either a fixed size or contain length information (See String encoding).

10.4.1. Rules for Scalar Components

The value for a scalar component is encoded as its binary representation. This especially applies to numerical values that are encoded directly in binary form in accordance to the selected data type and the value of the “byteOrder” attribute.

scalar-value = < binary value encoded according to data type, byte encoding and byte order specifications >

Listing

The last column of Table 2 indicates how each data type shall be binary encoded into a low level byte sequence. The actual order of bytes composing a multi-bytes data type depends on the value of the “byteOrder” attribute. The ‘bigEndian’ option indicates that multi-bytes data types are encoded with the most significant byte (MSB) first, while selecting ‘littleEndian’ signifies that encoding is done with the less significant byte (LSB) first. A UTF-8 string is not considered as a multi-byte data type and is always encoded in the same order, as specified by the Unicode Standard.

REQUIREMENT 96

IDENTIFIER /req/binary-encoding-rules/type-encoding-valid

INCLUDED IN Requirements class 19: /req/binary-encoding-rules

STATEMENT Binary data types in Table 2 shall be encoded according to their definition in the description column and the value of the “byteOrder” attribute.

Nil values are included in the stream just like normal scalar values. Since their data type has to match the field data type, there is no special treatment necessary for a decoder or encoder. It is the responsibility of the application to match the data value against the list of registered nil values for a given field in order to detect if it is associated to a nil reason or if it is an actual measurement value.

When the ‘raw’ byte encoding option is selected, bytes resulting from the data type encoding process defined above are inserted in the binary stream directly. This is referred to as ‘raw binary’ encoding. When the ‘base64’ option is selected, each byte resulting from the binary encoding process is also encoded in Base64 before being included in the stream. Scalar values

can be Base 64 encoded one by one or by blocks as long as the resulting stream is compatible with requirements of IETF RFC 2045.

REQUIREMENT 97

IDENTIFIER /req/binary-encoding-rules/base64-translation-applied

INCLUDED IN Requirements class 19: /req/binary-encoding-rules

STATEMENT When the 'base64' encoding option is selected, binary data shall be encoded with the Base64 technique defined in IETF RFC 2045 Section 6.8: Base64 Content-Transfer-Encoding.

10.4.2. Rules for Range Components

Range components are encoded as a sequence of two binary values (each one representing a scalar value):

min-value = scalar-value

max-value = scalar-value

range-values = min-value max-value

Listing

Values are always included in the same order: The lower bound of the range first, followed by the upper bound.

10.4.3. Rules for DataRecord and Vector

Values of fields of a "DataRecord" are recursively encoded following rules associated to the type of component used as the field's description (i.e., scalar, record, array, etc.) and appended to the binary block:

field-count = < Number of fields in the record. Greater or equal to 1 >

any-field-value = scalar-value / range-values / record-values / choice-values / array-values / block_values

mandatory-field-value = any-field-value

optional-field-value = ("Y" any-field-value) / "N"

field-value = mandatory-field-value / optional-field-value

record-values = <field-count>*field-values

Listing

When a field is marked as optional in the definition, the 1-byte value 'Y' (ASCII code 89) or 'N' (ASCII code 78) shall be inserted in the data block. When the field value is omitted, the token

'N' is inserted alone. When it is included, the token 'Y' is inserted followed by the actual field value.

REQUIREMENT 98

IDENTIFIER /req/binary-encoding-rules/optional-field-marker-present

INCLUDED IN Requirements class 19: /req/binary-encoding-rules

STATEMENT The one byte ASCII character 'Y' or 'N' shall be inserted in a binary encoded data block for all "Data Record" fields that have the "optional" attribute set to 'true'.

Coordinate values of "Vector" components are encoded with a similar syntax, but a coordinate value can only be scalar and cannot be omitted:

```
coord-count = < Number of coordinates in the vector. Greater or equal to 1 >
vector-values = <coord-count>*scalar-value
```

Listing

Vector coordinates cannot be optional.

10.4.4. Rules for DataChoice

A "DataChoice" is encoded with the binary method by providing the zero-based index of the selected item before the item values themselves. The index value ranges from 0 for the first choice item to (number_of_items - 1) for the last item.

```
selected-item-idx = < Index of the item selected >
selected-item-value = scalar-value / range-values / record-values / choice-
values / array-values
choice-values = selected-item-idx selected-item-value
```

Listing

REQUIREMENT 99

IDENTIFIER /req/binary-encoding-rules/choice-selection-marker-valid

INCLUDED IN Requirements class 19: /req/binary-encoding-rules

STATEMENT The value of the selected-item-idx flag shall be the zero-based index of the selected item (within the ordered list of items provided by the choice descriptor) and be encoded on a single unsigned byte.

10.4.5. Rules for DataArray and Matrix

Values of each “DataArray” or “Matrix” element are recursively encoded following rules associated to the type of component used for the element type (i.e., scalar, record, array, etc.). Groups of values (or single value in the case of a scalar element type) corresponding to each element are sequentially appended to the data block. Since a “DataArray” or “Matrix” can have a fixed or variable size, two slightly different syntaxes for encoding values are possible:

```
array-values = fixed-size-array-values / variable-size-array-values
```

```
element-value = scalar-value / range-values / record-values / choice-values /  
array-values / block_values
```

Listing

Fixed size arrays have a size of at least one, and are encoded as defined below:

```
fixed-element-count = < Number of elements in a fixed size array >
```

```
fixed-size-array-values = <fixed-element-count>*element-value
```

Listing

When a “DataArray” (“Matrix”) is defined as variable size, its size can be 0 and the array size is included as a token in the data block, before the actual array elements values are listed:

```
variable-element-count = < Number of elements in a variable size array >
```

```
variable-size-array-values = variable-element-count <variable-element-count>  
*element-value
```

Listing

When the array size is 0, only this number is encoded and no element values are included in the data block.

10.4.6. Rules for DataStream

Values of “DataStream” elements are encoded exactly as elements of an array:

```
stream-element-count = < Number of elements in a data stream >
```

```
stream-values = <stream-element-count>*element-value
```

Listing

A data stream usually contains at least one value but could be empty.

10.4.7. Rules for Geometry

The value of “Geometry” is encoded using the WKB format defined in the Simple Feature Access Standard (OGC 06-103r4).

REQUIREMENT 100

IDENTIFIER /req/binary-encoding-rules/geometry-valid

A	The value of a “Geometry” component shall be encoded using the WKB format defined in OGC 06-103r4, clause 8.
B	The WKB geometry type shall be one of the following types listed in OGC 06-103r4, clause 8.2.3, table 7: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, Point Z, LineString Z, Polygon Z, MultiPoint Z, MultiLineString Z, MultiPolygon Z. Other geometry type codes shall not be used.
C	The number of dimensions of the WKB geometry shall match the number of dimensions of the coordinate reference system identified by the “srs” attribute of the component.

No specific marker or length information needs to be pre-pended to the binary representation since the WKB format is self descriptive and parsable without knowing the total length ahead of time.

10.4.8. Block encoded components

When block encoding characteristics are also specified in the encoding description, the encryption and/or compression algorithm shall be applied after the binary encoding process described above is completed for the block. Extensions of this standard can define compression and encryption methods that fit the needs of particular communities.

In order to maximize compatibility with existing software, when compressing a binary encoded data stream results in a well known binary format, the corresponding mime type can be used instead of application/octet-stream. For instance video/h264 can be used when the entirety of the dataset (presumably a video stream) is compressed using the H264 video codec.

10.4.9. Media Types

When array or stream values are encoded with the Binary encoding method and provided standalone (i.e., outside of any wrapper format), one of the following media type identifiers shall be used:

1. One of **application/octet-stream** or **application/vnd.ogc.swe+binary** shall be used as the content-type for files and HTTP responses.

2. **application/vnd.ogc.swe+binary** shall be used for format negotiation between server and client (e.g., when the format is advertised by an API or web service). In particular, this media type shall be used in HTTP Accept and Link headers and in any server response used to advertise support or link to a resource encoded with this format.



ANNEX A (NORMATIVE) CONFORMANCE CLASS ABSTRACT TEST SUITE



ANNEX A

(NORMATIVE)

CONFORMANCE CLASS ABSTRACT TEST SUITE

A.1. Core Conformance Classes

A.1.1. Conformance Class: Core Concepts

CONFORMANCE CLASS A.1	
IDENTIFIER	/conf/core
REQUIREMENTS CLASS	Requirements class 1: /req/core
TARGET TYPE	Derived Models and Software Implementations
CONFORMANCE TESTS	Abstract test A.1: /conf/core/core-concepts-used Abstract test A.2: /conf/core/boolean-rep-valid Abstract test A.3: /conf/core/categorical-rep-valid Abstract test A.4: /conf/core/numerical-rep-valid Abstract test A.5: /conf/core/countable-rep-valid Abstract test A.6: /conf/core/textual-rep-valid Abstract test A.7: /conf/core/semantics-defined Abstract test A.8: /conf/core/semantics-resolvable Abstract test A.9: /conf/core/temporal-frame-defined Abstract test A.10: /conf/core/spatial-frame-defined Abstract test A.11: /conf/core/nil-reasons-defined Abstract test A.12: /conf/core/aggregates-model-valid Abstract test A.13: /conf/core/encoding-method-valid

ABSTRACT TEST A.1: CORE CONCEPTS ARE THE BASE OF ALL DERIVED MODELS

IDENTIFIER	/conf/core/core-concepts-used
REQUIREMENT	Requirement 1: /req/core/core-concepts-used
TEST PURPOSE	Verify that the target implementation correctly implements the core concepts.
TEST METHOD	Inspect the target implementation.

ABSTRACT TEST A.2: A BOOLEAN REPRESENTATION CONSISTS OF A BOOLEAN VALUE

IDENTIFIER	/conf/core/boolean-rep-valid
REQUIREMENT	Requirement 2: /req/core/boolean-rep-valid
TEST PURPOSE	Verify that the target implementation correctly implements the Boolean representation.
TEST METHOD	Inspect the target implementation.

ABSTRACT TEST A.3: A CATEGORICAL REPRESENTATION CONSISTS OF A TOKEN WITH A CODE SPACE

IDENTIFIER	/conf/core/categorical-rep-valid
REQUIREMENT	Requirement 3: /req/core/categorical-rep-valid
TEST PURPOSE	Verify that the target implementation correctly implements the Categorical representation.
TEST METHOD	Inspect the target implementation.

ABSTRACT TEST A.4: A CONTINUOUS NUMERICAL REPRESENTATION CONSISTS OF A NUMBER WITH A SCALE

IDENTIFIER	/conf/core/numerical-rep-valid
REQUIREMENT	Requirement 4: /req/core/numerical-rep-valid
TEST PURPOSE	Verify that the target implementation correctly implements the Numerical representation.

ABSTRACT TEST A.4: A CONTINUOUS NUMERICAL REPRESENTATION CONSISTS OF A NUMBER WITH A SCALE

TEST METHOD	Inspect the target implementation.
-------------	------------------------------------

ABSTRACT TEST A.5: A COUNTABLE REPRESENTATION CONSISTS OF AN INTEGER NUMBER

IDENTIFIER	/conf/core/countable-rep-valid
REQUIREMENT	Requirement 5: /req/core/countable-rep-valid
TEST PURPOSE	Verify that the target implementation correctly implements the Countable representation.
TEST METHOD	Inspect the target implementation.

ABSTRACT TEST A.6: A TEXTUAL REPRESENTATION IS IMPLEMENTED AS A CHARACTER STRING

IDENTIFIER	/conf/core/textual-rep-valid
REQUIREMENT	Requirement 6: /req/core/textual-rep-valid
TEST PURPOSE	Verify that the target implementation correctly implements the Textual representation.
TEST METHOD	Inspect the target implementation.

ABSTRACT TEST A.7: A SEMANTIC DEFINITION OF EACH PROPERTY SHALL BE PROVIDED

IDENTIFIER	/conf/core/semantics-defined
REQUIREMENT	Requirement 7: /req/core/semantics-defined
TEST PURPOSE	Verify that the target implementation allows attaching a semantic definition to all property representations.
TEST METHOD	Inspect the target implementation.

ABSTRACT TEST A.8: REFERENCES TO SEMANTICAL INFORMATION SHALL BE RESOLVABLE

IDENTIFIER	/conf/core/semantics-resolvable
REQUIREMENT	Requirement 8: /req/core/semantics-resolvable
TEST PURPOSE	Verify that the target implementation encodes the semantic links in a way that they can be resolved to an actual concept definition.
TEST METHOD	Inspect the target implementation.

ABSTRACT TEST A.9: A TEMPORAL QUANTITY IS ASSOCIATED TO A TEMPORAL REFERENCE FRAME

IDENTIFIER	/conf/core/temporal-frame-defined
REQUIREMENT	Requirement 9: /req/core/temporal-frame-defined
TEST PURPOSE	Verify that the target implementation allows providing a temporal reference frame along with any date/time quantity.
TEST METHOD	Inspect the target implementation.

ABSTRACT TEST A.10: A SPATIAL QUANTITY IS ASSOCIATED TO AN AXIS OF A SPATIAL REFERENCE FRAME

IDENTIFIER	/conf/core/spatial-frame-defined
REQUIREMENT	Requirement 10: /req/core/spatial-frame-defined
TEST PURPOSE	Verify that the target implementation allows providing a spatial reference frame and axis along with any quantity that is projected along a spatial dimension.
TEST METHOD	Inspect the target implementation.

ABSTRACT TEST A.11: A NIL VALUE MAPS A RESERVED VALUE TO A REASON

IDENTIFIER	/conf/core/nil-reasons-defined
REQUIREMENT	Requirement 11: /req/core/nil-reasons-defined
TEST PURPOSE	Verify that the target implementation allows providing a reason along with each NIL (reserved) value.

ABSTRACT TEST A.11: A NIL VALUE MAPS A RESERVED VALUE TO A REASON

TEST METHOD	Inspect the target implementation.
-------------	------------------------------------

ABSTRACT TEST A.12: AGGREGATE DATA TYPES ARE MODELED ACCORDING TO ISO 11404

IDENTIFIER	/conf/core/aggregates-model-valid
REQUIREMENT	Requirement 12: /req/core/aggregates-model-valid
TEST PURPOSE	Verify that the target implementation models aggregate data types according to ISO 11404 definitions.
TEST METHOD	Inspect the target implementation.

ABSTRACT TEST A.13: ENCODING METHODS SHALL BE DEFINED FOR ALL POSSIBLE DATA STRUCTURES

IDENTIFIER	/conf/core/encoding-method-valid
REQUIREMENT	Requirement 13: /req/core/encoding-method-valid
TEST PURPOSE	Verify that the target implementation provides encoding methods for all representations and all implemented data structures.
TEST METHOD	Inspect the target implementation.

A.2. UML Conformance Classes

A.2.1. Conformance Class: Basic Types and Simple Components UML Packages

CONFORMANCE CLASS A.2: BASIC TYPES AND SIMPLE COMPONENTS UML PACKAGES

IDENTIFIER	/conf/uml-simple-components
REQUIREMENTS CLASS	Requirements class 2: /req/uml-simple-components

CONFORMANCE CLASS A.2: BASIC TYPES AND SIMPLE COMPONENTS UML PACKAGES

PREREQUISITE	Conformance class A.1: /conf/core
TARGET TYPE	Derived Models and Software Implementations
CONFORMANCE TESTS	Abstract test A.23: /conf/uml-simple-components/ category-constraint-valid Abstract test A.24: /conf/uml-simple-components/ category-enum-defined Abstract test A.25: /conf/uml-simple-components/ category-value-valid Abstract test A.26: /conf/uml-simple-components/ time-ref-frame-defined Abstract test A.27: /conf/uml-simple-components/ time-ref-time-valid Abstract test A.28: /conf/uml-simple-components/ time-local-frame-valid Abstract test A.29: /conf/uml-simple-components/ range-value-valid Abstract test A.30: /conf/uml-simple-components/ category-range-valid Abstract test A.31: /conf/uml-simple-components/ category-range-codespace-order Abstract test A.32: /conf/uml-simple-components/ time-range-valid Abstract test A.14: /conf/uml-simple-components/ package-fully-implemented Abstract test A.33: /conf/uml-simple-components/ nil-reason-resolvable Abstract test A.34: /conf/uml-simple-components/ nil-value-type-coherent Abstract test A.35: /conf/uml-simple-components/ allowed-values-unit-coherent Abstract test A.15: /conf/uml-simple-components/ iso19103-implemented Abstract test A.16: /conf/uml-simple-components/ iso19108-implemented Abstract test A.17: /conf/uml-simple-components/ definition-present Abstract test A.18: /conf/uml-simple-components/ axis-valid Abstract test A.19: /conf/uml-simple-components/ axis-defined Abstract test A.20: /conf/uml-simple-components/ ref-frame-defined Abstract test A.21: /conf/uml-simple-components/ value-constraint-valid

CONFORMANCE CLASS A.2: BASIC TYPES AND SIMPLE COMPONENTS UML PACKAGES

Abstract test A.22: /conf/uml-simple-components/
value-attribute-present

ABSTRACT TEST A.14: COMPLIANCE WITH UML MODELS DEFINED IN THIS PACKAGE

IDENTIFIER	/conf/uml-simple-components/package-fully-implemented
REQUIREMENT	Requirement 14: /req/uml-simple-components/package-fully-implemented
TEST PURPOSE	Verify that the target implements all classes in the UML package.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.15: COMPLIANCE WITH UML MODELS DEFINED IN ISO 19103

IDENTIFIER	/conf/uml-simple-components/iso19103-implemented
REQUIREMENT	Requirement 15: /req/uml-simple-components/iso19103-implemented
TEST PURPOSE	Verify that the target implements all classes imported from ISO 19103 UML packages.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.16: COMPLIANCE WITH UML MODELS DEFINED IN ISO 19108

IDENTIFIER	/conf/uml-simple-components/iso19108-implemented
REQUIREMENT	Requirement 16: /req/uml-simple-components/iso19108-implemented
TEST PURPOSE	Verify that the target implements all classes imported from ISO 19108 UML packages.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.17: A DEFINITION URI IS MANDATORY ON ALL SIMPLE COMPONENTS

IDENTIFIER	/conf/uml-simple-components/definition-present
REQUIREMENT	Requirement 17: /req/uml-simple-components/definition-present
TEST PURPOSE	Verify that the target implementation has a constraint that enforces the requirement.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.18: THE VALUE OF THE AXISID AND AXISABBREV ATTRIBUTES MATCH

IDENTIFIER	/conf/uml-simple-components/axis-valid
REQUIREMENT	Requirement 18: /req/uml-simple-components/axis-valid
TEST PURPOSE	Verify that the target implementation has a constraint that enforces the requirement.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.19: THE AXIS ID IS ALWAYS SPECIFIED ON SCALAR SPATIAL PROPERTIES

IDENTIFIER	/conf/uml-simple-components/axis-defined
REQUIREMENT	Requirement 19: /req/uml-simple-components/axis-defined
TEST PURPOSE	Verify that the target implementation has a constraint that enforces the requirement.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.20: THE REFERENCE FRAME IS SPECIFIED ON SCALAR SPATIAL PROPERTIES NOT PART OF A VECTOR

IDENTIFIER	/conf/uml-simple-components/ref-frame-defined
REQUIREMENT	Requirement 20: /req/uml-simple-components/ref-frame-defined
TEST PURPOSE	Verify that the target implementation has a constraint that enforces the requirement.

ABSTRACT TEST A.20: THE REFERENCE FRAME IS SPECIFIED ON SCALAR SPATIAL PROPERTIES NOT PART OF A VECTOR

TEST METHOD	Inspect the model or software implementation.
-------------	---

ABSTRACT TEST A.21: THE VALUE OF A COMPONENT SATISFIES THE CONSTRAINTS

IDENTIFIER	/conf/uml-simple-components/value-constraint-valid
REQUIREMENT	Requirement 21: /req/uml-simple-components/value-constraint-valid
TEST PURPOSE	Verify that the target implementation has a constraint that enforces the requirement.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.22: ALL DERIVED SIMPLE COMPONENTS HAVE AN OPTIONAL VALUE ATTRIBUTE

IDENTIFIER	/conf/uml-simple-components/value-attribute-present
REQUIREMENT	Requirement 22: /req/uml-simple-components/value-attribute-present
TEST PURPOSE	Verify that the target implementation has a constraint that enforces the requirement.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.23: THE LIST OF VALUES ALLOWED IN A CATEGORY COMPONENT IS A SUBSET OF THE CODE SPACE

IDENTIFIER	/conf/uml-simple-components/category-constraint-valid
REQUIREMENT	Requirement 23: /req/uml-simple-components/category-constraint-valid
TEST PURPOSE	Verify that the target implementation has a constraint that enforces the requirement.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.24: A CATEGORY COMPONENT ALWAYS SPECIFIES A LIST OF POSSIBLE VALUES

IDENTIFIER	/conf/uml-simple-components/category-enum-defined
REQUIREMENT	Requirement 24: /req/uml-simple-components/category-enum-defined
TEST PURPOSE	Verify that the target implementation has a constraint that enforces the requirement.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.25: THE VALUE OF A CATEGORY COMPONENT IS ONE DEFINED IN THE CODE SPACE

IDENTIFIER	/conf/uml-simple-components/category-value-valid
REQUIREMENT	Requirement 25: /req/uml-simple-components/category-value-valid
TEST PURPOSE	Verify that the target implementation has a constraint that enforces the requirement.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.26: THE TEMPORAL REFERENCE FRAME IS DEFINED

IDENTIFIER	/conf/uml-simple-components/time-ref-frame-defined
REQUIREMENT	Requirement 26: /req/uml-simple-components/time-ref-frame-defined
TEST PURPOSE	Verify that the implementation correctly assumes the default value when the attribute is not set.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.27: THE TIME OF REFERENCE IS EXPRESSED RELATIVE TO THE ORIGIN OF THE REFERENCE FRAME

IDENTIFIER	/conf/uml-simple-components/time-ref-time-valid
REQUIREMENT	Requirement 27: /req/uml-simple-components/time-ref-time-valid
TEST PURPOSE	Verify that the target implementation has a constraint that enforces the requirement.

ABSTRACT TEST A.27: THE TIME OF REFERENCE IS EXPRESSED RELATIVE TO THE ORIGIN OF THE REFERENCE FRAME

TEST METHOD

Inspect the model or software implementation.

ABSTRACT TEST A.28: THE LOCAL AND REFERENCE FRAMES OF A TIME COMPONENT ARE DIFFERENT

IDENTIFIER

/conf/uml-simple-components/time-local-frame-valid

REQUIREMENT

Requirement 28: /req/uml-simple-components/time-local-frame-valid

TEST PURPOSE

Verify that the target implementation has a constraint that enforces the requirement.

TEST METHOD

Inspect the model or software implementation.

ABSTRACT TEST A.29: VALUES OF RANGE COMPONENTS SATISFY THE SAME REQUIREMENTS AS SCALAR VALUES

IDENTIFIER

/conf/uml-simple-components/range-value-valid

REQUIREMENT

Requirement 29: /req/uml-simple-components/range-value-valid

TEST PURPOSE

Verify that the target implementation has a constraint that enforces the requirement.

TEST METHOD

Inspect the model or software implementation.

ABSTRACT TEST A.30: CATEGORYRANGE COMPONENTS SATISFY ALL REQUIREMENTS OF A CATEGORY COMPONENT

IDENTIFIER

/conf/uml-simple-components/category-range-valid

REQUIREMENT

Requirement 30: /req/uml-simple-components/category-range-valid

TEST PURPOSE

Verify that the target implementation has constraints that enforce the requirement.

TEST METHOD

Inspect the model or software implementation. Apply the following conformance tests to the "CategoryRange" class:

- Abstract test A.23: /conf/uml-simple-components/category-constraint-valid
- Abstract test A.24: /conf/uml-simple-components/category-enum-defined

ABSTRACT TEST A.30: CATEGORYRANGE COMPONENTS SATISFY ALL REQUIREMENTS OF A CATEGORY COMPONENT

- Abstract test A.25: /conf/uml-simple-components/category-value-valid

ABSTRACT TEST A.31: THE CODE SPACE OF A CATEGORYRANGE COMPONENT IS WELL-ORDERED

IDENTIFIER	/conf/uml-simple-components/category-range-codespace-order
REQUIREMENT	Requirement 31: /req/uml-simple-components/category-range-codespace-order
TEST PURPOSE	Verify that the code space contains elements that have a specific order (either implied or defined).
TEST METHOD	Inspect instances generated by the implementation of the “CategoryRange” class, including a codespace, to verify the requirement.

ABSTRACT TEST A.32: TIMERANGE COMPONENTS SATISFY ALL REQUIREMENTS OF THE TIME CLASS

IDENTIFIER	/conf/uml-simple-components/time-range-valid
REQUIREMENT	Requirement 32: /req/uml-simple-components/time-range-valid
TEST PURPOSE	Verify that the target implementation has constraints that enforce the requirement.
TEST METHOD	<p>Inspect the model or software implementation. Apply the following conformance tests to the “TimeRange” class:</p> <ul style="list-style-type: none">• Abstract test A.26: /conf/uml-simple-components/time-ref-frame-defined• Abstract test A.27: /conf/uml-simple-components/time-ref-time-valid• Abstract test A.28: /conf/uml-simple-components/time-local-frame-valid

ABSTRACT TEST A.33: THE REASON ATTRIBUTE IS A URI THAT IS RESOLVABLE TO A DEFINITION

IDENTIFIER	/conf/uml-simple-components/nil-reason-resolvable
REQUIREMENT	Requirement 33: /req/uml-simple-components/nil-reason-resolvable
TEST PURPOSE	Verify that the target implementation allows the value of a NIL reason identifier to be either:

ABSTRACT TEST A.33: THE REASON ATTRIBUTE IS A URI THAT IS RESOLVABLE TO A DEFINITION

- a well known reason code defined by OGC
- a URI that can be resolved to the textual description of a custom reason.

TEST METHOD

Inspect the model or software implementation.

ABSTRACT TEST A.34: VALUES RESERVED FOR NIL REASONS ARE COMPATIBLE WITH THE COMPONENT DATA TYPE

IDENTIFIER	/conf/uml-simple-components/nil-value-type-coherent
REQUIREMENT	Requirement 34: /req/uml-simple-components/nil-value-type-coherent
TEST PURPOSE	Verify that the target implementation has a constraint that enforces the requirement.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.35: THE SCALE OF CONSTRAINTS IS THE SAME AS THE SCALE OF THE COMPONENT VALUE

IDENTIFIER	/conf/uml-simple-components/allowed-values-unit-coherent
REQUIREMENT	Requirement 35: /req/uml-simple-components/allowed-values-unit-coherent
TEST PURPOSE	Verify that numerical constraints are expressed with the correct scale.
TEST METHOD	Inspect instances generated by the implementation of the "Quantity", "Count" and "Time" classes, including an "AllowedValues" constraint, to verify the requirement.

A.2.2. Conformance Class: Record Components UML Package

CONFORMANCE CLASS A.3: RECORD COMPONENTS UML PACKAGE

IDENTIFIER	/conf/uml-record-components
REQUIREMENTS CLASS	Requirements class 3: /req/uml-record-components

CONFORMANCE CLASS A.3: RECORD COMPONENTS UML PACKAGE

PREREQUISITE	Conformance class A.2: /conf/uml-simple-components
TARGET TYPE	Derived Models and Software Implementations
CONFORMANCE TESTS	Abstract test A.36: /conf/uml-record-components/package-fully-implemented Abstract test A.37: /conf/uml-record-components/record-field-name-unique Abstract test A.38: /conf/uml-record-components/vector-coord-name-unique Abstract test A.39: /conf/uml-record-components/vector-component-no-ref-frame Abstract test A.40: /conf/uml-record-components/vector-component-axis-defined Abstract test A.41: /conf/uml-record-components/vector-local-frame-valid

ABSTRACT TEST A.36: COMPLIANCE WITH UML MODELS DEFINED IN THIS PACKAGE

IDENTIFIER	/conf/uml-record-components/package-fully-implemented
REQUIREMENT	Requirement 36: /req/uml-record-components/package-fully-implemented
TEST PURPOSE	Verify that the target implements all classes in the UML package.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.37: EACH DATARECORD FIELD HAS A UNIQUE NAME

IDENTIFIER	/conf/uml-record-components/record-field-name-unique
REQUIREMENT	Requirement 37: /req/uml-record-components/record-field-name-unique
TEST PURPOSE	Verify that the implementation of the “DataRecord” class has a constraint that enforces the requirement.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.38: EACH VECTOR COORDINATE HAS A UNIQUE NAME

IDENTIFIER	/conf/uml-record-components/vector-coord-name-unique
REQUIREMENT	Requirement 38: /req/uml-record-components/vector-coord-name-unique
TEST PURPOSE	Verify that the implementation of the “Vector” class has a constraint that enforces the requirement.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.39: THE REFERENCE FRAME IS NOT SPECIFIED ON INDIVIDUAL COORDINATES OF A VECTOR

IDENTIFIER	/conf/uml-record-components/vector-component-no-ref-frame
REQUIREMENT	Requirement 39: /req/uml-record-components/vector-component-no-ref-frame
TEST PURPOSE	Verify that the implementation of the “Vector” class has a constraint that enforces the requirement.
TEST METHOD	<ul style="list-style-type: none">Inspect the model or software implementation.The “referenceFrame” attribute shall be omitted from all data components used to define coordinates of a “Vector” instance.

ABSTRACT TEST A.40: THE AXIS ID IS SPECIFIED ON ALL COORDINATES OF A VECTOR

IDENTIFIER	/conf/uml-record-components/vector-component-axis-defined
REQUIREMENT	Requirement 40: /req/uml-record-components/vector-component-axis-defined
TEST PURPOSE	Verify that the implementation of the “Vector” class has a constraint that enforces the requirement.
TEST METHOD	<ul style="list-style-type: none">Inspect the model or software implementation.The “axisID” attribute shall be present on all data components used to define coordinates of a “Vector” instance.

ABSTRACT TEST A.41: THE LOCAL AND REFERENCE FRAMES OF A VECTOR COMPONENT ARE DIFFERENT

IDENTIFIER	/conf/uml-record-components/vector-local-frame-valid
REQUIREMENT	Requirement 41: /req/uml-record-components/vector-local-frame-valid

ABSTRACT TEST A.41: THE LOCAL AND REFERENCE FRAMES OF A VECTOR COMPONENT ARE DIFFERENT

TEST PURPOSE	Verify that the implementation of the “Vector” class has a constraint that enforces the requirement.
TEST METHOD	Inspect the model or software implementation.

A.2.3. Conformance Class: Choice Components UML Package

CONFORMANCE CLASS A.4: CHOICE COMPONENTS UML PACKAGE

IDENTIFIER	/conf/uml-choice-components
REQUIREMENTS CLASS	Requirements class 4: /req/uml-choice-components
PREREQUISITE	Conformance class A.2: /conf/uml-simple-components
TARGET TYPE	Derived Models and Software Implementations
CONFORMANCE TESTS	Abstract test A.42: /conf/uml-choice-components/package-fully-implemented Abstract test A.43: /conf/uml-choice-components/choice-item-name-unique

ABSTRACT TEST A.42: COMPLIANCE WITH UML MODELS DEFINED IN THIS PACKAGE

IDENTIFIER	/conf/uml-choice-components/package-fully-implemented
REQUIREMENT	Requirement 42: /req/uml-choice-components/package-fully-implemented
TEST PURPOSE	Verify that the target implements all classes in the UML package.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.43: EACH DATA CHOICE ITEM HAS A UNIQUE NAME

IDENTIFIER	/conf/uml-choice-components/choice-item-name-unique
REQUIREMENT	Requirement 43: /req/uml-choice-components/choice-item-name-unique

ABSTRACT TEST A.43: EACH DATACHOICE ITEM HAS A UNIQUE NAME

TEST PURPOSE	Verify that the implementation of the "DataChoice" class has a constraint that enforces the requirement.
TEST METHOD	Inspect the model or software implementation.

A.2.4. Conformance Class: Block Components UML Package

CONFORMANCE CLASS A.5: BLOCK COMPONENTS UML PACKAGE

IDENTIFIER	/conf/uml-block-components
REQUIREMENTS CLASS	Requirements class 5: /req/uml-block-components
PREREQUISITE	Conformance class A.2: /conf/uml-simple-components
TARGET TYPE	Derived Models and Software Implementations
CONFORMANCE TESTS	Abstract test A.44: /conf/uml-block-components/package-fully-implemented Abstract test A.45: /conf/uml-block-components/array-component-no-value Abstract test A.46: /conf/uml-block-components/array-values-properly-encoded Abstract test A.47: /conf/uml-block-components/matrix-element-type-valid

ABSTRACT TEST A.44: COMPLIANCE WITH UML MODELS DEFINED IN THIS PACKAGE

IDENTIFIER	/conf/uml-block-components/package-fully-implemented
REQUIREMENT	Requirement 44: /req/uml-block-components/package-fully-implemented
TEST PURPOSE	Verify that the target implements all classes in the UML package.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.45: COMPONENTS NESTED IN A BLOCK COMPONENT ARE DATA DESCRIPTORS

IDENTIFIER	/conf/uml-block-components/array-component-no-value
REQUIREMENT	Requirement 45: /req/uml-block-components/array-component-no-value
TEST PURPOSE	Verify that implementations of the block component classes have a constraint that enforces the requirement.
TEST METHOD	<ul style="list-style-type: none">Inspect the model or software implementation.Check that the “DataArray”, “Matrix” and “DataStream” classes have the constraint.

ABSTRACT TEST A.46: AN ENCODING METHOD IS SPECIFIED WHENEVER AN ENCODED DATA BLOCK IS INCLUDED

IDENTIFIER	/conf/uml-block-components/array-values-properly-encoded
REQUIREMENTS	Requirement 46: /req/uml-block-components/array-values-properly-encoded Requirement 48: /req/uml-block-components/datastream-array-valid
TEST PURPOSE	Verify that the implementation of block component classes have a constraint that enforces the requirement.
TEST METHOD	<ul style="list-style-type: none">Inspect the model or software implementation.Check that the “DataArray”, “Matrix” and “DataStream” classes have the constraint.Inspect instances of these classes generated by the implementation to verify that an encoding method is specified whenever there are encoded values present.

ABSTRACT TEST A.47: ELEMENTS OF A MATRIX ARE OF SCALAR TYPES OR NESTED MATRICES

IDENTIFIER	/conf/uml-block-components/matrix-element-type-valid
REQUIREMENT	Requirement 47: /req/uml-block-components/matrix-element-type-valid
TEST PURPOSE	Verify that the implementation of the “Matrix” class has a constraint that enforces the requirement.
TEST METHOD	Inspect the model or software implementation.

A.2.5. Conformance Class: Geometry Components UML Package

CONFORMANCE CLASS A.6: GEOMETRY COMPONENTS UML PACKAGE

IDENTIFIER	/conf/uml-geom-components
REQUIREMENTS CLASS	Requirements class 6: /req/uml-geom-components
PREREQUISITE	Conformance class A.2: /conf/uml-simple-components
TARGET TYPE	Derived Models and Software Implementations
CONFORMANCE TESTS	Abstract test A.48: /conf/uml-geom-components/package-fully-implemented Abstract test A.49: /conf/uml-geom-components/srs-valid Abstract test A.50: /conf/uml-geom-components/geom-value-valid

ABSTRACT TEST A.48

IDENTIFIER	/conf/uml-geom-components/package-fully-implemented
REQUIREMENT	Requirement 49: /req/uml-geom-components/package-fully-implemented
TEST PURPOSE	Verify that the target implements all classes in the UML package.
TEST METHOD	Inspect the model or software implementation.

ABSTRACT TEST A.49

IDENTIFIER	/conf/uml-geom-components/srs-valid
REQUIREMENT	Requirement 50: /req/uml-geom-components/srs-valid
TEST PURPOSE	Verify that the SRS is valid.
TEST METHOD	<ul style="list-style-type: none">Inspect the model or software implementation.Check that the “srs” attribute references a valid coordinate reference system.

ABSTRACT TEST A.50

IDENTIFIER	/conf/uml-geom-components/geom-value-valid
REQUIREMENT	Requirement 51: /req/uml-geom-components/geom-value-valid
TEST PURPOSE	Verify that the geometry value is valid.
TEST METHOD	<ul style="list-style-type: none">• Inspect the model or software implementation.• Check that the “value” attribute is either not set or contains a valid geometry object.

A.2.6. Conformance Class: Simple Encodings UML Package

CONFORMANCE CLASS A.7: SIMPLE ENCODINGS UML PACKAGE

IDENTIFIER	/conf/uml-simple-encodings
REQUIREMENTS CLASS	Requirements class 7: /req/uml-simple-encodings
PREREQUISITE	Conformance class A.1: /conf/core
TARGET TYPE	Derived Models and Software Implementations
CONFORMANCE TEST	Abstract test A.51: /conf/uml-simple-encodings/package-fully-implemented

ABSTRACT TEST A.51: COMPLIANCE WITH UML MODELS DEFINED IN THIS PACKAGE

IDENTIFIER	/conf/uml-simple-encodings/package-fully-implemented
REQUIREMENT	Requirement 52: /req/uml-simple-encodings/package-fully-implemented
TEST PURPOSE	Verify that the target implements all classes in the UML package.
TEST METHOD	Inspect the model or software implementation.

A.2.7. Conformance Class: Advanced Encodings UML Package

CONFORMANCE CLASS A.8: ADVANCED ENCODINGS UML PACKAGE

IDENTIFIER	/conf/uml-advanced-encodings
REQUIREMENTS CLASS	Requirements class 8: /req/uml-advanced-encodings
PREREQUISITE	Conformance class A.7: /conf/uml-simple-encodings
TARGET TYPE	Derived Models and Software Implementations
CONFORMANCE TEST	Abstract test A.52: /conf/uml-advanced-encodings/package-fully-implemented

ABSTRACT TEST A.52: COMPLIANCE WITH UML MODELS DEFINED IN THIS PACKAGE

IDENTIFIER	/conf/uml-advanced-encodings/package-fully-implemented
REQUIREMENT	Requirement 53: /req/uml-advanced-encodings/package-fully-implemented
TEST PURPOSE	Verify that the target implements all classes in the UML package.
TEST METHOD	Inspect the model or software implementation.

A.3. JSON Conformance Classes

A.3.1. Conformance Class: Basic Types and Simple Components JSON Schemas

CONFORMANCE CLASS A.9: BASIC TYPES AND SIMPLE COMPONENTS JSON SCHEMAS

IDENTIFIER	/conf/json-simple-components
REQUIREMENTS CLASS	Requirements class 9: /req/json-simple-components
INDIRECT PREREQUISITE	Conformance class A.1: /conf/core
TARGET TYPE	JSON Document

CONFORMANCE CLASS A.9: BASIC TYPES AND SIMPLE COMPONENTS JSON SCHEMAS

CONFORMANCE TESTS	Abstract test A.53: /conf/json-simple-components/component-types
	Abstract test A.54: /conf/json-simple-components/schema-valid
	Abstract test A.55: /conf/json-simple-components/special-numerical-values
	Abstract test A.56: /conf/json-simple-components/definition-resolvable
	Abstract test A.57: /conf/json-simple-components/inline-value-constraint-valid
	Abstract test A.58: /conf/json-simple-components/ucum-code-and-href-represent-same-unit
	Abstract test A.59: /conf/json-simple-components/ucum-code-used
	Abstract test A.60: /conf/json-simple-components/iso8601-uom-used

All tests in this conformance test class and in the following shall be used to check conformance of JSON documents created according to the schemas defined in this standard. They shall also be used to check conformance of software implementations that output JSON documents.

ABSTRACT TEST A.53

IDENTIFIER /conf/json-simple-components/component-types

REQUIREMENT Requirement 54: /req/json-simple-components/component-types

TEST METHOD Run the tests in this conformance class on a set of JSON documents containing instances of the following data component types: *Boolean*, *Text*, *Category*, *Count*, *Quantity*, *Time*, *CategoryRange*, *CountRange*, *QuantityRange*, *TimeRange*.

ABSTRACT TEST A.54: COMPLIANCE WITH JSON SCHEMAS

IDENTIFIER /conf/json-simple-components/schema-valid

REQUIREMENT Requirement 55: /req/json-simple-components/schema-valid

TEST PURPOSE Verify that the JSON document is valid against the schema.

TEST METHOD Validate the JSON document using the JSON schema "[sweCommon.json](#)".

ABSTRACT TEST A.55

IDENTIFIER /conf/json-simple-components/special-numerical-values

REQUIREMENT Requirement 56: /req/json-simple-components/special-numerical-values

TEST METHOD Check that special values -Infinity, +Infinity and NaN are supported by the standardization target.

ABSTRACT TEST A.56

IDENTIFIER /conf/json-simple-components/definition-resolvable

REQUIREMENT Requirement 57: /req/json-simple-components/definition-resolvable

TEST METHOD Check that the URI used for the definition attribute is either:

- An HTTP URL that resolves to a document (response code 200) containing a machine or human readable definition (can be RDF, GML, HTML, etc.).
- A URN that can be resolved to a document containing a machine or human readable definition, using a URN resolver provided separately.

ABSTRACT TEST A.57

IDENTIFIER /conf/json-simple-components/inline-value-constraint-valid

REQUIREMENT Requirement 58: /req/json-simple-components/inline-value-constraint-valid

TEST METHOD Check that the inline value provided as part of a data component satisfies the constraints of the component (if any).

ABSTRACT TEST A.58

IDENTIFIER /conf/json-simple-components/ucum-code-and-href-represent-same-unit

REQUIREMENT Requirement 59: /req/json-simple-components/ucum-code-and-href-represent-same-unit

TEST METHOD If the uom/code property is set to a UCUM code and the uom/href attribute is set to reference an external unit definition, check that both properties represent the same unit.

ABSTRACT TEST A.59

IDENTIFIER /conf/json-simple-components/ucum-code-used

REQUIREMENT Recommendation 1: /rec/json-simple-components/ucum-code-used

TEST METHOD If the uom attribute is set to a URI of a UCUM code, check that the specified unit cannot be represented by UCUM.

ABSTRACT TEST A.60

IDENTIFIER /conf/json-simple-components/iso8601-uom-used

REQUIREMENT Requirement 60: /req/json-simple-components/iso8601-uom-used

TEST METHOD Check that the uom attribute is set to the URI "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian" if the inline value is set to a ISO8601 string.

A.3.2. Conformance Class: Record Components JSON Schema

CONFORMANCE CLASS A.10: RECORD COMPONENTS JSON SCHEMA

IDENTIFIER /conf/json-record-components

REQUIREMENTS CLASS Requirements class 10: /req/json-record-components

PREREQUISITE Conformance class A.9: /conf/json-simple-components

TARGET TYPE JSON Document

CONFORMANCE TEST Abstract test A.61: /conf/json-record-components/component-types

ABSTRACT TEST A.61

IDENTIFIER /conf/json-record-components/component-types

REQUIREMENT Requirement 61: /req/json-record-components/component-types

TEST METHOD Run the tests in this conformance class (and its prerequisites) on a set of JSON documents containing instances of the following data component types: *DataRecord*, *Vector*

A.3.3. Conformance Class: Choice Components JSON Schema

CONFORMANCE CLASS A.11: CHOICE COMPONENTS JSON SCHEMA

IDENTIFIER	/conf/json-choice-components
REQUIREMENTS CLASS	Requirements class 11: /req/json-choice-components
PREREQUISITE	Conformance class A.9: /conf/json-simple-components
TARGET TYPE	JSON Document
CONFORMANCE TEST	Abstract test A.62: /conf/json-choice-components/component-types

ABSTRACT TEST A.62

IDENTIFIER	/conf/json-choice-components/component-types
REQUIREMENT	Requirement 62: /req/json-choice-components/component-types
TEST METHOD	Run the tests in this conformance class (and its prerequisites) on a set of JSON documents containing instances of the <i>DataChoice</i> component.

A.3.4. Conformance Class: Block Components JSON Schema

CONFORMANCE CLASS A.12: BLOCK COMPONENTS JSON SCHEMA

IDENTIFIER	/conf/json-block-components
REQUIREMENTS CLASS	Requirements class 12: /req/json-block-components
PREREQUISITES	Conformance class A.9: /conf/json-simple-components Conformance class A.14: /conf/json-simple-encodings
TARGET TYPE	JSON Document
CONFORMANCE TESTS	Abstract test A.63: /conf/json-block-components/component-types

CONFORMANCE CLASS A.12: BLOCK COMPONENTS JSON SCHEMA

Abstract test A.64: /conf/json-block-components/
encoded-values-valid
Abstract test A.65: /conf/json-block-components/
referenced-encoded-values-valid

ABSTRACT TEST A.63

IDENTIFIER /conf/json-block-components/component-types

REQUIREMENT Requirement 63: /req/json-block-components/component-types

TEST METHOD Run the tests in this conformance class (and its prerequisites) on a set of JSON documents containing instances of the following data component types: *DataArray*, *Matrix*, *DataStream*

ABSTRACT TEST A.64

IDENTIFIER /conf/json-block-components/encoded-values-valid

REQUIREMENT Requirement 64: /req/json-block-components/encoded-values-valid

TEST PURPOSE Verify that inline values provided in the JSON instance are valid.

TEST METHOD Run all tests from conformance class /conf/json-encoding-rules on the inline values.

ABSTRACT TEST A.65

IDENTIFIER /conf/json-block-components/referenced-encoded-values-valid

REQUIREMENT Requirement 65: /req/json-block-components/referenced-encoded-values-valid

TEST PURPOSE Verify that referenced values provided in the JSON instance are supported.

TEST METHOD Run the tests in this conformance class (and its prerequisites) on a set of JSON documents containing instances of the data component types *DataArray*, *Matrix* and *DataStream* having the value/href attribute set to URLs with the following schemes: http, https and data.

A.3.5. Conformance Class: Geometry Components JSON Schema

CONFORMANCE CLASS A.13: GEOMETRY COMPONENTS JSON SCHEMA

IDENTIFIER	/conf/json-geom-components
REQUIREMENTS CLASS	Requirements class 13: /req/json-geom-components
PREREQUISITES	Conformance class A.9: /conf/json-simple-components Conformance class A.14: /conf/json-simple-encodings
TARGET TYPE	JSON Document
CONFORMANCE TEST	Abstract test A.66: /conf/json-geom-components/component-types

ABSTRACT TEST A.66

IDENTIFIER	/conf/json-geom-components/component-types
REQUIREMENT	Requirement 66: /req/json-geom-components/component-types
TEST METHOD	Run the tests in this conformance class (and its prerequisites) on a set of JSON documents containing instances of the <i>Geometry</i> data component with the following value types: <i>Point</i> , <i>Line String</i> , <i>Polygon</i> .

A.3.6. Conformance Class: Simple Encodings JSON Schema

CONFORMANCE CLASS A.14: SIMPLE ENCODINGS JSON SCHEMA

IDENTIFIER	/conf/json-simple-encodings
REQUIREMENTS CLASS	Requirements class 14: /req/json-simple-encodings
PREREQUISITES	Conformance class A.18: /conf/text-encoding-rules Conformance class A.17: /conf/json-encoding-rules
TARGET TYPE	JSON Document
CONFORMANCE TESTS	Abstract test A.67: /conf/json-simple-encodings/encoding-types Abstract test A.68: /conf/json-simple-encodings/json-encoding-rules-applied

CONFORMANCE CLASS A.14: SIMPLE ENCODINGS JSON SCHEMA

Abstract test A.69: /conf/json-simple-encodings/
text-encoding-rules-applied

ABSTRACT TEST A.67

IDENTIFIER /conf/json-simple-encodings/encoding-types

REQUIREMENT Requirement 67: /req/json-simple-encodings/encoding-types

TEST METHOD Run the tests in this conformance class on a set of JSON documents containing block components with values encoded according to the following encoding types: *JSONEncoding*, *TextEncoding*

ABSTRACT TEST A.68

IDENTIFIER /conf/json-simple-encodings/json-encoding-rules-applied

REQUIREMENT Requirement 68: /req/json-simple-encodings/json-encoding-rules-applied

TEST PURPOSE Check that values of a block component are encoded according to the JSON encoding rules

TEST METHOD

1. Retrieve the content of the values property or the out-of-band data.
2. Check that the data fulfills all requirements from Conformance class A.17: /conf/json-encoding-rules

ABSTRACT TEST A.69

IDENTIFIER /conf/json-simple-encodings/text-encoding-rules-applied

REQUIREMENT Requirement 69: /req/json-simple-encodings/text-encoding-rules-applied

TEST PURPOSE Check that values of a block component are encoded according to the Text encoding rules

TEST METHOD

1. Retrieve the content of the out-of-band data.
2. Check that the data fulfills all requirements from Conformance class A.18: /conf/text-encoding-rules

A.3.7. Conformance Class: Advanced Encodings JSON Schema

CONFORMANCE CLASS A.15: ADVANCED ENCODINGS JSON SCHEMA

IDENTIFIER	/conf/json-advanced-encodings
REQUIREMENTS CLASS	Requirements class 15: /req/json-advanced-encodings
PREREQUISITES	Conformance class A.14: /conf/json-simple-encodings Conformance class A.19: /conf/binary-encoding-rules
TARGET TYPE	JSON Document
CONFORMANCE TESTS	Abstract test A.70: /conf/json-advanced-encodings/encoding-types Abstract test A.71: /conf/json-advanced-encodings/binary-encoding-rules-applied Abstract test A.72: /conf/json-advanced-encodings/ref-syntax-valid Abstract test A.73: /conf/json-advanced-encodings/scalar-ref-component-valid Abstract test A.74: /conf/json-advanced-encodings/datatype-valid Abstract test A.75: /conf/json-advanced-encodings/datatype-compatible Abstract test A.76: /conf/json-advanced-encodings/no-datatype-length Abstract test A.77: /conf/json-advanced-encodings/block-ref-component-valid

ABSTRACT TEST A.70

IDENTIFIER	/conf/json-advanced-encodings/encoding-types
REQUIREMENT	Requirement 70: /req/json-advanced-encodings/encoding-types
TEST PURPOSE	Check that the standardization target supports the BinaryEncoding option.
TEST METHOD	Run the tests in this conformance class on a set of JSON documents containing block components with values encoded according to the following encoding types: <i>BinaryEncoding</i>

ABSTRACT TEST A.71

IDENTIFIER	/conf/json-advanced-encodings/binary-encoding-rules-applied
REQUIREMENT	Requirement 71: /req/json-advanced-encodings/binary-encoding-rules-applied

ABSTRACT TEST A.71

TEST PURPOSE Check that values are encoded as defined by the binary encoding rules.

TEST METHOD Find all binary encoded value blocks included inline (base64 encoded) or referenced by the JSON document. Apply all tests from Conformance class A.19: /conf/binary-encoding-rules to the encoded data to validate its syntax and structure.

ABSTRACT TEST A.72

IDENTIFIER /conf/json-advanced-encodings/ref-syntax-valid

REQUIREMENT Requirement 72: /req/json-advanced-encodings/ref-syntax-valid

TEST PURPOSE Check that the path specified by the ref attribute has the correct syntax.

TEST METHOD

1. Inspect the section of the JSON instance describing the binary encoding options.
2. Check that the path formed by the '/' separated list of component names actually points to a component in the descriptor tree.

ABSTRACT TEST A.73

IDENTIFIER /conf/json-advanced-encodings/scalar-ref-component-valid

REQUIREMENT Requirement 73: /req/json-advanced-encodings/scalar-ref-component-valid

TEST PURPOSE Check that the path specified by the ref attribute points to a valid component.

TEST METHOD

1. Inspect the section of the JSON instance describing the BinaryComponent encoding options.
2. Resolve the path specified by the 'ref' attribute to a component of the dataset definition tree.
3. Verify that the component is a simple component, that is to say it is either a *Boolean*, *Count*, *Quantity*, *Time*, *Category*, *Text*, *CountRange*, *QuantityRange*, *TimeRange* or *CategoryRange*.

ABSTRACT TEST A.74

IDENTIFIER /conf/json-advanced-encodings/datatype-valid

REQUIREMENT Requirement 74: /req/json-advanced-encodings/datatype-valid

TEST PURPOSE Check that the chosen datatype is valid.

ABSTRACT TEST A.74

TEST METHOD Verify that the URI used to specify the binary data type is one of the item the list provided in Table 2.

ABSTRACT TEST A.75

IDENTIFIER /conf/json-advanced-encodings/datatype-compatible

REQUIREMENT Requirement 75: /req/json-advanced-encodings/datatype-compatible

TEST PURPOSE Check that the chosen datatype is compatible with the associated component.

TEST METHOD

For text components (i.e., “Category”, “Text” or “Time” with ISO-8601 encoding), verify that the data type is one of the string types.

For scalar numerical components (i.e., “Quantity”, “Count” or “Time” with a simple unit), verify that:

- The data type is also numerical (i.e., one of the integer or floating point types)
- The range of values it allows can cover all possible numbers within the allowed intervals and enumerated values (e.g., A short data type cannot be used for an interval constraint of [-100000; 10000]). When no interval constraint is specified, this test should be ignored.
- The data type can accommodate the desired precision indicated by the “significantFigures” constraint (e.g., a float cannot be used for a number of significant figures greater than 7). When no precision constraint is specified, this test should be ignored.

For a boolean component, verify that the data type is an unsigned byte (<http://www.opengis.net/def/dataType/OGC/0/unsignedByte>).

ABSTRACT TEST A.76

IDENTIFIER /conf/json-advanced-encodings/no-datatype-length

REQUIREMENT Requirement 76: /req/json-advanced-encodings/no-datatype-length

TEST PURPOSE Check that the length of a datatype is specified only when appropriate.

TEST METHOD Verify that the “bitLength” and “byteLength” attributes are used only when one of the UTF-8 String or Custom Integer data types is selected.

ABSTRACT TEST A.77

IDENTIFIER /conf/json-advanced-encodings/block-ref-component-valid

ABSTRACT TEST A.77

REQUIREMENT Requirement 77: /req/json-advanced-encodings/block-ref-component-valid

TEST PURPOSE Check that the binary block encoding specifications are associated to an aggregate component

TEST METHOD

1. Inspect the section of the JSON instance describing the BinaryBlock encoding options.
2. Resolve the path specified by the 'ref' attribute to a component of the dataset definition tree.
3. Verify that the component is an aggregate, that is to say it is either a *DataRecord*, *Vector*, *DataChoice*, *DataArray* or *Matrix*.

A.4. Datastream Encoding Conformance Classes

A.4.1. Conformance Class: General Encoding Rules

CONFORMANCE CLASS A.16: GENERAL ENCODING RULES

IDENTIFIER /conf/general-encoding-rules

REQUIREMENTS CLASS Requirements class 16: /req/general-encoding-rules

TARGET TYPE Encoded Values Instance

CONFORMANCE TESTS

- Abstract test A.78: /conf/general-encoding-rules/record-encoding-rule
- Abstract test A.79: /conf/general-encoding-rules/choice-encoding-rule
- Abstract test A.80: /conf/general-encoding-rules/array-encoding-rule
- Abstract test A.81: /conf/general-encoding-rules/array-size-encoding-rule

ABSTRACT TEST A.78: DATARECORD FIELDS AND VECTOR COORDINATES ARE ENCODED RECURSIVELY

IDENTIFIER /conf/general-encoding-rules/record-encoding-rule

REQUIREMENT Requirement 78: /req/general-encoding-rules/record-encoding-rule

ABSTRACT TEST A.78: DATARECORD FIELDS AND VECTOR COORDINATES ARE ENCODED RECURSIVELY

TEST PURPOSE	Verify that encoding rules are implemented correctly
TEST METHOD	Verify that the sequence of scalar values (obtained after decoding the section of the encoded data block corresponding to the “DataRecord” or “Vector”) includes values for the successive fields/coordinates in the right order.

ABSTRACT TEST A.79: DATACHOICE SELECTED ITEM IS PROPERLY ENCODED

IDENTIFIER	/conf/general-encoding-rules/choice-encoding-rule
REQUIREMENT	Requirement 79: /req/general-encoding-rules/choice-encoding-rule
TEST PURPOSE	Verify that encoding rules are implemented correctly
TEST METHOD	Verify that the sequence of scalar values (obtained after decoding the section of the encoded data block corresponding to the “DataChoice”) includes a value identifying the selected item as well as values for the item itself.

ABSTRACT TEST A.80: DATAARRAY ELEMENTS ARE ENCODED RECURSIVELY

IDENTIFIER	/conf/general-encoding-rules/array-encoding-rule
REQUIREMENT	Requirement 80: /req/general-encoding-rules/array-encoding-rule
TEST PURPOSE	Verify that encoding rules are implemented correctly
TEST METHOD	Verify that the sequence of scalar values obtained after decoding the section of the encoded data block corresponding to the “DataArray” includes values for the successive elements of the array.

ABSTRACT TEST A.81: THE LENGTH OF VARIABLE SIZE ARRAYS IS ENCODED IN THE DATA BLOCK

IDENTIFIER	/conf/general-encoding-rules/array-size-encoding-rule
REQUIREMENT	Requirement 81: /req/general-encoding-rules/array-size-encoding-rule
TEST PURPOSE	Verify that encoding rules are implemented correctly

ABSTRACT TEST A.81: THE LENGTH OF VARIABLE SIZE ARRAYS IS ENCODED IN THE DATA BLOCK

TEST METHOD	Verify that the sequence of values obtained after decoding the section of the encoded data block corresponding to a variable size "DataArray" includes a value specifying the size of the array.
-------------	--

A.4.2. Conformance Class: JSON Encoding Rules

CONFORMANCE CLASS A.17: JSON ENCODING RULES

IDENTIFIER	/conf/json-encoding-rules
REQUIREMENTS CLASS	Requirements class 17: /req/json-encoding-rules
PREREQUISITE	Conformance class A.16: /conf/general-encoding-rules
TARGET TYPE	Encoded Values Instance
CONFORMANCE TESTS	Abstract test A.82: /conf/json-encoding-rules/json-valid Abstract test A.83: /conf/json-encoding-rules/scalar-value-valid Abstract test A.84: /conf/json-encoding-rules/range-value-valid Abstract test A.85: /conf/json-encoding-rules/record-object-valid Abstract test A.86: /conf/json-encoding-rules/vector-object-valid Abstract test A.87: /conf/json-encoding-rules/choice-object-valid Abstract test A.88: /conf/json-encoding-rules/array-values-valid Abstract test A.89: /conf/json-encoding-rules/geometry-valid

ABSTRACT TEST A.82

IDENTIFIER	/conf/json-encoding-rules/json-valid
REQUIREMENT	Requirement 82: /req/json-encoding-rules/json-valid
TEST PURPOSE	Verify that encoding rules are implemented correctly

ABSTRACT TEST A.82

TEST METHOD Verify that the data in the encoded data block is valid JSON with a JSON validator.

ABSTRACT TEST A.83

IDENTIFIER /conf/json-encoding-rules/scalar-value-valid

REQUIREMENT Requirement 83: /req/json-encoding-rules/scalar-value-valid

TEST PURPOSE Verify that scalar encoding rules are implemented correctly

TEST METHOD Inspect the JSON of the encoded data block to verify that the value corresponding to a scalar component is encoded with the data type specified in Table 3.

ABSTRACT TEST A.84

IDENTIFIER /conf/json-encoding-rules/range-value-valid

REQUIREMENT Requirement 84: /req/json-encoding-rules/range-value-valid

TEST PURPOSE Verify that range encoding rules are implemented correctly

TEST METHOD Inspect the JSON of the encoded data block to verify that:

- Each JSON value corresponding to a range component is a JSON Array with two values.
- Each value in the array has the proper data type as specified in Table 3.

ABSTRACT TEST A.85

IDENTIFIER /conf/json-encoding-rules/record-object-valid

REQUIREMENT Requirement 85: /req/json-encoding-rules/record-object-valid

TEST PURPOSE Verify that record encoding rules are implemented correctly

TEST METHOD Inspect the JSON of the encoded data block to verify that:

- Each JSON value corresponding to a record component is a JSON Object
- This JSON object has members with the same names as the fields in the vector description
- Only members corresponding to a field marked as optional are omitted
- The value of each JSON member is valid according to the field descriptor

ABSTRACT TEST A.86

IDENTIFIER /conf/json-encoding-rules/vector-object-valid

REQUIREMENT Requirement 86: /req/json-encoding-rules/vector-object-valid

TEST PURPOSE Verify that vector encoding rules are implemented correctly

TEST METHOD

Inspect the JSON of the encoded data block to verify that:

- Each JSON value corresponding to a vector component is a JSON Object
- This JSON object has members with the same names as the coordinates in the vector description
- The value of each JSON member is valid according to the coordinate descriptor

ABSTRACT TEST A.87

IDENTIFIER /conf/json-encoding-rules/choice-object-valid

REQUIREMENT Requirement 87: /req/json-encoding-rules/choice-object-valid

TEST PURPOSE Verify that choice encoding rules are implemented correctly

TEST METHOD

Inspect the JSON of the encoded data block to verify that:

- Each JSON value corresponding to a choice component is a JSON Object
- This JSON object contains a single member and its name is the same as one of the items in the choice description.
- The value of the JSON member is valid according to the corresponding item descriptor.

ABSTRACT TEST A.88

IDENTIFIER /conf/json-encoding-rules/array-values-valid

REQUIREMENT Requirement 88: /req/json-encoding-rules/array-values-valid

TEST PURPOSE Verify that array encoding rules are implemented correctly

TEST METHOD

Inspect the JSON of the encoded data block to verify that:

- Each JSON value corresponding to an array or matrix component is a JSON Array
- All values in the array are valid according to the array element descriptor
- If the array has a fixed size, the number of elements in the array is equal to the size

ABSTRACT TEST A.89

IDENTIFIER /conf/json-encoding-rules/geometry-valid

REQUIREMENT Requirement 89: /req/json-encoding-rules/geometry-valid

TEST PURPOSE Verify that geometry encoding rules are implemented correctly

TEST METHOD

Inspect the JSON of the encoded data block to verify that:

- Each JSON value corresponding to a geometry component is a JSON Object
- The JSON Object is a valid GeoJSON Geometry
- The GeoJSON geometry type is either Point, LineString, Polygon, MultiPoint, MultiLineString, or MultiPolygon
- The number of dimensions in the GeoJSON geometry is the same as the one specified by the CRS of the Geometry descriptor.

A.4.3. Conformance Class: Text Encoding Rules

CONFORMANCE CLASS A.18: TEXT ENCODING RULES

IDENTIFIER /conf/text-encoding-rules

REQUIREMENTS CLASS Requirements class 18: /req/text-encoding-rules

PREREQUISITE Conformance class A.16: /conf/general-encoding-rules

TARGET TYPE Encoded Values Instance

CONFORMANCE TESTS

Abstract test A.90: /conf/text-encoding-rules/abnf-syntax-valid
Abstract test A.91: /conf/text-encoding-rules/separators-valid
Abstract test A.92: /conf/text-encoding-rules/optional-field-marker-present
Abstract test A.93: /conf/text-encoding-rules/choice-selection-marker-valid
Abstract test A.94: /conf/text-encoding-rules/geometry-valid

ABSTRACT TEST A.90: COMPLIANCE WITH ABNF GRAMMAR

IDENTIFIER /conf/text-encoding-rules/abnf-syntax-valid

ABSTRACT TEST A.90: COMPLIANCE WITH ABNF GRAMMAR

REQUIREMENT	Requirement 90: /req/text-encoding-rules/abnf-syntax-valid
TEST PURPOSE	Verify that encoding rules are implemented correctly
TEST METHOD	Verify that the text encoded data block is correct with respect to the ABNF grammar corresponding to the particular dataset (The complete ABNF grammar of the dataset should be dynamically constructed from the ABNF snippets provided in the specification).

ABSTRACT TEST A.91: SEPARATOR CHARACTERS ARE WELL CHOSEN

IDENTIFIER	/conf/text-encoding-rules/separators-valid
REQUIREMENT	Requirement 91: /req/text-encoding-rules/separators-valid
TEST PURPOSE	Verify that encoding rules are implemented correctly
TEST METHOD	Verify that the values encoded in the data block never include the reserved separator characters. This can be detected by looking for invalid or superfluous values.

ABSTRACT TEST A.92: SPECIAL FLAGS ARE INSERTED BEFORE OPTIONAL COMPONENT VALUES

IDENTIFIER	/conf/text-encoding-rules/optional-field-marker-present
REQUIREMENT	Requirement 92: /req/text-encoding-rules/optional-field-marker-present
TEST PURPOSE	Verify that encoding rules are implemented correctly
TEST METHOD	Verify that the sequence of values corresponding to the optional field starts with the 'Y' or 'N' flag.

ABSTRACT TEST A.93: THE NAME OF A SELECTED CHOICE ITEM IS INSERTED IN THE STREAM

IDENTIFIER	/conf/text-encoding-rules/choice-selection-marker-valid
REQUIREMENT	Requirement 93: /req/text-encoding-rules/choice-selection-marker-valid
TEST PURPOSE	Verify that encoding rules are implemented correctly

ABSTRACT TEST A.93: THE NAME OF A SELECTED CHOICE ITEM IS INSERTED IN THE STREAM

TEST METHOD

Verify that the sequence of values corresponding to the “DataChoice” starts with a character string matching the name of one item of the choice descriptor.

ABSTRACT TEST A.94

IDENTIFIER /conf/text-encoding-rules/geometry-valid

REQUIREMENT Requirement 94: /req/text-encoding-rules/geometry-valid

TEST PURPOSE Check that the encoded geometry value is valid

TEST METHOD

1. Verify that the geometry value is a valid WKT string.
2. Verify that the number of dimensions in the WKT is compatible with the specified geometry SRS.

A.4.4. Conformance Class: Binary Encoding Rules

CONFORMANCE CLASS A.19: BINARY ENCODING RULES

IDENTIFIER /conf/binary-encoding-rules

REQUIREMENTS CLASS Requirements class 19: /req/binary-encoding-rules

PREREQUISITE Conformance class A.16: /conf/general-encoding-rules

TARGET TYPE Encoded Values Instance

CONFORMANCE TESTS

Abstract test A.95: /conf/binary-encoding-rules/abnf-syntax-valid
Abstract test A.96: /conf/binary-encoding-rules/type-encoding-valid
Abstract test A.97: /conf/binary-encoding-rules/base64-translation-applied
Abstract test A.98: /conf/binary-encoding-rules/optional-field-marker-present
Abstract test A.99: /conf/binary-encoding-rules/choice-selection-marker-valid
Abstract test A.100: /conf/binary-encoding-rules/geometry-valid

ABSTRACT TEST A.95: COMPLIANCE WITH ABNF GRAMMAR

IDENTIFIER	/conf/binary-encoding-rules/abnf-syntax-valid
REQUIREMENT	Requirement 95: /req/binary-encoding-rules/abnf-syntax-valid
TEST PURPOSE	Verify that encoding rules are implemented correctly
TEST METHOD	Verify that the binary encoded data block is correct with respect to the ABNF grammar of the particular dataset (The complete ABNF grammar of the dataset should be dynamically constructed from the ABNF snippets provided in the specification).

ABSTRACT TEST A.96: DATA TYPES ARE ENCODED AS SPECIFIED IN THIS STANDARD

IDENTIFIER	/conf/binary-encoding-rules/type-encoding-valid
REQUIREMENT	Requirement 96: /req/binary-encoding-rules/type-encoding-valid
TEST PURPOSE	Verify that encoding rules are implemented correctly
TEST METHOD	Verify that valid and realistic scalar values are obtained when the binary data block is parsed by extracting the number of bits specified in the table and decoding the resulting bytes in the order specified by the "byteOrder" attribute. When the encoded data and the encoding parameters are not consistent, aberrant values (such as -65502 for a temperature field, etc...) are usually obtained, which can be easily detected.

ABSTRACT TEST A.97: BASE64 ENCODING IS IMPLEMENTED AS DEFINED BY IETF

IDENTIFIER	/conf/binary-encoding-rules/base64-translation-applied
REQUIREMENT	Requirement 97: /req/binary-encoding-rules/base64-translation-applied
TEST PURPOSE	Verify that encoding rules are implemented correctly
TEST METHOD	<ul style="list-style-type: none">• Verify that only characters allowed by base64 encoding are used in the encoded data content.• Verify that the data block can be properly parsed after the base64 data is decoded into a raw binary data stream.

ABSTRACT TEST A.98: SPECIAL FLAGS ARE INSERTED BEFORE OPTIONAL COMPONENT VALUES

IDENTIFIER	/conf/binary-encoding-rules/optional-field-marker-present
REQUIREMENT	Requirement 98: /req/binary-encoding-rules/optional-field-marker-present
TEST PURPOSE	Verify that encoding rules are implemented correctly
TEST METHOD	<ul style="list-style-type: none">• Verify that any optional field is preceded by the a 1-byte ASCII character with value 'Y' or 'N'.• Verify that the actual field value is only present if the flag has the 'Y' value.

ABSTRACT TEST A.99: THE NAME OF A SELECTED CHOICE ITEM IS INSERTED IN THE STREAM

IDENTIFIER	/conf/binary-encoding-rules/choice-selection-marker-valid
REQUIREMENT	Requirement 99: /req/binary-encoding-rules/choice-selection-marker-valid
TEST PURPOSE	Verify that encoding rules are implemented correctly
TEST METHOD	<ul style="list-style-type: none">• Verify that the sequence of bytes corresponding to the “DataChoice” starts with a byte value that is greater or equal to 0 and less than the total number of items defined in the choice descriptor.• Verify that the parsed index value corresponds to the proper item in the choice descriptor.

ABSTRACT TEST A.100

IDENTIFIER	/conf/binary-encoding-rules/geometry-valid
REQUIREMENT	Requirement 100: /req/binary-encoding-rules/geometry-valid
TEST PURPOSE	Check that the encoded geometry value is valid
TEST METHOD	<ol style="list-style-type: none">1. Verify that the geometry value is valid WKB data.2. Verify that the number of dimensions in the WKB data is compatible with the specified geometry SRS.



ANNEX B (INFORMATIVE) EXAMPLES

B

ANNEX B (INFORMATIVE) EXAMPLES

B.1. Text Encoding Rules Examples

B.1.1. DataArray with inline values (curve)

The following example shows how elements of an array defined as a “DataRecord” can be encoded inline with the text method:

```
{
  "type": "DataArray",
  "definition": "http://sweet.jpl.nasa.gov/2.0/mathFunction.owl#Function",
  "description": "Measurement error vs. temperature",
  "elementCount": { "value": 5 },
  "elementType": {
    "type": "DataRecord",
    "name": "point",
    "label": "Error vs. Temperature",
    "fields": [
      {
        "type": "Quantity",
        "name": "temp",
        "label": "Temperature",
        "definition": "http://sweet.jpl.nasa.gov/2.0/physThermo.owl#Temperature",
        "uom": { "code": "Cel" }
      },
      {
        "type": "Quantity",
        "name": "error",
        "label": "Relative Error",
        "definition": "http://sweet.jpl.nasa.gov/2.0/sciUncertainty.owl#Error",
        "uom": { "code": "%" }
      }
    ]
  },
  "encoding": {
    "type": "TextEncoding",
    "blockSeparator": " ",
    "tokenSeparator": ","
  }
}
```

```
}
```

Listing

The data array values encoded using the text encoding method as shown below:

```
0,5 10,2 50,2 80,5 100,15
```

Listing

In this example, each element consists of a record of two values. The array element structure also corresponds to one block so that tuples are separated by block separators (here the ‘;’ character). Since the array is of size 5, there are 5 tuples listed sequentially in the data block, each one composed of the two values of the data record separated by the token separator. The pattern is “temp,error temp,error ...” since values have to be listed in the same order as the fields.

B.1.2. Datastream with records (weather data)

The following snippet defines a datastream with an element of type record:

```
{
  "type": "DataStream",
  "label": "Weather Data",
  "elementType": {
    "type": "DataRecord",
    "name": "weatherData",
    "fields": [
      {
        "type": "Time",
        "name": "time",
        "label": "Time",
        "definition": "http://www.opengis.net/def/property/OGC/0/SamplingTime",
        "referenceFrame": "http://www.opengis.net/def/trs/BIPM/0/UTC",
        "uom": { "href": "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian" }
      },
      {
        "type": "Quantity",
        "name": "temp",
        "label": "Air Temperature",
        "definition": "http://mmisw.org/ont/cf/parameter/air_temperature",
        "uom": { "code": "Cel" }
      },
      {
        "type": "Quantity",
        "name": "press",
        "label": "Atmospheric Pressure",
        "definition": "http://mmisw.org/ont/cf/parameter/air_pressure_at_mean_sea_level",
        "uom": { "code": "HPa" }
      },
      {
        "type": "Quantity",
        "name": "windSpeed",
        "label": "Wind Speed",
        "definition": "http://mmisw.org/ont/cf/parameter/wind_speed",
        "uom": { "code": "km/h" }
      }
    ]
  }
}
```

```

        "type": "Quantity",
        "name": "windDir",
        "label": "Wind Direction",
        "definition": "http://mmisw.org/ont/cf/parameter/wind_to_direction",
        "uom": { "code": "deg" }
    }
]
},
"encoding": {
    "type": "TextEncoding",
    "blockSeparator": "\n",
    "tokenSeparator": ",",
}
}

```

Listing

The datastream records are encoded using the Text encoding method as shown below:

```

2023-03-20T15:40:00Z,15.3,1014,3.5,56.0
2023-03-20T15:45:00Z,15.4,1015,5.6,123.0
2023-03-20T15:50:00Z,15.8,1014,13.2,34.0
...

```

Listing

B.1.3. Datastream with records and optional fields (navigation data)

The following snippet defines a datastream with an element of type record that contains optional fields:

```

{
    "type": "DataStream",
    "label": "Aircraft Navigation",
    "elementType": {
        "type": "DataRecord",
        "name": "navData",
        "fields": [
            {
                "type": "Time",
                "name": "time",
                "label": "time",
                "definition": "http://www.opengis.net/def/property/OGC/0/SamplingTime",
                "referenceFrame": "http://www.opengis.net/def/trs/OGC/0/GPS",
                "uom": {
                    "href": "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian"
                }
            },
            {
                "type": "Quantity",
                "name": "speed",
                "label": "speed",
                "definition": "http://sweet.jpl.nasa.gov/2.0/humanTransportAir.owl#GroundSpeed",
                "uom": { "code": "m/s" }
            },
            {
                "type": "Vector",
                "name": "location",
                "label": "location",
            }
        ]
    }
}

```

```

        "definition": "http://sweet.jpl.nasa.gov/2.0/spaceCoordinates.
owl#Location",
        "referenceFrame": "http://www.opengis.net/def/crs/EPSG/0/4979",
        "optional": true,
        "coordinates": [
            {
                "type": "Quantity",
                "name": "lat",
                "label": "lat",
                "definition": "http://sweet.jpl.nasa.gov/2.0/spaceCoordinates.
owl#Latitude",
                "axisID": "Lat",
                "uom": { "code": "deg" }
            },
            {
                "type": "Quantity",
                "name": "lon",
                "label": "lon",
                "definition": "http://sweet.jpl.nasa.gov/2.0/spaceCoordinates.
owl#Longitude",
                "axisID": "Long",
                "uom": { "code": "deg" }
            },
            {
                "type": "Quantity",
                "name": "alt",
                "label": "alt",
                "definition": "http://sweet.jpl.nasa.gov/2.0/spaceExtent.
owl#Altitude",
                "axisID": "h",
                "uom": { "code": "m" }
            }
        ]
    },
    "encoding": {
        "type": "TextEncoding",
        "tokenSeparator": ",",
        "blockSeparator": "\n"
    }
}

```

Listing

The datastream records are encoded using the Text encoding method as shown below:

```

2007-10-23T15:46:12Z,15.3,Y,45.3,-90.5,311
2007-10-23T15:46:22Z,25.3,N
2007-10-23T15:46:32Z,20.6,Y,45.3,-90.6,312
2007-10-23T15:46:52Z,18.9,Y,45.4,-90.6,315
2007-10-23T15:47:02Z,22.3,N
...

```

Listing

In this example, the whole location “Vector” is marked as optional and thus the coordinate values are only included when the optional flag is set to ‘Y’ in the stream. Field values in each block have to be listed in the same order as the field properties in the record definition thus following the “time,speed,Y,lat,lon,alt” or “time,speed,N” pattern depending on whether or not the location is omitted.

B.1.4. Datastream with choice (navigation data)

This is illustrated by the following example:

```
{
  "type": "DataStream",
  "elementType": {
    "type": "DataChoice",
    "name": "message",
    "label": "Message",
    "items": [
      {
        "type": "DataRecord",
        "name": "TEMP",
        "label": "Temperature Measurement",
        "fields": [
          {
            "type": "Time",
            "name": "time",
            "label": "Time",
            "definition": "http://www.opengis.net/def/property/OGC/0/
SamplingTime",
            "uom": {
              "href": "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian"
            }
          },
          {
            "type": "Quantity",
            "name": "temp",
            "label": "Temperature",
            "definition": "http://mmisw.org/ont/cf/parameter/air_temperature",
            "uom": { "code": "Cel" }
          }
        ]
      },
      {
        "type": "DataRecord",
        "name": "WIND",
        "label": "Wind Measurement",
        "fields": [
          {
            "type": "Time",
            "name": "time",
            "label": "Time",
            "definition": "http://www.opengis.net/def/property/OGC/0/
SamplingTime",
            "uom": {
              "href": "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian"
            }
          },
          {
            "type": "Quantity",
            "name": "wind_speed",
            "label": "Wind Speed",
            "definition": "http://mmisw.org/ont/cf/parameter/wind_speed",
            "uom": { "code": "km/h" }
          },
          {
            "type": "Quantity",
            "name": "wind_dir",
```

```

        "label": "Wind Direction",
        "definition": "http://mmisw.org/ont/cf/parameter/wind_to_direction",
        "uom": { "code": "deg" }
      }
    ]
  },
  "encoding": {
    "type": "TextEncoding",
    "blockSeparator": "\n",
    "tokenSeparator": ",",
  }
}

```

Listing

The datastream records are encoded using the Text encoding method as shown below:

```

TEMP,2009-05-23T19:36:15Z,25.5
TEMP,2009-05-23T19:37:15Z,25.6
WIND,2009-05-23T19:37:17Z,56.3,226.3
TEMP,2009-05-23T19:38:15Z,25.5
...

```

Listing

This datastream interleaves different types of messages separated by the block separator character. The element type is a “DataChoice” which means that each encoded block is composed of the item name ‘TEMP’ or ‘WIND’, followed by values of the item. This example also demonstrates that items of a choice can be of different types and length.

B.1.5. Fixed size 2D array (stress matrix)

The following example illustrates how values of a fixed size 3×3 stress matrix can be text encoded inline:

```

{
  "type": "Matrix",
  "definition": "http://sweet.jpl.nasa.gov/2.0/physPressure.owl#Stress",
  "elementCount": { "value": 3 },
  "elementType": {
    "type": "Matrix",
    "name": "row",
    "definition": "http://sweet.jpl.nasa.gov/2.0/info.owl#Row",
    "elementCount": { "value": 3 },
    "elementType": {
      "type": "Quantity",
      "name": "coef",
      "label": "Coefficient",
      "definition": "http://sweet.jpl.nasa.gov/2.0/mathVector.owl#Coordinate",
      "uom": { "code": "MPa" }
    }
  }
},
"encoding": {
  "type": "TextEncoding",
  "blockSeparator": " ",
  "tokenSeparator": ",",
}

```

```
}
```

Listing

The matrix encoded using the Text encoding method as shown below:

```
0.36,0.48,-0.8 -0.8,0.6,0.0 0.48,0.64,0.6
```

Listing

Note that elements of the outer array (i.e., a matrix is a special kind of array) are separated by block separators (i.e., each block surrounded by spaces corresponds to one row of the matrix) while the inner array elements are separated by token separators.

B.1.6. Datastream of variable size 1D arrays (profile series)

The following example shows how SWE Common can be used to encode a series of irregular length profiles by using a variable size array:

```
{
  "type": "DataStream",
  "elementType": {
    "type": "DataRecord",
    "name": "profileData",
    "fields": [
      {
        "type": "Time",
        "name": "time",
        "label": "Time",
        "definition": "http://www.opengis.net/def/property/OGC/0/SamplingTime",
        "uom": { "href": "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian" }
      },
      {
        "type": "DataArray",
        "name": "profilePoints",
        "definition": "http://sweet.jpl.nasa.gov/2.0/info.owl#Profile",
        "elementType": {
          "type": "DataRecord",
          "name": "point",
          "label": "Point",
          "fields": [
            {
              "type": "Quantity",
              "name": "depth",
              "label": "Sampling Point Vertical Location",
              "definition": "http://mmisw.org/ont/cf/parameter/depth",
              "uom": { "code": "m" }
            },
            {
              "type": "Quantity",
              "name": "salinity",
              "label": "Salinity",
              "definition": "http://mmisw.org/ont/cf/parameter#sea_water_salinity",
              "uom": { "code": "[ppth]" }
            }
          ]
        }
      }
    ]
  }
}
```

```

    ]
  },
  "encoding": {
    "type": "TextEncoding",
    "blockSeparator": "@@\\n",
    "tokenSeparator": ",",
  }
}

```

Listing

The datastream records are encoded using the Text encoding method as shown below:

```

2005-05-16T21:47:12Z,5,0,45,10,20,20,30,30,35,40,40@@@
2005-05-16T22:43:05Z,4,0,45,10,20,20,30,30,35@@@
2005-05-16T23:40:52Z,5,0,45,10,20,20,30,30,35,40,40
...

```

Listing

The example shows data for 3 profiles with a variable number of measurements along the vertical dimension. The number of measurements is indicated in the encoded data block by a number inserted after the timestamp, and before the measurements themselves. Since the array is itself the element of a “DataStream”, elements of the array are separated by token separators.

B.1.7. Datastream with geometry (feature detection)

The following snippet is an example of datastream that contains a geometry. Here, each datastream record represents a feature detected in a video stream, and is composed of a timestamp, a scalar field and the geometry of the geolocated feature.

```

{
  "type": "DataStream",
  "label": "Feature Detections",
  "elementType": {
    "type": "DataRecord",
    "name": "detection",
    "fields": [
      {
        "type": "Time",
        "name": "time",
        "label": "Time",
        "definition": "http://www.opengis.net/def/property/OGC/0/SamplingTime",
        "referenceFrame": "http://www.opengis.net/def/trs/OGC/0/GPS",
        "uom": { "href": "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian" }
      },
      {
        "type": "Category",
        "name": "type",
        "label": "Feature Type",
        "definition": "http://www.opengis.net/def/featureType",
        "codeSpace": "http://x-myorg.net/def/VehicleTypes"
      },
      {
        "type": "Geometry",
        "definition": "http://www.opengis.net/def/property/OGC/0/Geometry",
        "srs": "http://www.opengis.net/def/crs/EPSG/0/4326",
        "constraint": {
          "geomTypes": ["Point", "Polygon"]
        }
      }
    ]
  }
}

```

```

    }
  }
]
},
"encoding": {
  "type": "TextEncoding",
  "blockSeparator": ";\n",
  "tokenSeparator": ";"
}
}

```

Listing

The datastream records are encoded using the Text encoding method as shown below:

```

2007-10-23T15:46:12Z;Car;POINT(-86.3254 35.4812)
2007-10-23T15:49:03Z;Truck;POLYGON((-86.3254 35.4812,-86.3253 35.4812,-86.3253
35.4811,-86.3254 35.4811,-86.3254 35.4812))
2007-10-23T15:56:45Z;Bus;POLYGON((-86.3254 35.4812,-86.3253 35.4812,-86.3253
35.4811,-86.3254 35.4811,-86.3254 35.4812))
...

```

Listing

B.2. JSON Encoding Rules Examples

The following examples build on the ones provided in the Text Encoding Rules Examples section. The datastream descriptions are kept the same, except that the encoding method would have to be changed to JSONEncoding (which is the default).

In the following sections, encoded values were kept identical to the ones used in the text encoding section, in order to facilitate comparison.

B.2.1. DataArray with inline values (curve)

This example is based on the same “DataArray” description as the one provided in Annex B.1.1.

The equivalent JSON description for this “DataArray” is provided below:

```

{
  "type": "DataArray",
  "definition": "http://sweet.jpl.nasa.gov/2.0/mathFunction.owl#Function",
  "description": "Measurement error vs. temperature",
  "elementCount": {
    "type": "Count",
    "value": 5
  },
  "elementType": {
    "name": "point",
    "type": "DataRecord",
    "label": "Error vs. Temperature",
    "fields": [
      {
        "name": "temp",

```

```

        "type": "Quantity",
        "definition": "http://sweet.jpl.nasa.gov/2.0/physThermo.owl#Temperature",
        "label": "Temperature",
        "uom": { "code": "Cel" }
    },
    {
        "name": "error",
        "type": "Quantity",
        "definition": "http://sweet.jpl.nasa.gov/2.0/sciUncertainty.owl#Error",
        "label": "Relative Error",
        "uom": { "code": "%" }
    }
]
},
"values": [
    {"temp": 0, "error": 5},
    {"temp": 10, "error": 2},
    {"temp": 50, "error": 2},
    {"temp": 80, "error": 5},
    {"temp": 70, "error": 3}
]
}

```

Listing

B.2.2. Datastream with records (weather data)

This example is based on the same datastream description as the one provided in Annex B.1.2.

The following snippet shows how the datastream records are encoded using the JSON encoding method:

```

[
  {
    "time": "2023-03-20T15:40:00Z",
    "temp": 15.3,
    "press": 1014,
    "windSpeed": 3.5,
    "windDir": 56.0
  },
  {
    "time": "2023-03-20T15:45:00Z",
    "temp": 15.4,
    "press": 1015,
    "windSpeed": 5.6,
    "windDir": 123.0
  },
  {
    "time": "2023-03-20T15:50:00Z",
    "temp": 15.8,
    "press": 1014,
    "windSpeed": 13.2,
    "windDir": 34.0
  },
  ...
]

```

Listing

B.2.3. Datastream with records and optional fields (navigation data)

This example is based on the same datastream description as the one provided in Annex B.1.3.

The following snippet shows how the datastream records are encoded using the JSON encoding method:

```
[
  {
    "time": "2007-10-23T15:46:12Z",
    "speed": 15.3,
    "location": {
      "lat": 45.3,
      "lon": -90.5,
      "alt": 311
    }
  },
  {
    "time": "2007-10-23T15:46:22Z",
    "speed": 25.3,
    "location": null
  },
  {
    "time": "2007-10-23T15:46:32Z",
    "speed": 20.6,
    "location": {
      "lat": 45.3,
      "lon": -90.6,
      "alt": 312
    }
  },
  ...
]
```

Listing

B.2.4. Datastream with records and vector fields encoded as arrays (navigation data)

This example is based on the same datastream description as the one provided in Annex B.1.3.

The following snippet shows how the datastream records are encoded using the JSON encoding method, when `vectorsAsArrays` is true:

```
[
  {
    "time": "2007-10-23T15:46:12Z",
    "speed": 15.3,
    "location": [ 45.3, -90.5, 311 ]
  },
  {
    "time": "2007-10-23T15:46:22Z",
    "speed": 25.3,
    "location": null
  },
  ...
]
```

```

    "time": "2007-10-23T15:46:32Z",
    "speed": 20.6,
    "location": [ 45.3, -90.6, 312 ]
  },
  ...
]

```

Listing

B.2.5. Datastream with choice (navigation data)

This example is based on the same datastream description as the one provided in Annex B.1.4.

The following snippet shows how the datastream records are encoded using the JSON encoding method:

```

[
  {
    "TEMP": {
      "time": "2009-05-23T19:36:15Z",
      "temp": 25.5
    }
  },
  {
    "TEMP": {
      "time": "2009-05-23T19:37:15Z",
      "temp": 25.6
    }
  },
  {
    "WIND": {
      "time": "2009-05-23T19:37:17Z",
      "wind_speed": 56.3,
      "wind_dir": 226.3
    }
  },
  {
    "TEMP": {
      "time": "2009-05-23T19:38:15Z",
      "temp": 25.5
    }
  },
  ...
]

```

Listing

B.2.6. Fixed size 2D array (stress matrix)

This example is based on the same “Matrix” description as the one provided in Annex B.1.5.

The equivalent JSON description for this “Matrix” is provided below:

```

{
  "type": "Matrix",
  "definition": "http://sweet.jpl.nasa.gov/2.0/physPressure.owl#Stress"
  "elementCount": {
    "type": "Count",

```



```

    "value": 3
  },
  "elementType": {
    "name": "row",
    "type": "Matrix",
    "elementCount": {
      "type": "Count",
      "value": 3
    },
  },
  "elementType": {
    "name": "coef",
    "type": "Quantity",
    "definition": "http://sweet.jpl.nasa.gov/2.0/mathVector.owl#Coordinate",
    "uom": { "code": "MPa" }
  }
},
"values": [[0.36,0.48,-0.8], [-0.8,0.6,0.0], [0.48,0.64,0.6]]
}

```

Listing

B.2.7. Datastream of variable size 1D arrays (profile series)

This example is based on the same datastream description as the one provided in Annex B.1.6.

The following snippet shows how the datastream records are encoded using the JSON encoding method:

```

[
  {
    "time": "2005-05-16T21:47:12Z",
    "profilePoints": [
      { "depth": 0, "salinity": 45 },
      { "depth": 10, "salinity": 20 },
      { "depth": 20, "salinity": 30 },
      { "depth": 30, "salinity": 35 },
      { "depth": 40, "salinity": 40 }
    ]
  },
  {
    "time": "2005-05-16T22:43:05Z",
    "profilePoints": [
      { "depth": 0, "salinity": 45 },
      { "depth": 10, "salinity": 20 },
      { "depth": 20, "salinity": 30 },
      { "depth": 30, "salinity": 35 }
    ]
  },
  {
    "time": "2005-05-16T23:40:52Z",
    "profilePoints": [
      { "depth": 0, "salinity": 45 },
      { "depth": 10, "salinity": 20 },
      { "depth": 20, "salinity": 30 },
      { "depth": 30, "salinity": 35 },
      { "depth": 40, "salinity": 40 }
    ]
  },
  ...
]

```

]

Listing

B.2.8. Datastream with geometry (feature detection)

This example is based on the same datastream description as the one provided in Annex B.1.7.

The following snippet shows how the datastream records are encoded using the JSON encoding method:

```
[
  {
    "time": "2007-10-23T15:46:12Z",
    "type": "Car",
    "geom": {
      "type": "Point",
      "coordinates": [-86.3254, 35.4812]
    }
  },
  {
    "time": "2007-10-23T15:49:03Z",
    "type": "Truck",
    "geom": {
      "type": "Polygon",
      "coordinates": [
        [-86.3254 35.4812,-86.3253 35.4812,-86.3253 35.4811,-86.3254 35.4811,-
86.3254 35.4812]
      ]
    }
  },
  {
    "time": "2007-10-23T15:56:45Z",
    "type": "Bus",
    "geom": {
      "type": "Polygon",
      "coordinates": [
        [-86.3254 35.4812,-86.3253 35.4812,-86.3253 35.4811,-86.3254 35.4811,-
86.3254 35.4812]
      ]
    }
  },
  ...
]
```

Listing



ANNEX C (INFORMATIVE) RELATIONSHIP WITH OTHER ISO MODELS



ANNEX C (INFORMATIVE) RELATIONSHIP WITH OTHER ISO MODELS

C.1. Feature model

SWE “Records” can sometimes be seen as feature data from which GML feature representations could be derived. Even if it is true that a SWE “Record” contains values of feature properties, it does not always represent an object like a “Feature” does. The “Record” is simply a logical collection of fields that may be grouped together for a different reason than the fact that they all represent properties of the same object.

The “Feature” model is a higher level model that is used to regroup property values inside the objects that they correspond to, as well as associate a meaning to the object itself.

A good example is a set of weather observations obtained from different sensors that may be grouped into a single “Record” in SWE Common, but which does not constitute a feature in the GIS sense.

C.2. Coverage model

SWE “Arrays” can sometimes be interpreted as coverage range data or grid data. However, SWE data arrays are lower level data types and don’t constitute a “Coverage” in themselves. The “Coverage” model described in OGC Abstract Topic 6 (OGC 07-011) can be used on top of the SWE “Array” model (which only provides means for describing and encoding the data), in order to provide a stronger link between range data and domain definition.

Additionally, sensor descriptions given in SensorML (and thus using the SWE Common model) can be used to define a geo-referencing transformation that can be associated with a coverage via the same model.



ANNEX D (INFORMATIVE) REVISION HISTORY



ANNEX D (INFORMATIVE) REVISION HISTORY

DATE	RELEASE	EDITOR	PRIMARY CLAUSES MODIFIED	DESCRIPTION
2008-08-20	2.0 draft	Alex Robin	All	Initial draft version
2008-10-30	2.0 draft	Ingo Simonis	All	General revision
2009-10-30	2.0 draft	Alex Robin	All	Draft candidate standard
2009-11-04	2.0 draft	Peter Taylor	Clauses 6 and 7	Additional examples, minor edits
2009-11-10	2.0 draft	Alex Robin	All	General revision, added section 8
2010-01-15	2.0 draft	Alex Robin	All	Clarifications in requirements
2010-03-10	2.0 final	Alex Robin	All	Corrections following RFC comments
2023-03-07	2.1 draft	Alex Robin	All	Conversion to AsciiDoc / Metanorma
2023-03-08	2.1 draft	Alex Robin	Clauses 7,8,9	Removed requirements that were redundant with dependencies
2023-03-16	2.1 draft	Alex Robin	Clause 7,8,9	Added Geometry class
2023-03-21	2.1 draft	Alex Robin	Clause 9	Added JSON datastream encoding rules
2023-03-21	2.1 draft	Alex Robin	Clause 9	Clarified use of media types
2024-04-29	3.0 draft	Alex Robin	All	Refactored to v3.0, added JSON encoding, removed XML encoding sections
2024-08-13	3.0 draft	Alex Robin	All	Updated ATS

DATE	RELEASE	EDITOR	PRIMARY CLAUSES MODIFIED	DESCRIPTION
2025-03-18	3.0 draft	Christian Autermann	All	Incorporated feedback from public comments



BIBLIOGRAPHY





BIBLIOGRAPHY

- [1] Katharina Schleidt, Ilkka Rinne: OGC 20-082r4, *Topic 20 – Observations, measurements and samples*. Open Geospatial Consortium (2023). <http://www.opengis.net/doc/as/om/3.0>.
- [2] Mike Botts, Alexandre Robin, Eric Hirschorn: OGC 12-000r2, *OGC SensorML: Model and XML Encoding Standard*. Open Geospatial Consortium (2020). <http://www.opengis.net/doc/IS/SensorML/2.1.0>.
- [3] Alexandre Robin: OGC 08-094r1, *OGC® SWE Common Data Model Encoding Standard*. Open Geospatial Consortium (2011). https://portal.ogc.org/files/?artifact_id=41157.
- [4] Arne Bröring, Christoph Stasch, Johannes Echterhoff: OGC 12-006, *OGC® Sensor Observation Service Interface Standard*. Open Geospatial Consortium (2012). <http://www.opengis.net/doc/IS/SOS/2.0.0>.
- [5] Ingo Simonis, Johannes Echterhoff: OGC 09-000, *OGC® Sensor Planning Service Implementation Standard*. Open Geospatial Consortium (2011). https://portal.ogc.org/files/?artifact_id=38478.