Open Geospatial Consortium Inc.

Date: 2010-02-15

Reference number of this OGC® project document: **OGC 08-094**

Version: 2.0.0

Category: OGC® Standard

Editor: Alexandre Robin

OGC® SWE Common Data Model Encoding Standard

Copyright notice

Warning

This document is not an OGC Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an OGC Standard.

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

Document type: OGC® Publicly Available Standard
Document subtype: Encoding
Document stage: Draft
Document language: English

(Page intentionally left blank)

Contents

Table of Figures

i. Preface

The primary focus of the SWE Common Data Model is to define and package sensor related data in a self-describing and semantically enabled way. The main objective is to obtain 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 sensor web nodes.

This standard is one of several implementation specifications produced under OGC's Sensor Web Enablement (SWE) activity. This is a revision of a first edition which was previously integrated to the SensorML standard specification (OGC 07-000). The SWE Common Data Models are now defined in a separate document that is intended to be used and referenced by other SWE encoding and service standards.

ii. Submitting Organizations

The following organizations submitted this Encoding Standard to the Open Geospatial Consortium Inc. as a Request for Comment (RFC):

- Spot Image, S.A.
- University of Alabama in Huntsville (UAH)
- Commonwealth Scientific and Industrial Research Organisation (CSIRO) Australia
- University of Muenster - Institute for Geoinformatics
- International Geospatial Services Institute GmbH (iGSI)
- 52° North Initiative for Geospatial Open Source Software GmbH
- Southeastern Universities Research Association (SURA)
- Oracle USA
- US Department of Homeland Security (DHS)
- Botts Innovative Research, Inc. (BIRI)

iii. Submission Contact Points

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

Contact	Company	Email
Alexandre Robin	Spot Image, S.A.	alexandre.robin at spotimage.fr
Michael E. Botts	University of Alabama in Huntsville	mike.botts at nsstc.uah.edu
Johannes Echterhoff	iGSI	johannes.echterhoff at igsi.eu
Ingo Simonis	iGSI	ingo.simonis at igsi.eu
Peter Taylor	CSIRO	peter.taylor at csiro.au
Arne Broering	52° North Initiative	broering at 52north.org
Luis Bermudez	SURA	bermudez at sura.org
John Herring	Oracle USA	john.herring at oracle.com
Barry Reff	US DHS	barry.reff at dhs.gov

iv. Revision History

Date	Release	Author	Paragraph modified	Description
08/20/08	2.0 draft	Alexandre Robin	All	Initial draft version
10/30/08	2.0 draft	Ingo Simonis	All	General revision
10/30/09	2.0 draft	Alexandre Robin	All	Draft candidate standard
11/04/09	2.0 draft	Peter Taylor	Clauses 6 and 7	Additional examples, minor edits
11/10/09	2.0 draft	Alexandre Robin	All	General revision, added section 8
01/15/10	2.0 draft	Alexandre Robin	All	Clarifications to requirements

v. Future Work

- More harmonization with GML foreseen
- Development of profiles for commonly used data structures (such as CSML)

vi. Changes to the OGC® Abstract Specification

The OGC® Abstract Specification does not require changes to accommodate this OGC® Standard.

Foreword

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. Open Geospatial Consortium Inc. shall not be held responsible for identifying any or all such patent rights. However, to date, no such rights have been claimed or identified. 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 specification set forth in this document, and to provide supporting documentation.

This document deprecates and replaces clauses 8 "SWE Common Conceptual Models" and 9 "SWE Common XML Encoding and Examples" of the first edition of OGC® Sensor Model Language Specification (OGC 07-000) from which they were extracted. Additionally these clauses have been technically revised and explanations have been improved. These clauses will be removed from version 2.0 of the SensorML standard.

The main changes from version 1.0 (part of SensorML 1.0) are additions of new features such as:

- *The DataChoice component providing support for variant (disjoint union) data type*
- *The DataStream object improving support for real-time (never-ending) streams*
- *The XMLBlock encoding providing support for simple XML encoded data*
- *Support for definition of NIL values and associated reasons*
- *The CategoryRange class to define ranges of ordered categorical quantities*

Additionally, some elements of the language have been removed and replaced by soft-typed equivalent defined using RelaxNG and/or Schematron. The list is given below:

- *Position, SquareMatrix*
- *SimpleDataRecord, ObservableProperty*
- *ConditionalData, ConditionalValue*
- *Curve, NormalizedCurve*

The derivation from GML has also been improved by making all elements substitutable for GML AbstractValue (and thus transitively for GML AbstractObject) so that they can be used directly by GML application schemas. The GML encoding rules as defined in ISO 19136 have also been used to generate XML schemas from the UML models with only minor modifications.

This release is not fully backward compatible with version 1.0 (which was part of the SensorML 1.0 standard) even though changes were kept to a minimum.

SWE Common Data Model: An Implementation Specification

1 Scope

This specification defines low level data models for exchanging sensor related data between nodes of the OGC® Sensor Web Enablement (SWE) framework. These models allow to present datasets in a self describing and semantically enabled way.

More precisely, the SWE Common 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 paragraph §6.

This 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. All categories of sensor observations are in scope ranging from simple in-situ temperature data to satellite imagery and full motion video streamed out of an aircraft.

The SWE Common language is an XML implementation of this model and is used by other existing OGC® Sensor Web Enablement standards such as Sensor Model Language (SensorML), Sensor Observation Service (SOS), Sensor Alert Service (SAS) and Sensor Planning Service (SPS). The Observations and Measurements Standard (O&M) also references the SWE Common data model, although the observation model defined in the O&M specification is decoupled from this standard. One goal of the SWE Common data model is thus to maintain the functionality required by all these related specifications.

2 Conformance

This standard has been 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.

Conformance with this specification 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 ISO 19105: Geographic information — Conformance and Testing. In order to conform to this OGC™ encoding standard, a standardization target shall implement the core conformance class, and choose to implement any one of the other conformance classes (i.e. extensions).

Additionally, it is highly recommended that XML based implementations of this standard implement requirement classes from clause §8 "XML Implementation (normative)" of this standard instead of defining new XML encodings.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of document OGC 08-094. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this document are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies.

- OGC 08-131r3 – The Specification Model – A Standard for Modular Specification
- ISO/IEC 11404:2007 – General-Purpose Datatypes
- ISO/IEC 14977:1996 – Syntactic Metalanguage - Extended BNF
- ISO 8601:2004 – Representation of Dates and Times
- ISO 19103:2005 – Conceptual Schema Language
- ISO 19111:2007 – Spatial Referencing by Coordinates
- ISO 19136:2007 – Geographic Markup Language
- Unified Code for Units of Measure (UCUM) – Version 1.8, July 2009
- Unicode Technical Std #18 – Unicode Regular Expressions, Version 13, Aug. 2009
- The Unicode Standard, Version 5.2, October 2009
- Extensible Markup Language (XML) – Version 1.0 (Fourth Edition), August 2006
- XML Schema – Version 1.0 (Second Edition), October 2004
- IEEE 754:2008 – Standard for Binary Floating-Point Arithmetic
- IETF RFC 2045 – Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies, November 1996

4 Terms and Definitions

For the purpose of this document, the following terms and definitions apply:

Feature (OGC 07-097)

Abstraction of a real world phenomenon perceived in the context of an application. Features may but need not contain geospatial properties. In this general sense, a feature corresponds to an "object" in analysis and design models.

Measurement System (or Procedure)

System used to estimate values of feature properties. A measurement system is usually composed of at least one sensor that initially transforms a real world phenomenon into digital information after which further processing can be done. A procedure is a more general concept that also encapsulates the method that is used to collect particular measurements (see OGC 07-022).

Observation (OGC 07-022)

Act of observing a property or phenomenon, with the goal of producing an estimate of the value of the property.

Property

Concept that is a characteristic of one or more feature types, the value for which may be estimated by application of some procedure in an observation.

Sensor

Entity that provides information about an observed property at its output. 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 produce signals to be processed.

Sensor Network

A collection of sensors and processing nodes, in which information on properties observed by the sensors may be transferred and processed. Note: A particular type of a sensor network is an ad hoc sensor network.

Sensor Data

List of digital values produced by a measurement system that represent estimated values of one or more measured properties of one or more features. Sensor data is usually available in the form of data streams or computer files.

Sensor Related Data

List of numerical values produced by a measurement system that contains auxiliary information that is not directly related to the value of measured properties. Examples of auxiliary data values are sensor status, quality of measure, quality of service, etc… When such data is measured, it is sometimes considered sensor data as well.

Data Component

Element of sensor data definition corresponding to an atomic or aggregate data type. A data component is a part of the overall dataset definition which can be seen as a hierarchical tree of data components.

5 Conventions

5.1 Abbreviated terms

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

API	Application Program Interface
GPS	Global Positioning System
ISO	International Organization for Standardization
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	eXtended Markup Language
1D	One Dimensional
2D	Two Dimensional
3D	Three Dimensional

5.2 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.

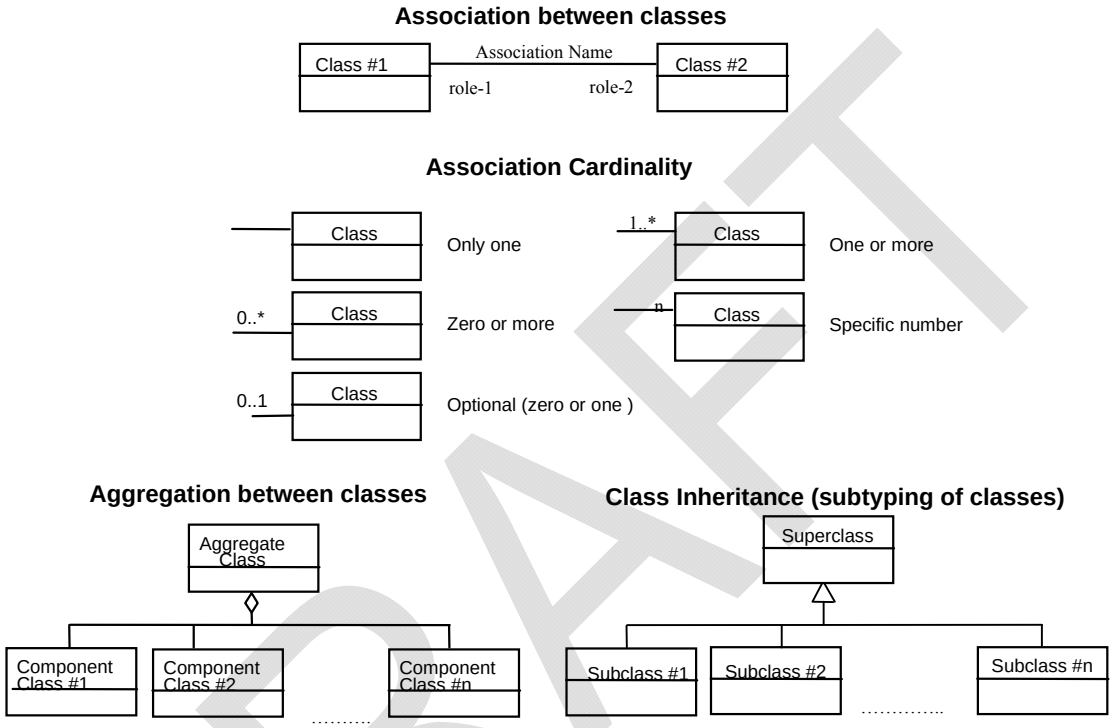


Figure 5.1 – UML Notation

5.3 Finding requirements and recommendations

For clarity, each normative statement in this standard is in one and only one place and is set in a **bold font**. If the statement of the requirement or recommendation is repeated for clarification, the "bold font" home of the statement is considered the official statement of the normative requirement or recommendation. In this sense, all requirements in this standard are listed in the Table of Requirements at the beginning of this standard.

In this standard, all requirements are associated to tests in the abstract test suite in Annex A. The reference to the requirement in the test case is done by a requirements label (in the form "**Req #**", where "#" is a number) associated to the "bold font" home of the statement described above. Recommendations are not tested and are not labeled, although they still use a bold font for their unique home statement.

Requirements classes are separated into their own clauses and named, and specified according to inheritance (direct dependencies). The Conformance test classes in the test suite are similarly named to establish an explicit and mnemonic link between requirements classes and conformance test classes.

6 Requirements Class: Core Concepts (normative core)

6.1 Introduction

The generic SWE Common data model defined by this standard aims at providing verbose information to robustly describe sensor related datasets. We define **Sensor Data** as data resulting from the observation of properties of virtual or real world objects (or features) by any type of **Measurement System** (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, we 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 concepts defined in this section shall be correctly implemented by all models or software seeking compliance with this standard.

Req 1 **A conformant model or software shall implement the concepts defined in the core of this standard in a way that is consistent with their definition.**

6.2 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.

6.2.1 Boolean

A Boolean representation of a property can take only two values which 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.

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 in on, FALSE if the system is off.

Req 2 A boolean representation shall at least consist of a boolean value.

The "Boolean" data type detailed in clause §7.2.5 is used to define a data component with a Boolean representation.

6.2.2 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.

Examples

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

A camera mode can be represented by a categorical value – AUTO_FOCUS, MANUAL_FOCUS, etc…

Req 3 A categorical representation shall at least consist of a category identifier and information describing the value space of this identifier.

The "Category" data type detailed in clause §7.2.7 is used to define a data component with a categorical representation.

6.2.3 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.).

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.

Req 4 **A continuous numerical representation shall at least consist of a decimal number and the scale (or unit) used to express this number.**

The "Quantity" data type detailed in clause §7.2.9 is used to define a data component with a decimal representation and a unit of measure.

6.2.4 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.

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.

Req 5 A countable representation shall at least consist of an integer number.

The "Count" data type detailed in clause §7.2.8 is used to define a data component with an integer representation and no unit of measure.

6.2.5 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.

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).

Req 6 A textual representation shall at least consist of a character string.

The "Text" data type detailed in clause §7.2.6 is used to define a data component with a textual representation.

6.2.6 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.

Examples

A decimal representation of an angular property such as latitude can be constrained to the [-90° 90°] interval.

A temperature reading produced by a sensor can be constrained to the [-50°C +250°C] range.

6.3 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 §6.4.2.

6.3.1 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.

6.3.2 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.

Req 7 **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 XML schemas desribed 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.

Req 8 **If robust semantics are provided by referencing out-of-band information, the locators or identifiers used to point to this information shall be resolvable by some well-defined method.**

6.3.3 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.

Examples

Satellite location 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).

Req 9 **A temporal quantity shall be expressed with respect to a well defined temporal reference frame and this frame shall be specified.**

Req 10 **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 detailed in clause §7.2.10 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 §7.3.3 is a special type of record used to assign a reference frame to all its child-components.

The "*Matrix*" class defined in clause §7.4.2 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.

6.4 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.

6.4.1 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.

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.

6.4.2 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.

Req 11 **A model of a NIL value shall always include a mapping between the selected reserved value and a well-defined reason.**

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

Examples

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'

- A reserved value to indicate that a pixel value was "Above Detection Limit" usually set to '255'

6.4.3 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 and Measurements Standard (O&M). In this standard, the "*procedure*" property of the "*Observation*" class allows attaching detailed information about the measurement procedure, that is to say a description of how the data was obtained (i.e. lineage), to the data itself.

6.5 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.

Req 12 A conformant model or software shall implement aggregate data structures 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 6.2.

Examples

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

- XML 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.

- 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.
- A data component used as a data container equally defines the dataset but also includes the actual property values.

6.6 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 XML, 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 encodings 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.

Req 13 **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 6.5.**

7 UML Conceptual Models (normative)

This standard defines normative UML models with which all future separate extensions should be compliant. The standardization target types for the UML conformance classes are:

- Software implementations seeking compliance to this standard
- Encoding models derived from the conceptual models of this standard

7.1 Package Dependencies

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

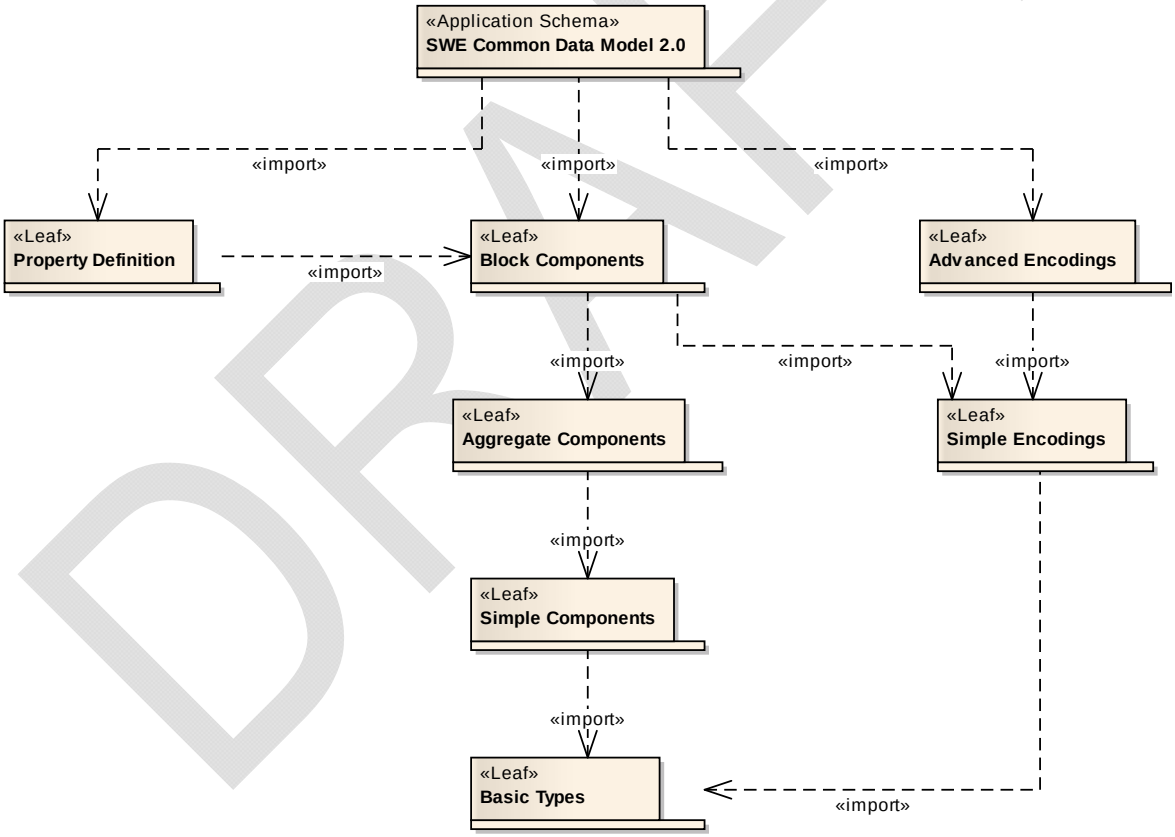


Figure 7.1 – Internal Package Dependencies

This standard also has dependencies on external packages defined by other standards, namely ISO 19103 and ISO 19136, as show below:

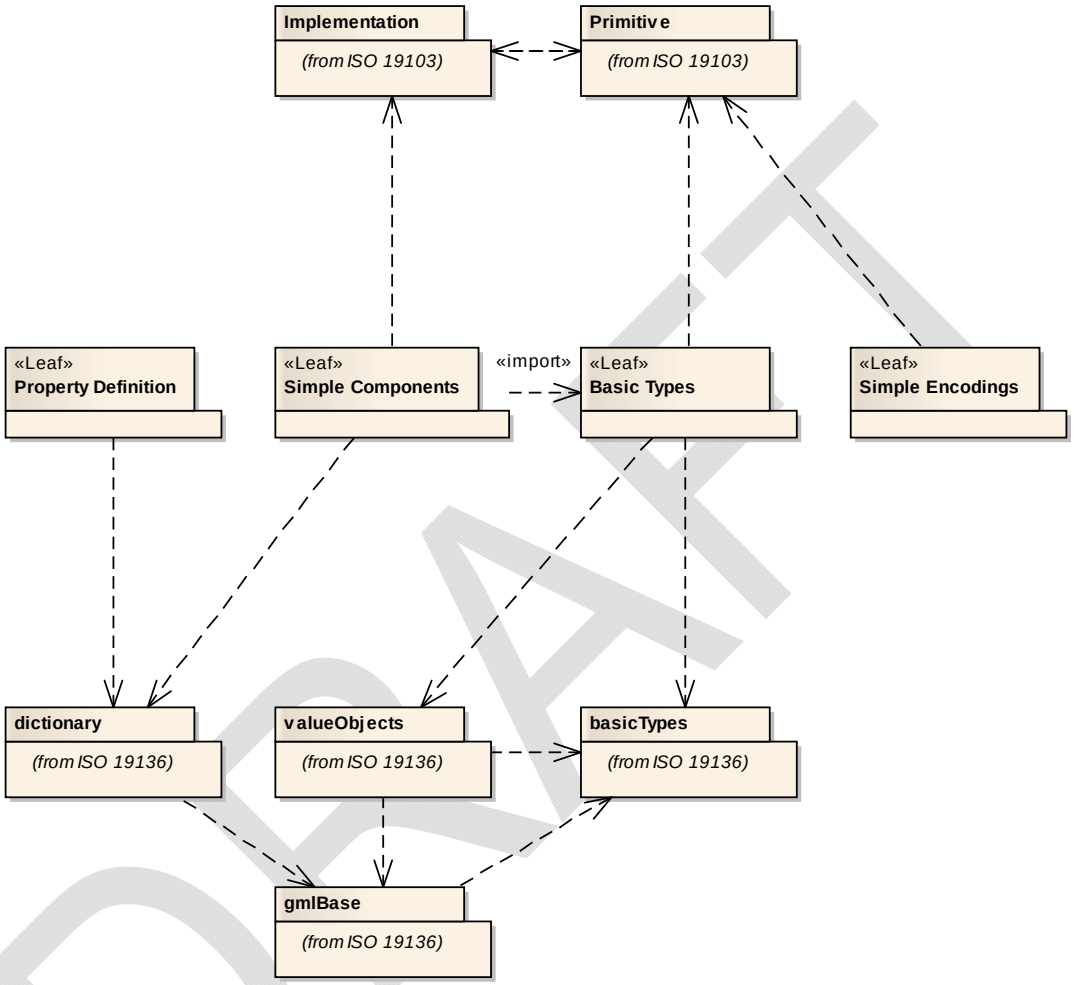


Figure 7.2 – External Package Dependencies

7.2 Requirements Class: Basic Types and Simple Components Packages

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.

Req 14 An implementation passing the "Simple Components UML Package" conformance test class shall first pass the core conformance test class.

The "Basic Types" UML package from which the "Simple Components" package is dependent is included in this requirement class.

Req 15 A compliant encoding or software shall correctly implement all UML classes defined in the "Simple Components" and "Basic Types" packages.

Data types from the "Primitive" and "Implementation" 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*", "*GenericName*", "*ScopedName*".

Req 16 A compliant encoding or software shall correctly implement all UML classes defined in ISO 19103 that are used in this standard.

Classifiers (i.e. classes and data types) from the "basicTypes", "valueObjects" and "dictionary" packages of the ISO 19136 (GML) standard are also used. These packages are thus dependencies of this requirement class. The GML classifiers used are "*AbstractValue*", "*URI*", "*Definition*".

Req 17 A compliant encoding or software shall correctly implement all UML classes defined in ISO 19136 (GML) that are used in this standard.

Classes of the "Simple Components" package are designed to collect information about nature, representation and quality of data as introduced in previous sections. These include six scalar types – *Boolean*, *Text, Category*, *Count, Quantity,* and *Time* – as well as four range types – *CategoryRange, CountRange, QuantityRange* and *TimeRange*.

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

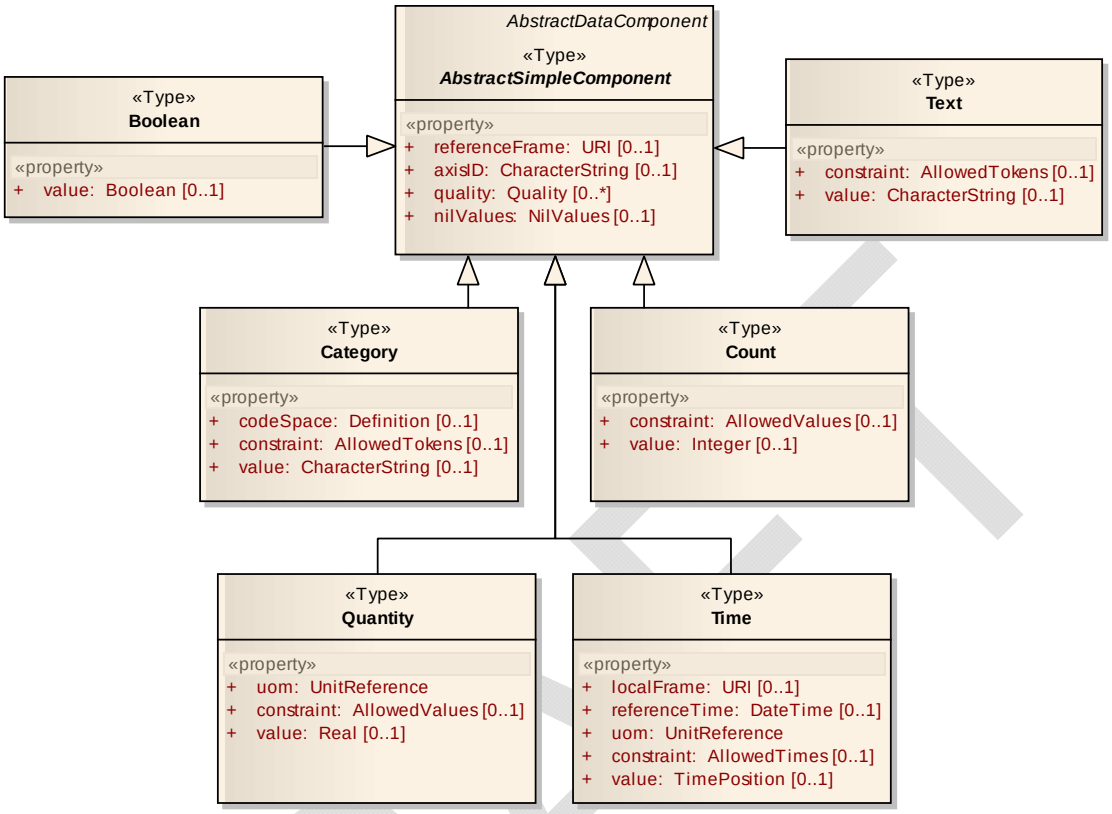


Figure 7.3 – Simple Data Components

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

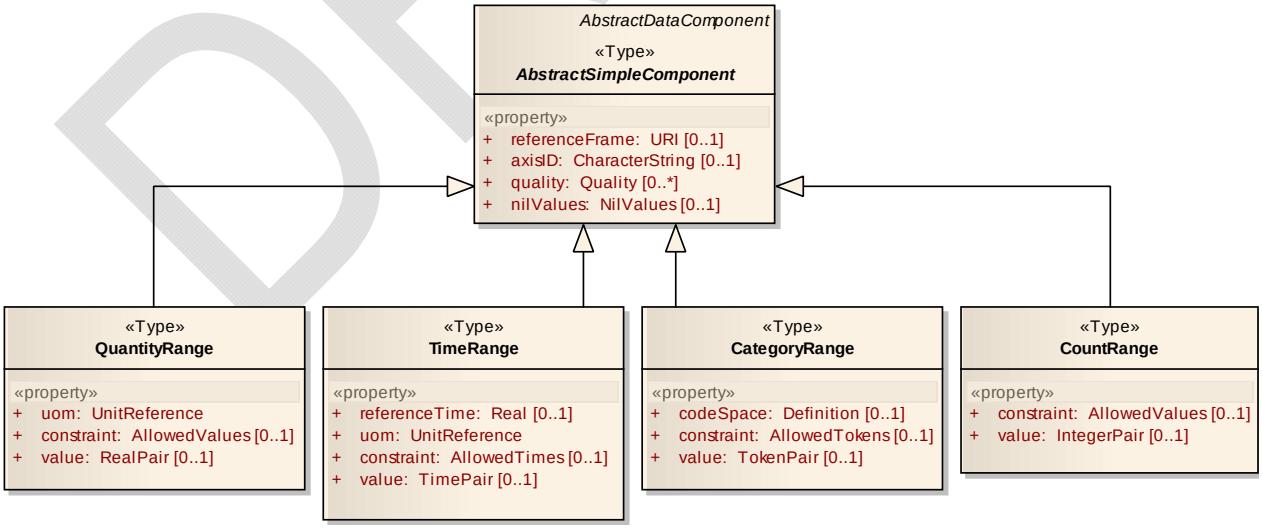


Figure 7.4 – Range Data Components

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

7.2.1 Relationship with GML Value Objects

SWE Common data components are enhanced versions of GML value objects. They are used to associate robust metadata information described in clauses §6.2 to §6.4 to the actual property value. As with GML value objects, this can be done in two ways:

- Scalar data components can contain the property value inline. In order to achieve this, the "*value*" attribute of the data component object is filled with the property value. In GML, value objects take this value directly as text content.
- Data components can be used as descriptors for a data structure which values are given separately (i.e. for example encoded in Text, Binary or XML in the "*values*" attribute of a "*DataArray*" object). This is similar to the way GML value objects are used to specify range parameters of the "*DataBlock*" class.

All SWE Common classes representing data components are sub-classes of the GML "*AbstractValue*" class (see Figure 7.7), which enables the use of these classes within other GML application schemas. They are especially intended to be used for describing range parameters in the GML coverage sub-schema.

7.2.2 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.

The first derived data type defined in this package allows expressing a temporal position value as shown on the diagram below:

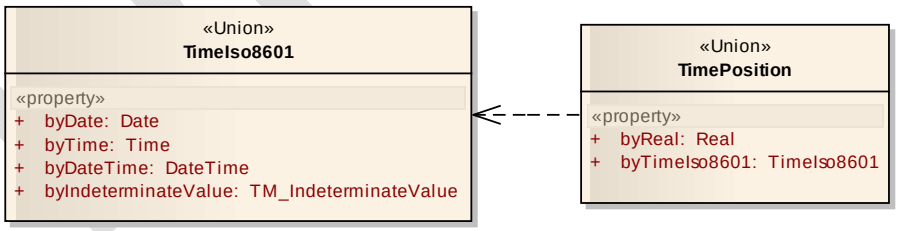


Figure 7.5 – TimePosition Data Type

By using this data type, a time position can be expressed in five different ways:

- A simple calendar date split into year, month and day of month.
- A time with a time zone, split into hours, minutes and seconds.

- A combination of date and time with a time zone, split into year, month, day, hours, minutes and seconds.
- A decimal value expressed in a temporal unit such as seconds, milliseconds, etc.
- An indeterminate value used to specify special temporal values such as 'now'.

All these time position values are expressed relative to an epoch that is either implicitly or explicitly defined elsewhere (i.e. by a separate property than the one carrying the time position value).

The other data types defined in this package all relate to defining pairs of data types defined in ISO 19103:

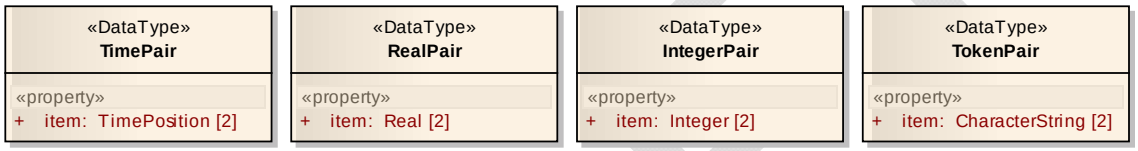


Figure 7.6 – Basic types for pairs of scalar types

7.2.3 Attributes shared by all data components

All SWE Common data component classes carry standard attributes inherited (transitively) from the "*AbstractDataComponent*" and "*AbstractSWEValue*" classes (The "*AbstractSWEValue*" 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:

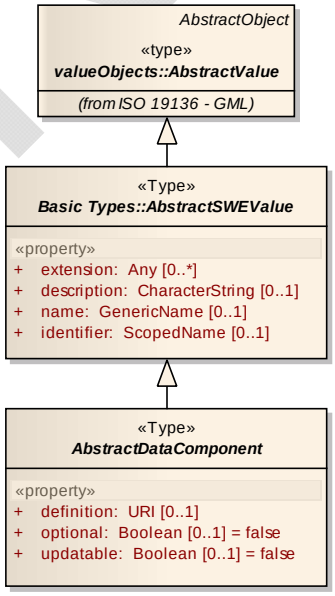


Figure 7.7 – AbstractDataComponent Class

The "*extension*" attribute is used as a container for future extensions. Each extension should put its content in a separate extension property. It is available by inheritance to all sub-classes of "*AbstractSWEValue*". This extension point can be used at runtime (i.e. at the instance level in the case of XML encoding) to add new extended properties to an existing class.

Req 18 A compliant implementation shall not generate errors when the content of an "extension" attribute is unknown.

The optional "*name*" and "*description*" attributes can be used to provide human readable information describing what property the component represents. The "*name*" 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 0). Instead, the "*definition*" attribute described below should be used for that purpose.

The optional "*identifier*" 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 a universal constant.

The "*definition*" attribute provides a resolvable reference (generally a URI but usually a URL or a URN) to the component semantics. It should point to controlled terms that are defined in online dictionaries, registries or ontologies. These 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.

Examples

The definition may indicate that the value represents an atmospheric temperature using a URN such as "urn:ogc:def:property:OGC:AtmosphericTemperature" 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.my-org.com/ /ontologies/observedProperties/atmospheric.owl#temperature"

The name could be "Atmospheric Temperature", which allows quick identification by human data consumers.

The description could be "Temperature of the atmosphere measured by the exterior thermometer" which adds contextual details.

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). It is '*false*' by default.

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.

- 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.

7.2.4 Attributes shared by all simple data components

As shown on Figure 7.3, classes modeling simple data components inherit attributes from the "*AbstractSimpleComponent*" class from which they are directly derived. This abstract class is shown again below:

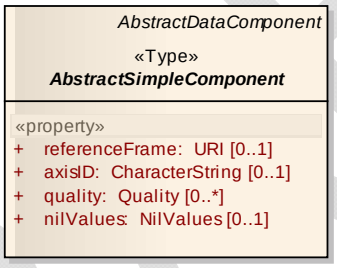


Figure 7.8 – AbstractSimpleComponent Class

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

Req 19 The "definition" attribute shall be specified by all instances of concrete classes derived from "AbstractSimpleComponent".

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 §6.3.3. These attributes are used for a component which value is the projection of a property along a temporal or spatial axis.

The "*referenceFrame*" attribute takes a URI that uniquely identifies the reference frame relative to which the coordinate value is given.

Req 20 The URI used as the value of the "referenceFrame" attribute shall identify a coordinate reference system as defined by ISO 19111.

The "*axisID*" attribute takes a string that uniquely identifies one of the reference frame's axes along which the coordinate value is given.

Req 21 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.

A component representing a projected quantity can be defined in isolation or can be contained within a "*Vector*" aggregate when it contributes to the specification of a multi-dimensional quantity (see clause §7.3.3). In this last case the reference frame definition is usually inherited from the parent "*Vector*" instance and is thus omitted from the scalar component itself. However, the "*axisID*" attribute still needs to be specified.

Req 22 The "axisID" attribute shall be specified by all instances of concrete classes derived from "AbstractSimpleComponent" and representing a property projected along a spatial axis.

Req 23 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).

The optional "*quality*" attribute is used to provide simple quality information as discussed in §6.4.1. It is of type "*Quality*" which is a union of several classes as defined in clause §7.2.16. Its multiplicity is more than one which means that several quality measures can be given on for a single data component.

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).

The optional "*nilValues*" attribute is used to provide a list (i.e. one or more) of NIL values as defined in clause §6.4.2. The model of the "*NilValues*" class is detailed in clause §7.2.17.

Although this is not shown on Figure 7.8, most concrete sub-classes of "*AbstractSimpleComponent*" also define a "*constraint*" attribute that allows further restriction of the possible values allowed by the corresponding representation. This implements concepts defined in clause §6.2.6. 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).

Req 24 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).

Req 25 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 out-of-band. 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.

7.2.5 Boolean Class

The "*Boolean*" class is used to specify a scalar data component with a Boolean representation as defined in clause §6.2.1. It derives from "*AbstractSimpleComponent*" and is shown below:

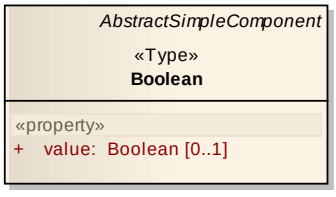


Figure 7.9 – Boolean Class

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.

7.2.6 Text Class

The "*Text*" class is used to specify a component with a textual representation as defined in clause §6.2.5. It derives from "*AbstractSimpleComponent*" and is shown below:

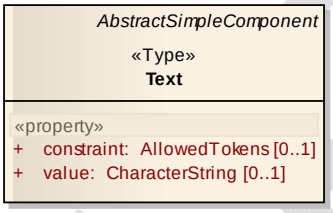


Figure 7.10 – Text Class

The "*constraint*" attribute allows further restricting the range of possible values by using the "*AllowedTokens*" class defined in clause §7.2.18. 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.

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 an XML document or in a SWE Common data stream.

7.2.7 Category Class

The "*Category*" class is used to specify a scalar data component with a categorical representation as defined in clause §6.2.2. It derives from "*AbstractSimpleComponent*" and is shown below:

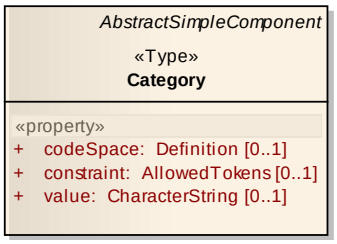


Figure 7.11 – Category Class

The "*codeSpace*" attribute is of type "*Definition*" and allows listing and defining the meaning of all possible values for this component. It is expected that instances of the "*Definition*" class will usually be referenced by implementations of this class since the code space definition is usually obtained from a remote controlled vocabulary. This type of implementation is indeed used in the XML encodings defined by this standard.

The "*constraint*" attribute allows further restricting the list of possible values by using the "*AllowedTokens*" class defined in clause §7.2.18. This is usually done by specifying a limited list of possible values, which have to be extracted from the code space.

Req 26 **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.

Req 27 **An instance of the "Category" class shall either specify a code space or an enumerated list of allowed tokens, or both.**

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.

Req 28 **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.**

7.2.8 Count Class

The "*Count*" class is used to specify a scalar data component with a discrete countable representation as defined in clause §6.2.4. It derives from "*AbstractSimpleComponent*" and is shown below:

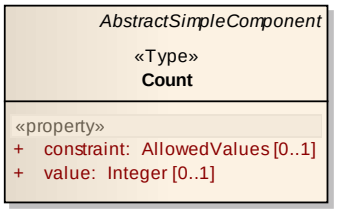


Figure 7.12 – Count Class

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 §7.2.19. 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.

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.

7.2.9 Quantity Class

The "*Quantity*" class is used to specify a component with a continuous numerical representation as defined in clause §6.2.3. It derives from "*AbstractSimpleComponent*" and is shown below:

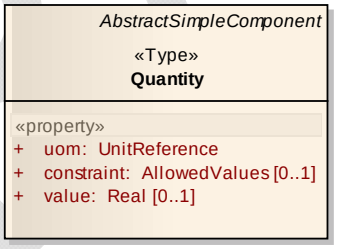


Figure 7.13 – Quantity Class

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.

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 §7.2.19. 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.

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.

7.2.10 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:

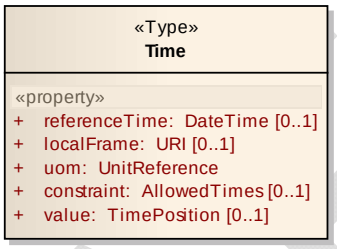


Figure 7.14 – Time Class

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.

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.

Req 29 The "referenceFrame" attribute inherited from "AbstractSimpleComponent" shall be set on all instances of the "Time" class.

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

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 constants 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.

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.

Req 30 The value of the "referenceTime" attribute shall be expressed with respect to the system of reference indicated by the "referenceFrame" attribute.

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.

- 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

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.

Req 31 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

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).

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.

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.

Req 32 The "uom" attribute of an instance of the "Time" class shall specify a base or derived time unit.

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 §7.2.20.

The "*value*" attribute (or the corresponding value in out-of-band data) is of type "*TimePosition*" (see clause §7.2.2) and must match the constraint.

7.2.11 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.

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).

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.

Req 33 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.

7.2.12 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:

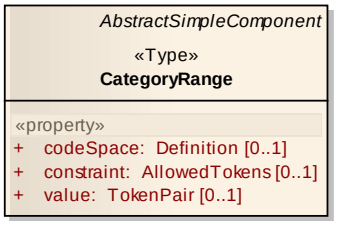


Figure 7.15 – CategoryRange Class

Req 34 All requirements associated to the "Category" class defined in clause §7.2.7 apply to the "CategoryRange" class.

The "*CategoryRange*" class also requires that the underlying code space is ordered so that the range is meaningful.

Req 35 The code space specified by the "codeSpace" attribute of an instance of the "CategoryRange" class shall define a well-ordered set of categories.

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].

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

7.2.13 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:

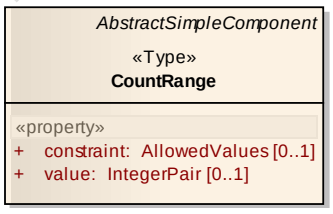


Figure 7.16 – CountRange Class

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

7.2.14 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:

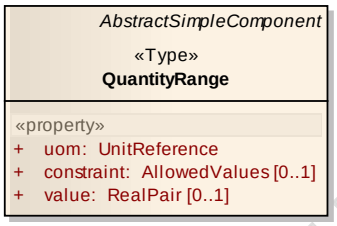


Figure 7.17 – QuantityRange Class

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

7.2.15 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:

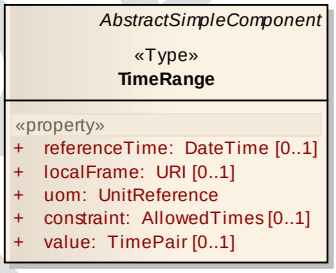


Figure 7.18 – TimeRange Class

Req 36 All requirements associated to the "Time" class defined in clause §7.2.10 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.

7.2.16 Quality Union

The "*Quality*" class is a union allowing the use of different representations of quality.

Quality can be 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:

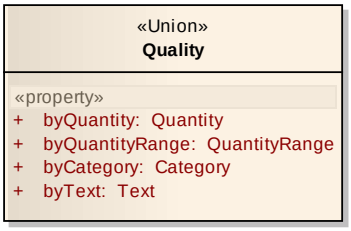


Figure 7.19 – Quality Union

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

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. See the XML implementation clause for more information on this use case.

7.2.17 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 §6.4.2). The UML model of this class is given below:

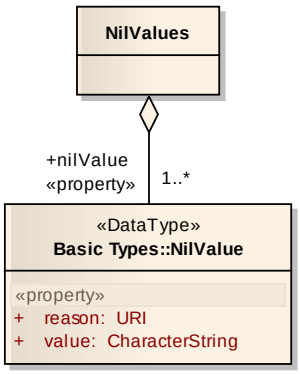


Figure 7.20 – NilValues Class

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).

Req 37 The "reason" attribute of an instance of the "NilValue" class shall contain a URI that can be resolved 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.

Req 38 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.

7.2.18 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:

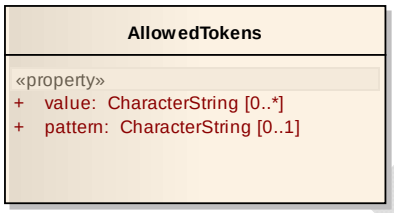


Figure 7.21 – AllowedTokens Class

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.

7.2.19 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:

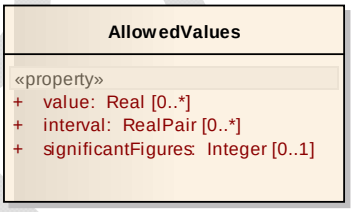


Figure 7.22 – AllowedValues Class

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.

Req 39 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.

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.

7.2.20 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:

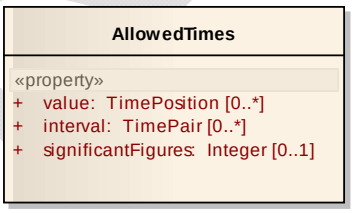


Figure 7.23 – AllowedTimes Class

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 §7.2.2.

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

7.2.21 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:

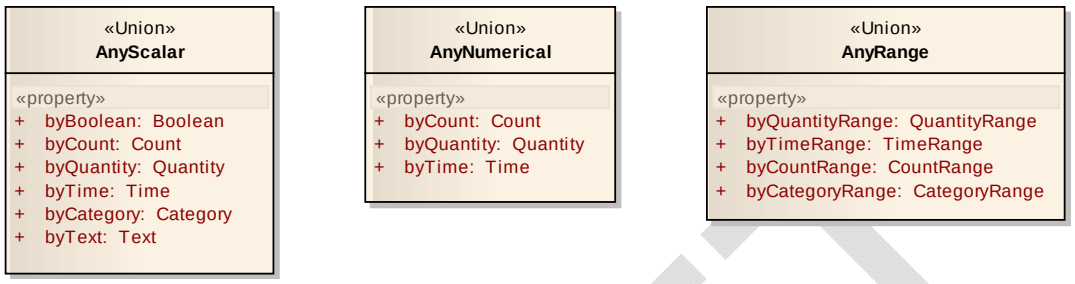


Figure 7.24 – Simple component unions

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 regroups all range components.

7.3 Requirements Class: Aggregate Components Package

As detailed in the following clauses, this package defines classes modeling aggregate component types that can be nested to build complex structures from the simple component types introduced in §7.2.

Req 40 **An implementation passing the "Aggregate Components UML Package" conformance test class shall first pass the "Basic Types and Simple Components UML Packages" conformance test class.**

Req 41 **A compliant encoding or software shall correctly implement all UML classes defined in the "Aggregate Components" package.**

Simple component types can be wrapped by aggregates in order to be inserted in a larger structure. The classes modeling aggregate components defined in this package are "*DataRecord*", "*DataChoice*" and "*Vector*" (other aggregates are defines in the "Block Components" package defined in clause §7.4). The UML model is exposed below:

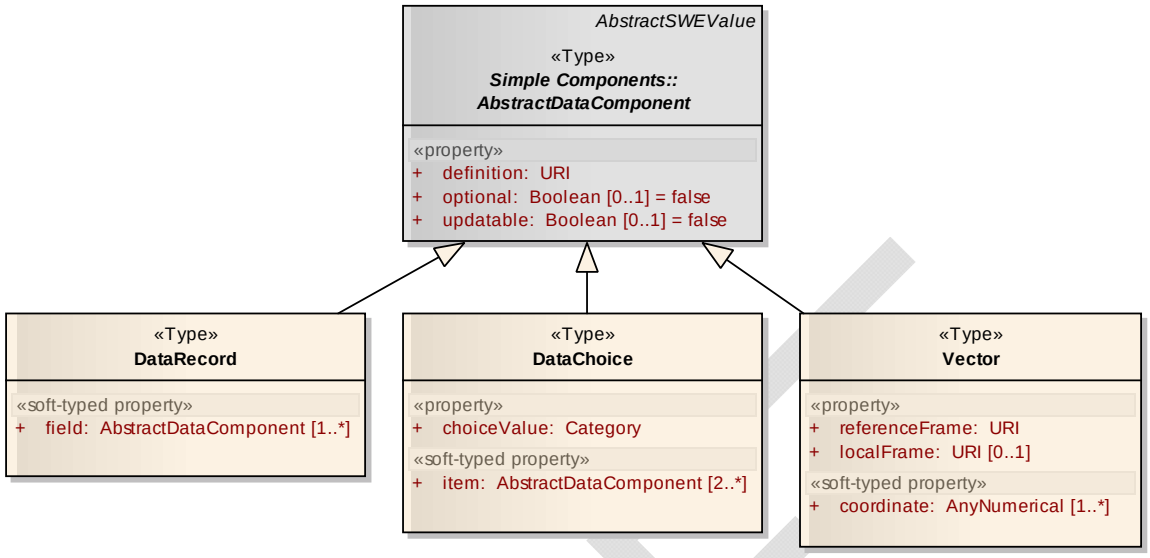


Figure 7.25 – Aggregate Data Components

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

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.

This requirements class has a parameter that can take one of two possible values: "simple nesting" or "unlimited nesting". When "simple nesting" is selected, only DataRecords, Vectors and data components defined in the Simple Components package can be nested within other aggregates. When "full nesting" is selected, agregates can be nested within each other in any order.

7.3.1 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 aggregate component types.

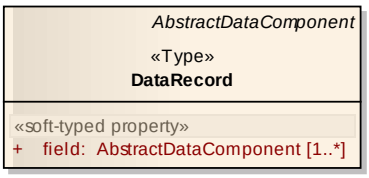


Figure 7.26 – DataRecord Class

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.

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. Its stereotype is "*soft-typed property*" which means that the property will be fully identified with a name provided by implementations that realize that class. This name 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 fields.

Req 42 Each "field" attribute in a given instance of the "DataRecord" class shall be identified by a name that is unique to this instance.

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.

7.3.2 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:

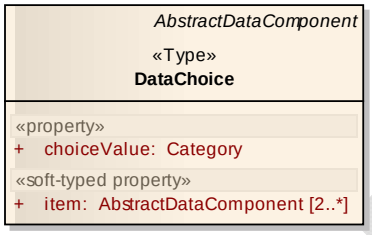


Figure 7.27 – DataChoice Class

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).

Each "*item*" attribute can thus take an instance of any concrete sub-class of "*AbstractDataComponent*", which is the superset of all data component types defined in this standard. Its stereotype is "*soft-typed property*" which means that the property will be fully identified with a name provided by implementations that realize that class. This name must 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 items.

Req 43 **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.

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.

7.3.3 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:

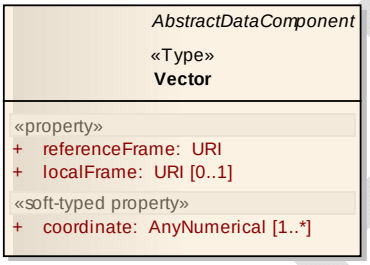


Figure 7.28 – Vector Class

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.

Req 44 The "referenceFrame" attribute shall be ommited from all data components used to define coordinates of a "Vector" instance.

Req 45 The "axisID" attribute shall be specified on all data components used to define coordinates of a "Vector" instance.

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).

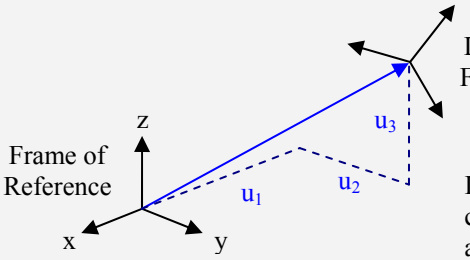
Req 46 The "localFrame" attribute of an instance of the "Vector" class shall have a different value than the "referenceFrame" attribute.

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 remote sensor data.

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

Example #1

A 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".



In the case of a 3D location vector, each coordinate u_1, u_2, u_3 represents a distance along the x, y, z axes respectively.

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.

7.4 Requirements Class: Block Components Package

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.

Req 47 **An implementation passing the "Block Components UML Package" conformance test class shall first pass the "Aggregate Components UML Package" and "Simple Encodings UML Package" conformance test classes.**

Req 48 **A compliant encoding or software shall correctly implement all UML classes defined in the "Block Components" package.**

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

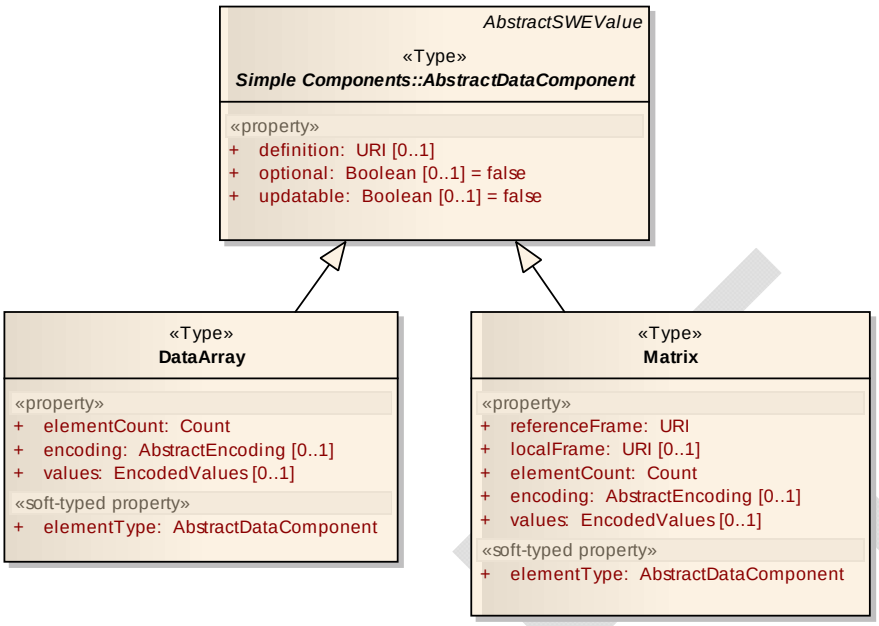


Figure 7.29 – Array Components

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 is not a data component.

7.4.1 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:

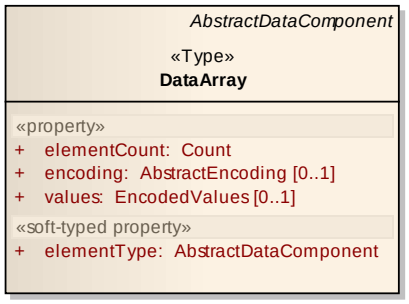


Figure 7.30 – DataArray Class

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.

The "*elementCount*" attribute is used to indicate the size of the array (i.e. the number of elements of the given type in the array).

The content of the "*elementType*" attribute defines the 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*".

Req 49 **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.

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 §0 and §7.6. The "*encoding*" field shall have a value if the "*values*" field is present, and the data shall be encoded using the specified encoding.

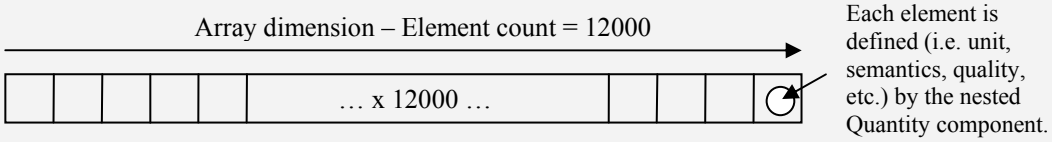
Req 50 **Whenever an instance of a block component contains values, an encoding method shall be specified by the "encoding" property and array values shall be encoded as specified by this method.**

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

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.

Example

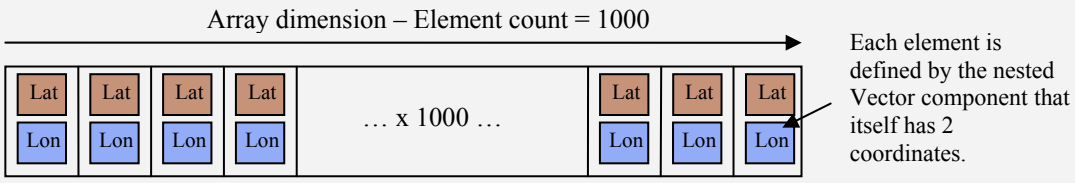
The "*DataArray*" class can be used to describe a simple 1D array of measurements such as radiance values obtained using a 12000 cells (1 row) CCD strip for instance. This can be done by using the "*Quantity*" class as the element type. In such a case, describing the dataset as a "*DataRecord*" would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



The possibility of nesting a "*DataRecord*" or "*Vector*" inside a "*DataArray*" allows the construction of arrays of more complex structures, useful to describe trajectories, profiles, images, etc.

Example

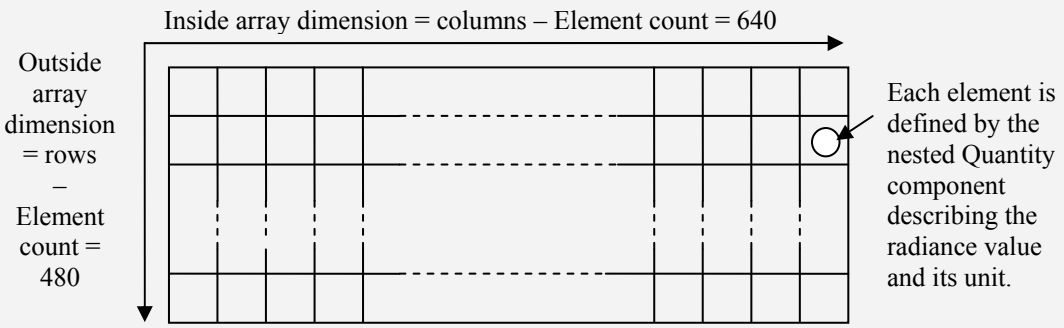
The "*DataArray*" class can be used as a descriptor for a trajectory dataset by using a vector of [latitude, longitude] coordinates as its element type. Note that this can also be considered as a 1D coverage in a 2D CRS.



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

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).

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.

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 XML implementation section.

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.

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

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.

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.

7.4.2 Matrix Class

The "*Matrix*" class is essentially the same as the "*DataArray*" class except that it provides 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:

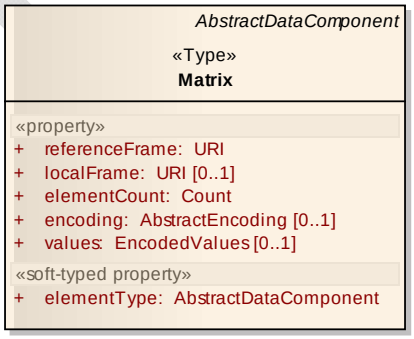


Figure 7.31 – Matrix Class

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.

The "*elementCount*", "*elementType*", "*encoding*" and "*values*" attributes are equivalent to the ones in the "*DataArray*" class.

Req 49 and Req 50 also apply to the "Matrix" class.

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.

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.

Examples

The "*Matrix*" class can be used to represent for instance:

- A 3D 3x3 stress tensor

- A 4D 4x4 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.

7.4.3 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:

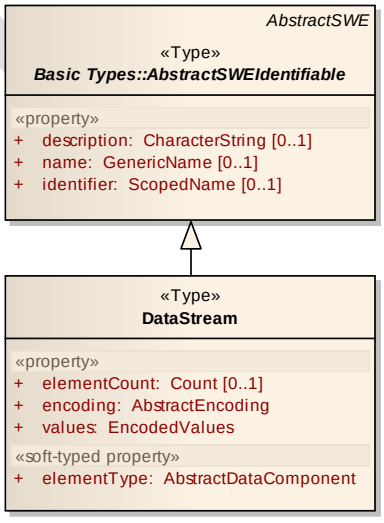


Figure 7.32 – DataStream Class

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.

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.

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*".

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 §7.5 and §7.6.

Req 49 also applies to the "DataStream" class.

7.5 Requirements Class: Simple Encodings Package

Encoding methods describe how structured array and stream data is encoded into a low level byte stream (see related concepts in clause §6.6). 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.

Req 51 An implementation passing the "Simple Encodings UML Package" conformance test class shall first pass "Basic Types and Simple Components UML Package" conformance test class.

Req 52 A compliant encoding or software shall correctly implement all UML classes defined in the "Simple Encodings" package.

This package defines two encoding methods that can be used in conjunction with a classes defined in the "*Block Components*" package. There model is shown on the diagram below:

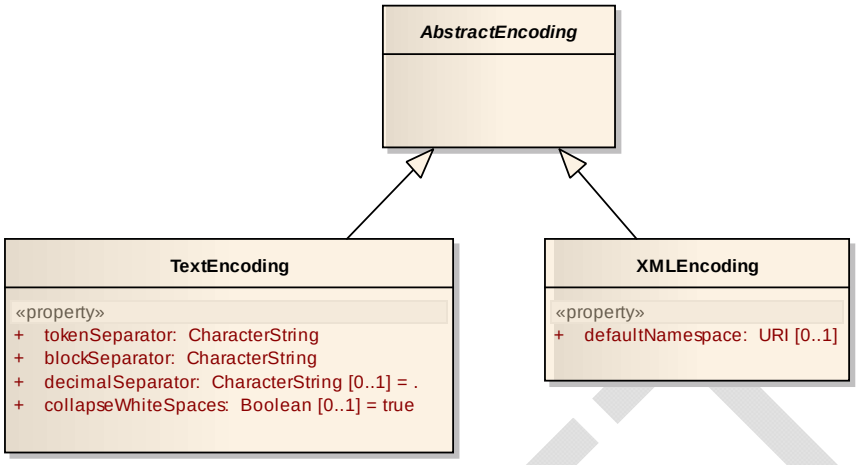


Figure 7.33 – Simple Encodings

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.

7.5.1 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:

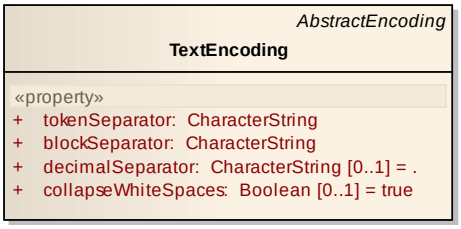


Figure 7.34 – TextEncoding Class

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 clause §8.4.

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

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 ',' 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 is useful for encoded blocks of data that are embedded in an XML document, since formatting (indenting with spaces or tabs especially) is often done in XML content.

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.

7.5.2 XMLEncoding Class

The "*XMLEncoding*" class defines a method that allows encoding structured data into a stream of nested XML tags, which names are obtained from the data structure definition. The class is shown below in UML:

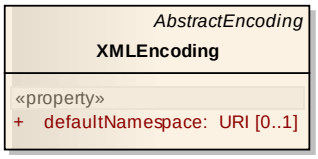


Figure 7.35 – XMLEncoding Class

The only attribute of this class, "*defaultNamespace*", is used to specify what namespace should be used for the tags used in the data stream.

This encoding method is used when compactness is not an issue and a high level of readability and error detection is required. Furthermore, a data stream encoded in this way can be easily transformed and formatted using XSLT like languages.

The main drawback of this method is the verbosity and high degree of redundancy of the information contained in the stream due to the repetitive XML tags. This problem can be minimized by compressing the XML data using well known techniques such as GZIP or BZIP, but such an approach may not be viable in the case of streaming (e.g. real time) data.

7.6 Requirements Class: Advanced Encodings Package

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.

Req 53 **An implementation passing the "Advanced Encodings UML Package" conformance test class shall first pass the "Simple Encodings UML Package" conformance test class.**

Req 54 **A compliant encoding or software shall correctly implement all UML classes defined in the "Advanced Encodings" package.**

7.6.1 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:

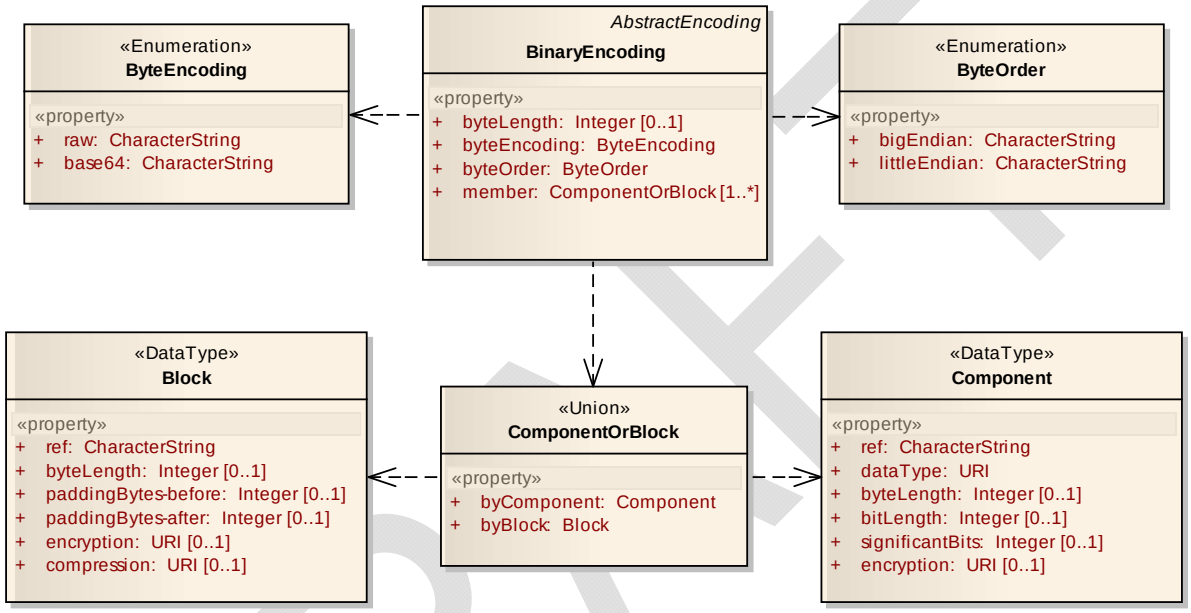


Figure 7.36 – BinaryEncoding Class

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 an 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.

8 XML Implementation (normative)

This standard defines normative XML schemas and ISO Schematron patterns with which all future separate extensions should be compliant. The standardization target types for all XML related conformance classes are:

- XML instances compliant to this standard
- Software reading, writing or processing these XML instances
- XML schemas that have a dependency on the SWE Common Data Model schemas

XML schemas defined in this section are a compliant implementation of the UML conceptual models described in clause §7. They are auto-generated from these models by strictly following well-defined encoding rules. All attributes and composition/aggregation associations contained in the UML models are encoded either as XML elements or XML attributes but the names are fully consistent. One XML schema file is produced for each UML package.

Schematron patterns implement most additional requirements stated in clause §7. One Schematron file is produced for each UML package.

All example instances given in this section are informative and are used solely for illustrating features of the normative model. They are marked by a light gray background.

8.1 Requirements Class: XML Encoding Principles

This section lists common requirements associated to the XML encoding rules used in the context of this standard.

Req 55 An implementation passing the "XML Encoding Principles" conformance test class shall first pass the core conformance test classes.

As mentioned above, the normative XML schemas in this standard have been generated by strictly following UML to XML encoding rules, such that the schemas are the exact image of the UML models. The same encoding principles shall be used by all extensions of this standard.

8.1.1 XML Encoding Conventions

The rules used to encode the SWE Common data models into an XML Schema are similar to those used to derive GML application schemas and defined in ISO 19136. Extensions to these rules have been defined to allow:

- Use of "soft-typed" properties. These properties are encoded as XML elements with a generic element name but provide a "*name*" attribute for further desambiguation.
- Encoding of certain properties as XML attributes. This type of encoding adds to the "element-only" rules defined by ISO 19136. It is restricted to properties with a primitive type and indicated by a new tagged value in the UML model.
- Use of a new abstract base type. A custom base type called "*AbstractSWEType"* are used for all complex types instead of "*gml:AbstractGMLType*".

Following ISO 19136 encoding rules, each UML class with a <<Type>> or <<DataType>> stereotype, or no stereotype at all, is implemented in the schema as a global XML complex type with a corresponding global XML element (called object element). Each of these elements has the same name as the UML class (i.e. always UpperCamelCase) and the name of the associated complex type is a concatenation of this name and the word "Type".

Each UML class attribute is implemented either as a global complex type and a corresponding local element (called property element), or as an XML attribute. Each property complex type is given a name composed of the UML attribute name (always lowerCamelCase) and the words "PropertyType". The element is defined locally within the complex type representing the class carrying the attribute and named exactly like the attribute in UML (i.e. no global elements are created for class attributes). Class associations are implemented similarly except they cannot be implemented as an XML attributes.

8.1.2 IDs and Linkable Properties

As in GML, the schemas defined in this standard make extensive use of "*xlink*" features to support hypertext referencing in XML. This allows most property elements to reference content either internally or externally to the instance document, instead of including this content inline. This is supported by extensive use of the "*gml:id*" attribute (taking an xs:ID as its value) on most object elements, and of the GML attribute group, "*gml:AssociationAttributeGroup*", on most property elements.

In properties that support "*xlink*" attributes, one can usually choose to define that property value inline, as in:

```
<swe:field>
 <swe:Quantity id="TEMP" … />
</swe:field>
```

One can then reference an object within the same document by its ID:

```
<swe:field xlink:href="#TEMP"/>
```

An object within an external document can also be referenced by including the full URI:

```
<swe:field xlink:href="http://www.my.com/fields.xml#TEMP"/>
```

Typically, "*xlink*" references will be specified as URLs or as URNs that can be easily resolved through registries. It is required that the property has either the "*xlink:href*" attribute set or contain an inline value, even though this cannot be enforced by XML schema.

Req 56 **A property element supporting the "gml:AssociationAttributeGroup" shall contain the value inline or populate the "xlink:href" attribute with a valid reference but shall not be empty.**

8.1.3 Extensibility Points

The SWE Common Data Model schemas define extensibility points that can be used to insert ad-hoc XML content that is not defined by this standard. This is done via optional "*extension*" elements of type "*xs:anyType*" in the base abstract complex type "*AbstractSWEType*". Since all object types defined in this standard derive from this base type, extensions can be added anywhere in a SWE Common instance.

This mechanism allows for a "laxist" way of including extended content in XML instances as the extended content is by default ignored by the validator. However, it is also possible to formally validate extended content by writing an XML schema for the extension and feeding it to the validator via the "*xsi:schemaLocation*" attribute in all instances using the extension.

The recommended way of extending the XML schema of this standard is by defining new properties on existing objects by inserting them in an "*extension*" slot. Additionally this should be done in a way that these new properties can be safely ignored by an implementation that is not compatible with a given extension. Defining new XML object elements (such as new data component objects) rather than new properties will greatly reduce forward compatibility of implementations compliant to this standard with XML instances containing extensions of this standard.

In any case, all extensions of the XML schema described in this standard shall be defined in a new namespace (other than the namespaces used by this standard and its dependencies) in order to allow easy detection of extensions by implementations.

Req 57 **All extensions of the XML schemas described in this standard shall be defined in a new unique namespace.**

Extensions are not allowed to change the meaning or behavior of elements and types defined by this standard in any way (in this case, new classes or properties shall me

defined). This guarantees that implementations that have no knowledge of an extension can still properly use XML instances containing these extensions.

Req 58 **Extensions of this standard shall not redefine or change the meaning or behavior of XML elements and types defined in this standard.**

8.2 Requirements Class: Basic Types and Simple Components Schemas

XML Schema elements and types defined in the "*basic_types.xsd*" and "*simple_components.xsd*" schema files implement all classes defined respectively in the "Basic Types" and "Simple Components" UML packages.

Req 59 **An implementation passing the "Basic Types and Simple Components Schemas" conformance test class shall first pass the "XML Encoding Principles" and core conformance test classes.**

Req 60 **An implementation passing the "Basic Types and Simple Components Schemas" conformance test class shall first pass the "Abstract test suite for GML documents" conformance test class of the GML 3.2.1 standard.**

Req 61 **A compliant XML instance shall be valid with respect to the XML grammar defined in the "basic_types.xsd" and "simple_components.xsd" XML as well as satisfy all Schematron patterns defined in "simple_components.sch".**

8.2.1 Base Abstract Complex Types

Several base abstract types are defined in the "*basic_types.xsd*" schema file. They are used as base substitution groups for all global XML elements defined in this standard. Below are XML schema snippet for the "*AbstractSWE*", "*AbstractSWEIdentifiable*" and "*AbstractSWEValue*" elements and corresponding complex types:

```
<element name="AbstractSWE" abstract="true" substitutionGroup="gml:AbstractObject"
         type="swe:AbstractSWEType"/>

<complexType name="AbstractSWEType">
  <sequence>
    <element name="extension" minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <any namespace="##other" processContents="lax"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
  <attribute ref="gml:id" use="optional"/>
</complexType>
```

The "*AbstractSWE*" complex type is the base for all derived complex types defined in this standard. It defines a first extension mechanism as an optional "*extension*" element that allows for any extended element content (in a namespace other than the SWE Common Data Model namespace). It also has an optional "*gml:id*" attribute allowing referencing the object that derives from it.

```
<element name="AbstractSWEIdentifiable" abstract="true" substitutionGroup="swe:AbstractSWE"
         type="swe:AbstractSWEIdentifiableType"/>

<complexType name="AbstractSWEIdentifiableType">
  <complexContent>
    <extension base="swe:AbstractSWEType">
      <sequence>
        <element ref="gml:description" minOccurs="0"/>
        <element ref="gml:identifier" minOccurs="0"/>
        <element ref="gml:name" minOccurs="0"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

The "*AbstractSWEIdentifiable*" complex type derives from "*AbstractSWE*" and adds three identification elements extracted from the GML schema. These elements are to be used as described in the UML section of this standard which is compliant with the definition of these elements in the GML standard.

```
<element name="AbstractSWEValue" abstract="true" substitutionGroup="gml:AbstractValue"
         type="swe:AbstractSWEValueType"/>

<complexType name="AbstractSWEValueType">
  <sequence>
    <element name="extension" minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <any namespace="##other" processContents="lax"/>
        </sequence>
      </complexType>
    </element>
    <element ref="gml:description" minOccurs="0"/>
    <element ref="gml:identifier" minOccurs="0"/>
    <element ref="gml:name" minOccurs="0"/>
  </sequence>
  <attribute ref="gml:id" use="optional"/>
</complexType>
```

The "*AbstractSWEValue*" complex type resembles "*AbstractSWEIdentifiable*" but does not derive from it to avoid double derivation. This base type is indeed used to place all elements derived from it (i.e. all data components) in the "*gml:AbstractValue*" substitution group. This is important because data components defined in this standard are enhanced version of value objects defined in GML. This derivation method thus allows reuse of SWE Common elements within GML application schemas and in particular for describing coverage range parameters.

The following XML elements and complex types are defined in the "*simple_components.xsd*" schema file:

```
<element name="AbstractDataComponent" abstract="true" substitutionGroup="swe:AbstractSWEValue"
         type="swe:AbstractDataComponentType"/>
```

```
<complexType name="AbstractDataComponentType" abstract="true">
  <complexContent>
    <extension base="swe:AbstractSWEValueType">
      <attribute name="definition" type="anyURI" use="required"/>
      <attribute name="updatable" type="boolean" use="optional" default="false"/>
      <attribute name="optional" type="boolean" use="optional" default="false"/>
    </extension>
  </complexContent>
</complexType>
```

The "*AbstractDataComponent*" complex type adds XML attributes as defined in the UML class with the same name. The meaning of the corresponding UML class attributes is detailed in clause §7.2.3.

Req 62 The "definition" attribute shall contain a URI that can be resolved to the complete human readable definition of the property that is represented by the data component.

```
<element name="AbstractSimpleComponent" abstract="true"
         substitutionGroup="swe:AbstractDataComponent" type="swe:AbstractSimpleComponentType"/>

<complexType name="AbstractSimpleComponentType" abstract="true">
  <complexContent>
    <extension base="swe:AbstractDataComponentType">
      <sequence>
        <element name="quality" type="swe:QualityPropertyType" minOccurs="0"
                 maxOccurs="unbounded"/>
        <element name="nilValues" type="swe:NilValuesPropertyType" minOccurs="0"/>
      </sequence>
      <attribute name="referenceFrame" type="anyURI" use="optional"/>
      <attribute name="axisID" type="string" use="optional"/>
    </extension>
  </complexContent>
</complexType>
```

The "*AbstractSimpleComponent*" complex type adds XML attributes as defined in the UML class with the same name. The meaning of the corresponding UML properties is detailed in clause §7.2.4. The "*definition*" attribute is mandatory on all elements derived from "*AbstractSimpleComponentType*" (see Req 19 of UML model). This is enforced by a Schematron pattern.

As the XML schema snippet shows, this abstract element contains two important property elements "*quality*" and "*nilValues*" as well as two attributes "*referenceFrame*" and "*axisID*" implementing the corresponding attributes in the UML. Since all simple data components defined in this schema derive from this base type, these elements and attributes are available on all of them. Examples in the following sub-clauses show their usage. Detailed content of the "*Quality*" and "*NilValues*" elements that are the values of "*QualityPropertyType*" and "*NilValuesPropertyType*" respectively are given in clause §8.2.12 and §8.2.13.

Most simple data components (defined in the following sub-clauses) also allow for the definition of constraints via the "constraint" property element. When such constraints are specified, the value of the component (either inline or in a separate data block) shall always satisfy these constraints.

Req 63 The inline value included in an instance of a simple data component shall satisfy the constraints specified by this instance.

8.2.2 Boolean Element

The "*Boolean*" element is the XML schema implementation of the "*Boolean*" UML class defined in clause §7.2.5. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="Boolean" substitutionGroup="swe:AbstractSimpleComponent"
type="swe:BooleanType"/>

<complexType name="BooleanType">
  <complexContent>
    <extension base="swe:AbstractSimpleComponentType">
      <sequence>
        <element name="value" maxOccurs="1" minOccurs="0" type="boolean"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

The following instance shows how this element is used in practice to wrap a value generated by a motion detector:

```
<swe:Boolean definition="urn:ogc:def:property:OGC::Motion">
  <gml:description>True when motion was detected in the room</gml:description>
  <gml:name>Motion Detected</gml:name>
  <swe:value>true</swe:value>
</swe:Boolean>
```

Without the value it can serve as data descriptor for values that are encoded separately. The following example shows how it is used in SPS to define a Boolean tasking parameter. It is used as the definition of the input parameter, and so does not contain the value:

```
<swe:Boolean definition="urn:ogc:def:property:OGC::Reset">
  <gml:description>Set to true to force sensor reset</gml:description>
  <gml:name>Sensor Reset</gml:name>
</swe:Boolean>
```

8.2.3 Text Element

The "*Text*" element is the XML schema implementation of the "*Text*" UML class defined in clause §7.2.6. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="Text" substitutionGroup="swe:AbstractSimpleComponent" type="swe:TextType"/>

<complexType name="TextType">
  <complexContent>
    <extension base="swe:AbstractSimpleComponentType">
      <sequence>
```

```
      <element name="constraint" maxOccurs="1" minOccurs="0"
               type="swe:AllowedTokensPropertyType"/>
      <element name="value" maxOccurs="1" minOccurs="0" type="string"/>
    </sequence>
   </extension>
  </complexContent>
</complexType>
```

Constraints can be expressed by using the "*AllowedTokens*" element detailed in clause §8.2.14. The "*value*" property element is of the XML schema type "*string*" and it can contain XML entities if special characters (i.e. not allowed in XML) are required.

The following instance shows how this element can be used to describe a "plate number":

```
<swe:Text definition="urn:ogc:def:property:OGC::PlateNumber">
  <gml:description>Plate number detected by traffic webcam</gml:description>
  <gml:name>Plate Number</gml:name>
  <swe:value>5689 ABT 31</swe:value>
</swe:Text>
```

The plate number value that would be present in the corresponding data file or stream would then have to include a value that matches the pattern such as "5491 AB 31".

8.2.4 Category Element

The "*Category*" element is the XML schema implementation of the "*Category*" UML class defined in clause §7.2.7. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="Category" substitutionGroup="swe:AbstractSimpleComponent" type="swe:CategoryType"/>

<complexType name="CategoryType">
  <complexContent>
    <extension base="swe:AbstractSimpleComponentType">
      <sequence>
        <element name="codeSpace" maxOccurs="1" minOccurs="0" type="gml:ReferenceType"/>
        <element name="constraint" maxOccurs="1" minOccurs="0"
                 type="swe:AllowedTokensPropertyType"/>
        <element name="value" maxOccurs="1" minOccurs="0" type="string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

The "*codeSpace*" element is of type "*gml:ReferenceType*" and thus makes use of an "*xlink:href*" XML attribute to reference an external dictionary, taxonomy or ontology representing the code space (Please see the full description of "*gml:ReferenceType*" in the GML 3.2.1 specification for more details).

Constraints can be expressed by using the "*AllowedTokens*" element detailed in clause §0. The "*value*" property element is of the XML schema type "*string*". The text content of this element should however be limited to short tokens in the case of a categorical representation.

The following example shows how this XML element can be used to give the value of a geological period:

```
<swe:Category definition="urn:ogc:def:property:OGC:GeologicalPeriod">
  <gml:description>
    Name of the geological period according to the nomenclature of the
    International Commission on Stratigraphy
  </gml:description>
  <gml:name>Geological Period</gml:name>
  <swe:codeSpace xlink:href="urn:cgi:classifierScheme:BGS:DIC_GEOCHRON.TRANSLATION"/>
  <swe:value>Jurassic</swe:value>
</swe:Category>
```

Note that the code space references an existing dictionary defined by CGI (Commission for the Management and Application of Geoscience Information). This shows how SWE Common can leverage an existing community managed terminology.

Alternatively it can be used without the value to describe, for instance, a “bird species” field in a biology dataset:

```
<swe:Category definition="urn:ogc:def:property:OGC::Species">
  <gml:description>
    Bird species according to the classification of the World Bird Database
  </gml:description>
  <gml:name>Bird Species</gml:name>
  <swe:codeSpace xlink:href="http://www.birdlife.org/datazone/species/index.html"/>
</swe:Category>
```

In this example, no official code space URI was found so the “*codeSpace*” element is used to reference the online source of the taxonomy (The “birdlife.org” website hosts the international database that is the reference in this case).

If no code space is specified, Req 27 must be satisfied by inserting a constraint with a list of allowed values as shown in the example below:

```
<swe:Category definition="urn:ogc:def:property:OGC::SensorStatus">
  <gml:description>Current status of the sensor</gml:description>
  <gml:name>Sensor Status</gml:name>
  <swe:constraint>
    <swe:AllowedTokens>
      <swe:value>Off</swe:value>
      <swe:value>Stand-by</swe:value>
      <swe:value>Ready</swe:value>
      <swe:value>Busy</swe:value>
    </swe:AllowedTokens>
  </swe:constraint>
</swe:Category>
```

Note that in this case, the data consumer has no way of knowing the exact meaning of each allowed value, since there is no associated description. A code space is thus more explicit as it defines not only the list of allowed terms but should also give a textual description. This is why a code space is preferred whenever it is possible to implement it.

8.2.5 Count Element

The "*Count*" element is the XML schema implementation of the "*Count*" UML class defined in clause §7.2.8. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="Count" substitutionGroup="swe:AbstractSimpleComponent" type="swe:CountType"/>

<complexType name="CountType">
  <complexContent>
    <extension base="swe:AbstractSimpleComponentType">
      <sequence>
        <element name="constraint" maxOccurs="1" minOccurs="0"
                 type="swe:AllowedValuesPropertyType"/>
        <element name="value" maxOccurs="1" minOccurs="0" type="integer"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

Constraints are expressed by using the "*AllowedValues*" element detailed in clause §8.2.15. The "*value*" property element is of the XML schema type "*integer*".

The following example shows how this XML element can be used to give the value of a geological period:

```
<swe:Count definition="urn:ogc:def:property:OGC::NumberOfPixels">
  <gml:description>Number of pixels in each row of the image</gml:description>
  <gml:name>Row Size</gml:name>
  <swe:value>1024</swe:value>
</swe:Count>
```

Alternatively it can be used without the value just like any other component.

8.2.6 Quantity Element

The "*Quantity*" element is the XML schema implementation of the "*Quantity*" UML class defined in clause §7.2.9. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="Quantity" substitutionGroup="swe:AbstractSimpleComponent" type="swe:QuantityType"/>

<complexType name="QuantityType">
  <complexContent>
    <extension base="swe:AbstractSimpleComponentType">
      <sequence>
        <element name="uom" type="swe:UnitReferencePropertyType"/>
        <element name="constraint" maxOccurs="1" minOccurs="0"
                 type="swe:AllowedValuesPropertyType"/>
        <element name="value" maxOccurs="1" minOccurs="0" type="double"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

The "*uom*" property element is of type "*UnitReferencePropertyType*" whose complex type definition is given below:

```
<complexType name="UnitReferencePropertyType">
  <attribute name="code" type="gml:UomSymbol" use="optional"/>
  <attributeGroup ref="gml:AssociationAttributeGroup"/>
</complexType>
```

This type allows defining a unit of measure by its code (using the "*code*" attribute) or by using an "*xlink:href*" from the "*gml:AssociationAttributeGroup*" to reference a unit defined externally. Defining the unit of measure by its code expressed using the Unified Code for Units of Measure (UCUM) is the mandatory way unless the unit cannot be properly expressed with the elements defined in this specification (in which case "*xlink:href*" should be used).

Req 64 **The UCUM code for a unit of measure shall be used as the value of the "code" XML attribute whenever it can be constructed using the UCUM 1.8 specification. Otherwise the "href" XML attribute shall be used to reference an external unit definition.**

Constraints can be expressed by using the "*AllowedValues*" element detailed in clause §8.2.15. The "*value*" property element takes a decimal value of the XML schema type "*double*". This means that special values "–INF", "+INF" and "NaN" (for Not a Number) are allowed as well as numbers in exponent notation (ex: 1.2e-9).

The following example shows how this XML element can be used to wrap a continuous measurement value. In this case, a temperature measurement with a value of 21.5°C is shown:

```
<swe:Quantity definition="urn:ogc:def:property:OGC::AtmosphericTemperature">
  <gml:description>Outside temperature taken at the top of the antenna</gml:description>
  <gml:name>Outside Temperature</gml:name>
  <swe:uom code="Cel"/>
  <swe:value>21.5</swe:value>
</swe:Quantity>
```

The following example illustrates the use of a more complex UCUM unit for a radiance measurement (The value is 0.0283 watts per square meter per steradian per micrometer):

```
<swe:Quantity definition="urn:ogc:def:property:OGC::Radiance">
  <gml:description>Radiance measured on band1</gml:description>
  <gml:name>Radiance</gml:name>
  <swe:uom code="W.m-2.Sr-1.um-1"/>
  <swe:value>2.83e-2</swe:value>
</swe:Quantity>
```

The "*Quantity*" element is also often used to define projected quantities. For example, it can be used to define the altitude of a plane with respect to a well defined vertical reference system:

```
<swe:Quantity definition="urn:ogc:def:property:OGC::Height" axisID="H"
              referenceFrame="urn:ogc:def:crs:EPSG:7.1:5714">
  <gml:description>Height above mean sea level</gml:description>
  <gml:name>MSL Height</gml:name>
```

```
  <swe:uom code="m"/>
  <swe:value>1320</swe:value>
</swe:Quantity>
```

The "*referenceFrame*" attribute is used here to reference a well know vertical coordinate reference system unambiguously defined in the EPSG database. This example means that the height is measured along the H axis of the EPSG reference system code 5714 (Mean Sea Level Height), and has a value of 1320 meters.

Like in any other data component the "*value*" property element can be omitted when this element is used as a data descriptor for a field which value is provided separately.

8.2.7 Time Element

The "*Time*" element is the XML schema implementation of the "*Time*" UML class defined in clause §7.2.10. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="Time" substitutionGroup="swe:AbstractSimpleComponent" type="swe:TimeType"/>

<complexType name="TimeType">
  <complexContent>
    <extension base="swe:AbstractSimpleComponentType">
      <sequence>
        <element name="uom" type="swe:UnitReferencePropertyType"/>
        <element name="constraint" maxOccurs="1" minOccurs="0"
                 type="swe:AllowedTimesPropertyType"/>
        <element maxOccurs="1" minOccurs="0" name="value" type="swe:TimePosition"/>
      </sequence>
      <attribute name="referenceTime" type="dateTime" use="optional"/>
      <attribute name="localFrame" type="anyURI" use="optional"/>
    </extension>
  </complexContent>
</complexType>
```

The "*uom*" property element is of type "*UnitReferencePropertyType*". It has the same requirements as its equivalent in the "*Quantity*" element. When ISO 8601 calendar notation is used, it is specified as a unit by using a special value in the "*xlink:href*" attribute (i.e. for simplicity, a calendar representation is considered here as a complex unit composed of year, month, day, hours, minutes and seconds).

Req 65 **When ISO 8601 notation is used to express the measurement value associated to a "Time" element, the URI "urn:ogc:def:unit:ISO:8601" shall be used as the value of the "xlink:href" XML attribute on the "uom" element.**

Additional constraints on the value can be expressed by using the "*AllowedTimes*" element detailed in clause §8.2.16. The "*value*" property element takes either a decimal value or a calendar value encoded according to the ISO 8601 standard. This is enforced by using the "*TimePosition*" simple type defined below as the union of the "*double*" and "*TimeIso8601*" data types:

```
<simpleType name="TimePosition">
```

```
  <union memberTypes="double swe:TimeIso8601"/>
</simpleType>

<simpleType name="TimeIso8601">
  <union memberTypes="date time dateTime gml:TimeIndeterminateValueType"/>
</simpleType>
```

The "*double*" data type is used to express time as a scalar decimal number (i.e. a duration) and can be any of the special values "-INF", "+INF" and "NaN" just like the value of a "*Quantity*" component.

The "*date*", "*time*" and "*dateTime*" data types are built-in types of XML Schema and are implemented according to ISO 8601 complete representations of date, time and combination of date and time respectively (see XML schema 1.0 specification and ISO 8601 for details on the format).

The example below shows how to use this XML element to specify the sampling time of a measurement:

```
<swe:Time definition="urn:ogc:def:property:OGC::SamplingTime"
          referenceFrame="urn:ogc:def:crs:OGC::GPS">
  <gml:description>Time at which the measurement was made</gml:description>
  <gml:name>Sampling Time</gml:name>
  <swe:uom xlink:href="urn:ogc:def:unit:ISO:8601"/>
  <swe:value>2009-11-05T16:29:26Z</swe:value>
</swe:Time>
```

Note the "*referenceFrame*" attribute which clarifies the time reference system used. Here the GPS time standard, which is different from UTC and TAI is used. The presence of the mandatory "*referenceFrame*" attribute (see Req 29) is enforced by an additional Schematron assertion.

As mentioned above, the "*Time*" element can also be used to specify a time after an epoch by specifying a time of reference and using a scalar numerical value. This is shown in the following example with a UNIX time:

```
<swe:Time definition="urn:ogc:def:property:OGC::RunTime"
          referenceFrame="urn:ogc:def:crs:OGC::UTC" referenceTime="1970-01-01T00:00:00Z">
  <gml:description>Run time of the model express as a Unix time</gml:description>
  <gml:name>Model Run Time</gml:name>
  <swe:uom code="s"/>
  <swe:value>1257415633</swe:value>
</swe:Time>
```

The "*localFrame*" attribute can be used to indicate the time frame whose origin is given by the time component value. This way several time positions can be defined relative to each other. The next example shows how this can be used to express times of high frequency scan lines acquired by an airborne scanner relative to the flight's start time:

```
<swe:Time definition="urn:ogc:def:property:OGC::MissionTime"
          localFrame="urn:ift:MISSION_01265:START_TIME"
          referenceFrame="urn:ogc:def:crs:OGC::UTC">
  <gml:description>Time at take-off in UTC</gml:description>
```

```
  <gml:name>Flight Time</gml:name>
  <swe:uom xlink:href="urn:ogc:def:unit:ISO:8601"/>
  <swe:value>2009-01-26T10:21:45+01:00</swe:value>
</swe:Time>
```

Scan times are then expressed relative to the flight's start time :

```
<swe:Time definition="urn:ogc:def:property:OGC::ScanTime"
          referenceFrame="urn:ift:MISSION_01265:START_TIME">
  <gml:description>Acquisition time of the scan line</gml:description>
  <gml:name>Scanline Time</gml:name>
  <swe:uom code="s"/>
  <swe:value>1256.235</swe:value>
</swe:Time>
```

In the snippet above the reference frame is the previously defined mission start time which means that the time value is relative to this time of reference. The value can then be encoded as a float for better efficiency.

Note: A simple duration expressed outside of any time reference system should be defined by using a "Quantity" rather than a "Time" element.

8.2.8 CategoryRange Element

The "*CategoryRange*" element is the XML schema implementation of the "*CategoryRange*" UML class defined in clause §7.2.12. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="CategoryRange" substitutionGroup="swe:AbstractSimpleComponent"
         type="swe:CategoryRangeType"/>

<complexType name="CategoryRangeType">
  <complexContent>
    <extension base="swe:AbstractSimpleComponentType">
      <sequence>
        <element name="codeSpace" maxOccurs="1" minOccurs="0" type="gml:DictionaryEntryType"/>
        <element name="constraint" maxOccurs="1" minOccurs="0"
                 type="swe:AllowedTokensPropertyType"/>
        <element name="value" maxOccurs="1" minOccurs="0" type="swe:TokenPair"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

This element is used exactly in the same way as the "*Category*" element except that the "*value*" property takes a space separated pair of tokens. The example below illustrates the representation of an approximative dating as a range of geological eras:

```
<swe:CategoryRange definition="urn:ogc:def:property:CGI::GeologicalDating">
  <gml:description>
    Approximate geological dating expressed as a range of geological eras
  </gml:description>
  <gml:name>Approximate Dating</gml:name>
  <swe:codeSpace xlink:href="urn:cgi:classifierScheme:BGS:DIC_GEOCHRON.TRANSLATION"/>
  <swe:value>Paleozoic Mesozoic</swe:value>
</swe:CategoryRange>
```

The pair of values can be omitted like with any other data component in the case where it is provided in a separate data stream.

8.2.9 CountRange Element

The "*CountRange*" element is the XML schema implementation of the "*CountRange*" UML class defined in clause §7.2.13. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="CountRange" substitutionGroup="swe:AbstractSimpleComponent"
         type="swe:CountRangeType"/>

<complexType name="CountRangeType">
  <complexContent>
    <extension base="swe:AbstractSimpleComponentType">
      <sequence>
        <element name="constraint" maxOccurs="1" minOccurs="0"
                 type="swe:AllowedValuesPropertyType"/>
        <element name="value" maxOccurs="1" minOccurs="0" type="swe:IntegerPair"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

This element is used exactly in the same way as the "*Count*" element except that the "*value*" property takes a space separated pair of integers. The next example shows how to specify an array index range:

```
<swe:Count definition="urn:ogc:def:property:OGC::IndexRange">
  <gml:name>Index Range</gml:name>
  <swe:value>0 9999</swe:value>
</swe:Count>
```

The pair of values can be omitted like with any other data component in the case where it is provided in a separate data stream.

8.2.10 QuantityRange Element

The "*QuantityRange*" element is the XML schema implementation of the "*QuantityRange*" UML class defined in clause §7.2.14. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="QuantityRange" substitutionGroup="swe:AbstractSimpleComponent"
         type="swe:QuantityRangeType"/>

<complexType name="QuantityRangeType">
  <complexContent>
    <extension base="swe:AbstractSimpleComponentType">
      <sequence>
        <element name="uom" type="swe:UnitReferencePropertyType"/>
        <element name="constraint" maxOccurs="1" minOccurs="0"
                 type="swe:AllowedValuesPropertyType"/>
        <element name="value" maxOccurs="1" minOccurs="0" type="swe:RealPair"/>
      </sequence>
    </extension>
  </complexContent>
```

```
</complexType>
```

This element is used exactly in the same way as the "*Quantity*" element except that the "*value*" property takes a space separated pair of double values. The next example shows how to express the dynamic range of a thermometer in Kelvins:

```
<swe:QuantityRange definition="urn:ogc:def:property:OGC::DynamicRange">
  <gml:description>Dynamic range of the cryogenic thermometer</gml:description>
  <gml:name>Dynamic Range</gml:name>
  <swe:uom code="K"/>
  <swe:value>10 300</swe:value>
</swe:QuantityRange>
```

The pair of values can be omitted like with any other data component in the case where it is provided in a separate data stream.

8.2.11 TimeRange Element

The "*TimeRange*" element is the XML schema implementation of the "*TimeRange*" UML class defined in clause §7.2.15. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="TimeRange" substitutionGroup="swe:AbstractSimpleComponent"
         type="swe:TimeRangeType"/>

<complexType name="TimeRangeType">
  <complexContent>
    <extension base="swe:AbstractSimpleComponentType">
      <sequence>
        <element name="uom" type="swe:UnitReferencePropertyType"/>
        <element name="constraint" maxOccurs="1" minOccurs="0"
                 type="swe:AllowedTimesPropertyType"/>
        <element name="value" maxOccurs="1" minOccurs="0" type="swe:TimePair"/>
      </sequence>
      <attribute name="referenceTime" type="dateTime" use="optional"/>
      <attribute name="localFrame" type="anyURI" use="optional"/>
    </extension>
  </complexContent>
</complexType>
```

This element is used exactly in the same way as the "*Time*" element except that the "*value*" property takes a space separated pair of time positions. The next example shows how to express a time period with such a component:

```
<swe:TimeRange definition="urn:ogc:def:property:OGC::SurveyPeriod"
               referenceFrame="urn:ogc:def:crs:OGC::UTC">
  <gml:name>Survey Period</gml:name>
  <swe:uom xlink:href="urn:ogc:def:unit:ISO:8601"/>
  <swe:value>2008-01-05T11:02:54Z 2009-11-05T16:29:26Z</swe:value>
</swe:TimeRange>
```

The pair of time positions can be omitted like with any other data component in the case where it is provided in a separate data stream.

8.2.12 Quality Element Group

The "*Quality*" group is the XML schema implementation of the "*Quality*" Union UML classifier defined in clause §7.2.16. The schema snippet for this XML schema group is shown below:

```
<group name="Quality">
  <choice>
    <element ref="swe:Quantity"/>
    <element ref="swe:QuantityRange"/>
    <element ref="swe:Category"/>
    <element ref="swe:Text"/>
  </choice>
</group>
```

This group allows the use of some of the XML elements define above to add qualitative information to a simple data component. The following examples illustrate how this is done in a SWE Common XML instance.

This first example shows that quality is expressed by wrapping the value in one of the data components defined previously that is appropriate for the desired representation. Here a "*Quantity*" element is used to specify a decimal value representing relative accuracy:

```
<swe:Quantity definition="urn:ogc:def:property:OGC::RelativeAccuracy">
  <gml:name>Relative Accuracy</gml:name>
  <swe:uom code="%"/>
  <swe:value>5</swe:value>
</swe:Quantity>
```

This snippet is then inserted within the data component element whose value's quality needs to be expressed. This is shown below:

```
<swe:Quantity definition="urn:ogc:def:property:OGC::Velocity">
  <gml:description>Linear velocity of the vehicle</gml:description>
  <gml:name>Velocity</gml:name>
  <swe:quality>
    <swe:Quantity definition="urn:ogc:def:property:OGC::RelativeAccuracy">
      <gml:name>Relative Accuracy</gml:name>
      <swe:uom code="%"/>
      <swe:value>5</swe:value>
    </swe:Quantity>
  </swe:quality>
  <swe:uom code="m/s"/>
  <swe:value>23.5</swe:value>
</swe:Quantity>
```

This example is a velocity measurement of 23.5 meters per seconds, with a relative accuracy of 5%. Absolute accuracy could have been specified as well by using a different definition URI and setting the unit of the accuracy value to "m/s".

Bidirectional tolerance is a measure of quality that is often used for specification of mechanical parts. Such a use case is shown below:

```
<swe:Quantity definition="urn:ogc:def:property:OGC::Diameter">
  <gml:description>Diameter of the cylinder</gml:description>
```

```
  <gml:name>Diameter</gml:name>
  <swe:quality>
    <swe:QuantityRange definition="urn:ogc:def:property:OGC::Tolerance">
      <gml:name>Dimensional Tolerance</gml:name>
      <swe:uom code="um"/>
      <swe:value>-20 +0</swe:value>
    </swe:QuantityRange>
  </swe:quality>
  <swe:uom code="mm"/>
  <swe:value>5.6</swe:value>
</swe:Quantity>
```

In the previous example, the cylinder is specified to have a diameter between 5.58 and 5.6 millimeters. Note that a different unit (i.e. micrometer) is used for the tolerance value.

The following example shows the use of a categorical representation of quality in order to implement a pass/fail quality control flag as defined in the MMI (Marine Metadata Interoperability) ontology:

```
<swe:Quantity definition="urn:ogc:def:property:OGC::Pressure">
  <gml:description>Water pressure measured by CTD</gml:description>
  <gml:name>Water Pressure</gml:name>
  <swe:quality>
    <swe:Category definition="urn:ogc:def:property:OGC::QualityControlFlag">
      <gml:name>QC Flag</gml:name>
      <swe:codeSpace xlink:href="http://mmisw.org/ont/q2o/flag"/>
      <swe:value>fail</swe:value>
    </swe:Category>
  </swe:quality>
  <swe:uom code="dbar"/>
  <swe:value>1084</swe:value>
</swe:Quantity>
```

All previous examples shows how quality can be given along with the inline value. However this standard allows specifying quality in a data descriptor, which means that the qualitative information applies to all data values represented by the component in a separately encoded data stream. This is just achieved by using the component with no inline values.

Additionally the quality value can be given in the encoded data stream along with the measurement values when the quality component is defined itself as a field of the dataset. This is shown in clause §7.4.3 describing the "*DataStream*" element.

The "*quality*" property element should never be used recursively by an implementation (i.e. This property should not be used within a data component that is himself used as an instance of the "*Quality*" group). Indeed, although it is theoretically acceptable to describe the quality of the qualitative information itself, it is a practice that would greatly complexify the analysis of such metadata and is thus strongly discouraged.

8.2.13 NilValues Element

The "*NilValues*" element is the XML schema implementation of the "*NilValues*" UML class defined in clause §7.2.17. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="NilValues" substitutionGroup="gml:AbstractSWE" type="swe:NilValuesType"/>

<complexType name="NilValuesType">
  <complexContent>
    <extension base="swe:AbstractSWEType">
      <sequence>
        <element name="nilValue" type="swe:NilValuePropertyType" maxOccurs="unbounded"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="NilValuePropertyType">
  <simpleContent>
    <extension base="token">
      <attribute name="reason" type="anyURI" use="required"/>
    </extension>
  </simpleContent>
</complexType>
```

This element allows specifying a list of nil value for a particular data component. The next example shows how it can be used to reserve values for indicating a bad measurement in a radiation sensor data stream.

```
<swe:Quantity definition="urn:ogc:def:property:OGC::RadiationDose">
  <gml:description>Radiation dose measured by Gamma detector</gml:description>
  <gml:name>Radiation Dose</gml:name>
  <swe:nilValues>
    <swe:NilValues>
      <swe:nilValue reason="urn:ogc:def:nil:OGC::BelowDetectionLimit">-INF</swe:nilValue>
      <swe:nilValue reason="urn:ogc:def:nil:OGC::AboveDetectionLimit">+INF</swe:nilValue>
    </swe:NilValues>
  </swe:nilValues>
  <swe:uom code="uR"/>
</swe:Quantity>
```

This means that if the "*-INF*" or "*+INF*" values (these are allowed values for a floating point encoding) are found in the data stream, they should not be taken as real measurement values but instead carry a specific meaning that is given by the "*reason*" attribute. In this example, all other values (i.e. all decimal numbers) should be interpreted as a radation dose expressed in micro-roentgens.

One important feature is that the "*NilValues*" object can be referenced instead of being included inline. In addition to allowing their definition in shared repositories, this also enables sharing nil value definitions between several components of the same dataset. This is for instance useful for describing multispectral and hyperspectral images since all bands in these types of images usually share the same nil values. The field representing the first band can then be defined as shown below:

```
<swe:Count definition="urn:ogc:def:property:OGC::Radiance">
  <gml:name>Band 1</gml:name>
  <swe:nilValues>
    <swe:NilValues gml:id="NIL_VALUES">
      <swe:nilValue reason="urn:ogc:def:nil:OGC::BelowDetectionLimit">0</swe:nilValue>
      <swe:nilValue reason="urn:ogc:def:nil:OGC::AboveDetectionLimit">255</swe:nilValue>
    </swe:NilValues>
  </swe:nilValues>
</swe:Count>
```

And the following bands can have a much shorter description as it just references the nil values group previously defined:

```
<swe:Count definition="urn:ogc:def:property:OGC::Radiance">
  <gml:name>Band 2</gml:name>
  <swe:nilValues xlink:href="#NIL_VALUES"/>
</swe:Count>
...
<swe:Count definition="urn:ogc:def:property:OGC::Radiance">
  <gml:name>Band 33</gml:name>
  <swe:nilValues xlink:href="#NIL_VALUES"/>
</swe:Count>
```

An important requirement of nil values is that they shall be expressible with the data component data type in order to guarantee that they can be properly encoded. This is enforced by a Schematron pattern.

For a field with a string data type (i.e. "*Category*" and "*Text*" components), each nil value can be any string but it is recommended to use short upper case alphabetical tokens for better readability.

For a field with a floating point data type (i.e. "*Quantity*" and "*Time*" components), nil values are restricted to decimal numbers and the three special values '+INF', '-INF', 'NaN'. These tokens shall be used when encoding nil values using the "*TextEncoding*" method. It is recommended to use these values to represent nil reasons whenever possible for clarity, but it is also possible to use special numbers such as '-9999' or '9e99', which are usually chosen outside of the sensor measurement range, for carrying NIL semantics.

For a field with an integer data type (i.e. "*Count*" component), only integer numbers such as '255' or '999' can be used for expressing NIL values. These are usually chosen outside of the measurement range, and in a way that the smallest possible data type can be used to store the data in memory (i.e. reserved values should be outside of the measurement range but as small as possible).

8.2.14 AllowedTokens Element

The "*AllowedTokens*" element is the XML schema implementation of the "*AllowedTokens*" UML class defined in clause §7.2.18. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="AllowedTokens" substitutionGroup="gml:AbstractSWE"
         type="swe:AllowedTokensType"/>

<complexType name="AllowedTokensType">
  <complexContent>
    <extension base="swe:AbstractSWEType">
      <sequence>
        <element name="value" type="string" minOccurs="0" maxOccurs="unbounded"/>
        <element name="pattern" type="string" minOccurs="0"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

This element is used to restrict the values allowed by both categorical and textual representations. An enumeration constraint used with a "*Category*" element is shown below:

```
<swe:Category definition="urn:ogc:def:property:OGC::SensorType">
  <gml:name>Instrument Type</gml:name>
  <swe:codeSpace xlink:href="http://mmisw.org/ont/bodc/instrument"/>
  <swe:constraint>
    <swe:AllowedTokens>
      <swe:value>Multi beam echosounder</swe:value>
      <swe:value>Temperature sensor</swe:value>
      <swe:value>Underwater camera</swe:value>
    </swe:AllowedTokens>
  </swe:constraint>
</swe:Category>
```

In this example, the values allowed by the code space (OWL ontology located at http://mmisw.org/ont/bodc/instrument) are further restricted by allowing only three of its members.

This element can also be used to specify a constraint with a regular expression pattern. This is shown below with a serial number example using a "*Text*" element:

```
<swe:Text definition="urn:ogc:def:property:OGC::SerialNumber">
  <gml:name>Serial Number</gml:name>
  <swe:constraint>
    <swe:AllowedTokens>
      <swe:pattern>^[0-9][A-Z]{3}[0-9]{2}S1$</swe:pattern>
    </swe:AllowedTokens>
  </swe:constraint>
</swe:Text>
```

The pattern shall follow the unicode regular expression syntax described in Unicode Technical Std #18. This is the same syntax as the one used by the XML Schema specification.

Req 66 **The "pattern" child element of the "AllowedTokens" element shall be a regular expression valid with respect to Unicode Technical Standard #18, Version 13.**

8.2.15 AllowedValues Element

The "*AllowedValues*" element is the XML schema implementation of the "*AllowedValues*" UML class defined in clause §7.2.19. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="AllowedValues" substitutionGroup="gml:AbstractSWE"
         type="swe:AllowedValuesType"/>

<complexType name="AllowedValuesType">
  <complexContent>
    <extension base="swe:AbstractSWEType">
      <sequence>
        <element name="value" type="double" minOccurs="0" maxOccurs="unbounded"/>
```

```
      <element name="interval" type="swe:RealPair" minOccurs="0" maxOccurs="unbounded"/>
      <element name="significantFigures" type="integer" minOccurs="0"/>
    </sequence>
   </extension>
  </complexContent>
</complexType>
```

This element is used to specify numerical constraints for the "*Count*" and "*Quantity*" elements and the corresponding range components. The XML snippet below illustrates how to constrain an angular value to the -180/180 degrees domain:

```
<swe:Quantity definition="urn:ogc:def:property:OGC::Angle">
  <gml:name>Planar Angle</gml:name>
  <swe:uom code="deg"/>
  <swe:constraint>
    <swe:AllowedValues>
      <swe:interval>-180 +180</swe:interval>
    </swe:AllowedValues>
  </swe:constraint>
</swe:Quantity>
```

Several intervals can be specified to generate holes with forbidden values. Using several intervals means that the value is constrained to be within one of the intervals:

```
<swe:AllowedValues>
  <swe:interval>-INF -20</swe:interval>
  <swe:interval>20 50</swe:interval>
  <swe:interval>60 +INF</swe:interval>
</swe:AllowedValues>
```

Note that the –Infnity and +Infinity are allowed in order to specify unbounded intervals. The above example indicates that the value must be either less than or equal to -20, between 20 and 50 included, or greater than or equal to 60. Intervals specified with this element are always inclusive and should not overlap.

Allowed values can also be enumerated as shown in the example below:

```
<swe:Count definition="urn:ogc:def:property:OGC::ObjectCount">
  <gml:description>Number of active cameras</gml:description>
  <gml:name>Active Cameras</gml:name>
  <swe:constraint>
    <swe:AllowedValues>
      <swe:value>1</swe:value>
      <swe:value>3</swe:value>
      <swe:value>6</swe:value>
    </swe:AllowedValues>
  </swe:constraint>
</swe:Count>
```

Several allowed intervals and values can also be combined to express a complex constraint even though this is rarely used in practice:

```
<swe:AllowedValues>
  <swe:value>5</swe:value>
  <swe:value>10</swe:value>
  <swe:interval>20 30</swe:interval>
  <swe:interval>40 60</swe:interval>
```

```
</swe:AllowedValues>
```

In the above example, the actual value must be 5, or 10, or between 20 and 30 included, or between 40 and 60 included. All numbers used within "*interval*" and "*value*" elements shall be expressed in the same unit as the enclosing data component (Req 39).

The last option to specify a constraint on a decimal number (so only applicable to "*Quantity*") is to limit the number of significant figures:

```
<swe:AllowedValues>
  <swe:significantFigures>5</swe:significantFigures>
</swe:AllowedValues>
```

Constraining a number to 5 significant figures means that a total of only 5 digits are or can be used for the representation of the value. The numbers 1.2345, 5.4823e-3, 0.98655 and 00235 are all composed of 5 significant figures, but 1.23450 and 658970 have six significant figures (leading zeros are ignored).

When decimal values are encoded with a binary floating point data type rather than text, restricting the number of significant figures also means that the lowest order fractional digits of the mantissa should be ignored even though they may be encoded due to rounding errors.

8.2.16 AllowedTimes Element

The "*AllowedTimes*" element is the XML schema implementation of the "*AllowedTimes*" UML class defined in clause §7.2.20. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="AllowedTimes" substitutionGroup="gml:AbstractSWE"
         type="swe:AllowedTimesType"/>

<complexType name="AllowedTimesType">
  <complexContent>
    <extension base="swe:AbstractSWEType">
      <sequence>
        <element name="value" type="swe:TimePosition" minOccurs="0" maxOccurs="unbounded"/>
        <element name="interval" type="swe:TimePair" minOccurs="0" maxOccurs="unbounded"/>
        <element name="significantFigures" type="integer" minOccurs="0"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

This element is used to specify numerical constraints with the "*Time*" and "*TimeRange*" elements. It is used in the same way as the "*AllowedValues*" element in "*Quantity*" when the temporal value is expressed as a decimal number, but also supports encoding enumerated values and intervals as ISO 8601 date times. The definition of a temporal field with a restriction to a certain period is shown below:

```
<swe:Time definition="urn:ogc:def:property:OGC::SamplingTime">
  <gml:name>Acquisition Time</gml:name>
```

```
  <swe:uom xlink:href="urn:ogc:def:unit:ISO:8601"/>
  <swe:constraint>
    <swe:AllowedTimes>
      <swe:interval>2009-01-01 +INF</swe:interval>
    </swe:AllowedTimes>
  </swe:constraint>
</swe:Time>
```

Note that unbounded intervals are also possible when ISO 8601 notation is used. In all other cases mixing decimal and ISO 8601 notation is forbidden. If the "*uom*" element indicates that ISO 8601 is the unit, then constraints shall be expressed using ISO 8601 notation as well, otherwise constraints shall be expressed with decimal numbers with the scale specified by "*uom*". This is enforced by a Schematron pattern (Req 39).

8.2.17 Simple Component Groups

Three XML element groups as well as the corresponding property types are defined in the schema in order to simplify their use in external schemas.

```
<group name="AnyScalar">
  <choice>
    <element ref="swe:Boolean"/>
    <element ref="swe:Count"/>
    <element ref="swe:Quantity"/>
    <element ref="swe:Time"/>
    <element ref="swe:Category"/>
    <element ref="swe:Text"/>
  </choice>
</group>

<group name="AnyNumerical">
  <choice>
    <element ref="swe:Count"/>
    <element ref="swe:Quantity"/>
    <element ref="swe:Time"/>
  </choice>
</group>

<group name="AnyRange">
  <choice>
    <element ref="swe:QuantityRange"/>
    <element ref="swe:TimeRange"/>
    <element ref="swe:CountRange"/>
    <element ref="swe:CategoryRange"/>
  </choice>
</group>
```

The "*AnyScalar*" group contains all scalar representations, "*AnyNumerical*" only numerical representations and the "*AnyRange*" group includes all range components.

8.3 Requirements Class: Aggregate Components Schema

XML Schema elements and types defined in the "*aggregate_components.xsd*" schema implement all classes defined in the "Aggregate Components" UML packages.

Req 67 **An implementation passing the "Aggregate Components Schema" conformance test class shall first pass the "Basic Types and Simple Components Schemas" conformance test class.**

Req 68 **A compliant XML instance shall be valid with respect to the XML grammar defined in the "aggregate_components.xsd" XML schema as well as satisfy all Schematron patterns defined in "aggregate_components.sch".**

8.3.1 DataRecord Element

The "*DataRecord*" element is the XML schema implementation of the "*DataRecord*" UML class defined in clause §7.3.1. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="DataRecord" type="swe:DataRecordType"
         substitutionGroup="swe:AbstractDataComponent"/>

<complexType name="DataRecordType">
  <complexContent>
    <extension base="swe:AbstractDataComponentType">
      <sequence>
        <element name="field" maxOccurs="unbounded">
          <complexType>
            <complexContent>
              <extension base="swe:AbstractDataComponentPropertyType">
                <attribute name="name" type="NCName" use="required"/>
              </extension>
            </complexContent>
          </complexType>
        </element>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

The element contains all sub-elements inherited from "*AbstractDataComponentType*" as well as a list of (at least one) "*field*" property elements, each with a "*name*" attribute and containing the data component element that defines the field.

The XML example below describes a record composed of weather data fields. In this case the "*DataRecord*" element is used as a data descriptor and the corresponding data stream is usually composed of several tuples of values, each tuple corresponding to one record as defined here:

```
<swe:DataRecord>
  <gml:description>Record of synchronous weather measurements</gml:description>
  <gml:name>Weather Data Record</gml:name>
  <swe:field name="time">
    <swe:Time definition="urn:ogc:def:property:OGC::SamplingTime">
      <gml:name>Sampling Time</gml:name>
      <swe:uom xlink:href="urn:ogc:def:unit:ISO:8601"/>
    </swe:Time>
  </swe:field>
  <swe:field name="temperature">
    <swe:Quantity definition="urn:ogc:def:property:OGC::AirTemperature">
      <gml:name>Air Temperature</gml:name>
      <swe:uom xlink:href="Cel"/>
```

```
    </swe:Quantity>
  </swe:field>
  <swe:field name="pressure">
    <swe:Quantity definition="urn:ogc:def:property:OGC::AtmosphericPressure">
      <gml:name>Atmospheric Pressure</gml:name>
      <swe:uom code="mbar"/>
    </swe:Quantity>
  </swe:field>
  <swe:field name="windSpeed">
    <swe:Quantity definition="urn:ogc:def:property:OGC::WindSpeed">
      <gml:name>Wind Speed</gml:name>
      <swe:uom code="km/h"/>
    </swe:Quantity>
  </swe:field>
  <swe:field name="windDirection">
    <swe:Quantity definition="urn:ogc:def:property:OGC::WindDirectionToNorth">
      <gml:name>Wind Direction</gml:name>
      <swe:uom code="deg"/>
    </swe:Quantity>
  </swe:field>
</swe:DataRecord>
```

Each field shall have a unique name within the record (Req 42). This is enforced by a Schematron pattern.

The "*DataRecord*" element can also carry its own "*definition*" attribute to carry semantics about the whole group of values. The next example shows how to encode radial distortion coefficients of a frame camera sensor:

```
<swe:DataRecord definition="urn:ogc:def:property:CSM:RadialDistortionCoefficients">
  <gml:name>Radial Distortion Coefficients</gml:name>
  <swe:field name="k1">
    <swe:Quantity definition="urn:ogc:def:property:CSM:DISTOR_RAD1">
      <swe:uom code="mm-2"/>
      <swe:value>1.92709e-005</swe:value>
    </swe:Quantity>
  </swe:field>
  <swe:field name="k2">
    <swe:Quantity definition="urn:ogc:def:property:CSM:DISTOR_RAD2">
      <swe:uom code="mm-2"/>
      <swe:value>-5.14206e-010</swe:value>
    </swe:Quantity>
  </swe:field>
  <swe:field name="k3">
    <swe:Quantity definition="urn:ogc:def:property:CSM:DISTOR_RAD3">
      <swe:uom code="mm-2"/>
      <swe:value>-3.33356e-012</swe:value>
    </swe:Quantity>
  </swe:field>
</swe:DataRecord>
```

The "*DataRecord*" element is fully recursive so that each field can itself be a "*DataRecord*", but most importantly each field can be any other data component defined in this standard (such as "*Vector*", "*DataChoice*" and "*DataArray*").

Examples above only make use of field components with minimum metadata, but each of these fields can have all the possible content defined in clause §8.2, including quality, constraints, etc.

8.3.2 DataChoice Element

The "*DataChoice*" element is the XML schema implementation of the "*DataChoice*" UML class defined in clause §7.3.2. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="DataChoice" type="swe:DataChoiceType"
         substitutionGroup="swe:AbstractDataComponent"/>

<complexType name="DataChoiceType">
  <complexContent>
    <extension base="swe:AbstractDataComponentType">
      <sequence>
        <element name="choiceValue" type="swe:CategoryPropertyType"/>
        <element name="item" minOccurs="2" maxOccurs="unbounded">
          <complexType>
            <complexContent>
              <extension base="swe:AbstractDataComponentPropertyType">
                <attribute name="name" type="NCName" use="required"/>
              </extension>
            </complexContent>
          </complexType>
        </element>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

This element contains a list of (at least two) "*item*" property elements, each with a "*name*" attribute and containing the data component element that defines the field.

The following "*DataChoice*" example illustrates how it can be used to define an element of a data stream that can either be a temperature or a pressure measurement, in both cases associated to a time tag:

```
<swe:DataChoice>
  <!-- -->
  <swe:choiceValue>
    <swe:Category definition="urn:ogc:def:data:OGC:MessageType"/>
  </swe:choiceValue>
  <!-- -->
  <swe:item name="TEMP">
    <swe:DataRecord>
      <gml:name>Temperature Measurement</gml:name>
      <swe:field name="time">
        <swe:Time definition="urn:ogc:def:property:OGC:SamplingTime"
                  referenceFrame="urn:ogc:def:crs:OGC::GPS">
          <swe:uom xlink:href="urn:ogc:def:unit:ISO:8601"/>
        </swe:Time>
      </swe:field>
      <swe:field name="temp">
        <swe:Quantity definition="urn:ogc:def:property:OGC:Temperature">
          <swe:uom code="Cel"/>
        </swe:Quantity>
      </swe:field>
    </swe:DataRecord>
  </swe:item>
  <!-- -->
  <swe:item name="PRESS">
    <swe:DataRecord>
      <gml:name>Pressure Measurement</gml:name>
      <swe:field name="time">
        <swe:Time definition="urn:ogc:def:property:OGC:SamplingTime"
                  referenceFrame="urn:ogc:def:crs:OGC::GPS">
          <swe:uom xlink:href="urn:ogc:def:unit:ISO:8601"/>
```

```
      </swe:Time>
    </swe:field>
    <swe:field name="press">
      <swe:Quantity definition="urn:ogc:def:property:OGC:Pressure">
        <swe:uom code="HPa"/>
      </swe:Quantity>
    </swe:field>
  </swe:DataRecord>
 </swe:item>
 <!-- -->
</swe:DataChoice>
```

A dataset element defined by the structure above would be of a variant type, meaning that each instance (actual data values) of this structure could be either a pair of time and temperature values OR a pair of time and pressure values.

Each choice item shall have a unique name within a given "*DataChoice*" element (Req 43). This is enforced by a Schematron pattern.

The "*DataChoice*" element is fully recursive so that each field can itself be any type of component defined in this standard, although implementations are not required to support nested "*DataChoice*" elements.

8.3.3 Vector Element

The "*Vector*" element is the XML schema implementation of the "*Vector*" UML class defined in clause §7.3.3. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="Vector" type="swe:VectorType" substitutionGroup="swe:AbstractDataComponent"/>

<complexType name="VectorType">
 <complexContent>
  <extension base="swe:AbstractDataComponentType">
   <sequence>
    <element name="coordinate" maxOccurs="unbounded">
     <complexType>
      <complexContent>
       <extension base="swe:AnyNumericalPropertyType">
        <attribute name="name" type="NCName" use="required"/>
       </extension>
      </complexContent>
     </complexType>
    </element>
   </sequence>
   <attribute name="referenceFrame" type="anyURI" use="required"/>
   <attribute name="localFrame" type="anyURI" use="optional"/>
  </extension>
 </complexContent>
</complexType>
```

This element is similar to the "*DataRecord*" element except that it is composed of a list of coordinates instead of fields. Each "*coordinate*" element is restricted to numerical component types (see "*AnyNumerical*" element group defined in clause §8.2.17) and inherits the reference frame from the "*Vector*" element. A Schematron pattern enforces that an "*axisID*" attribute is specified and that no "*referenceFrame*" attribute is used on each coordinate component (see Req 44 and Req 45).

The example below illustrates how to use the "*Vector*" element to encode geographic location:

```
<swe:Vector definition="urn:ogc:def:data:OGC::LocationVector"
            referenceFrame="urn:ogc:def:crs:EPSG:7.1:4326">
  <swe:coordinate name="lat">
    <swe:Quantity definition="urn:ogc:def:property:OGC::GeodeticLatitude" axisID="Lat">
      <gml:name>Latitude</gml:name>
      <swe:uom xlink:href="deg"/>
      <swe:value>45.36</swe:value>
    </swe:Quantity>
  </swe:coordinate>
  <swe:coordinate name="lon">
    <swe:Quantity definition="urn:ogc:def:property:OGC::Longitude" axisID="Long">
      <gml:name>Longitude</gml:name>
      <swe:uom code="deg"/>
      <swe:value>5.2</swe:value>
    </swe:Quantity>
  </swe:coordinate>
</swe:Vector>
```

This snippet indicates that the location coordinates are given in the EPSG 4326 (WGS 84 Lat/Lon) coordinate reference system. Note the use of a "*definition*" attribute to indicate what type of vector it is.

This definition is very important because the "*Vector*" element can be used to represent other vector quantities than location. For instance, the velocity vector of a spacecraft can be defined as show below:

```
<swe:Vector definition="urn:ogc:def:data:OGC::VelocityVector"
            referenceFrame="urn:ogc:def:crs:OGC::ECI">
  <swe:coordinate name="Vx">
    <swe:Quantity definition="urn:ogc:def:property:OGC::Velocity" axisID="X">
      <gml:name>Velocity X</gml:name>
      <swe:uom xlink:href="m/s"/>
    </swe:Quantity>
  </swe:coordinate>
  <swe:coordinate name="Vy">
    <swe:Quantity definition="urn:ogc:def:property:OGC::Velocity" axisID="Y">
      <gml:name>Velocity Y</gml:name>
      <swe:uom code="m/s"/>
    </swe:Quantity>
  </swe:coordinate>
  <swe:coordinate name="Vz">
    <swe:Quantity definition="urn:ogc:def:property:OGC::Velocity" axisID="Z">
      <gml:name>Velocity Z</gml:name>
      <swe:uom code="m/s"/>
    </swe:Quantity>
  </swe:coordinate>
</swe:Vector>
```

This instance is a data descriptor (i.e. there are no inline values) for an element of a dataset containing coordinates of a velocity vector. Each coordinate is projected along one axis of the Earth Centered Inertial (ECI) coordinate reference system and the unit of each vector component is the meter per second.

The "*localFrame*" attribute can also be used to identify the frame that the positioning information applies to:

```
<swe:Vector definition="urn:ogc:def:data:OGC::Quaternion"
            referenceFrame="urn:ogc:def:crs:OGC::ECI"
            localFrame="urn:ogc:id:CEOS:platform:SPOT5#PLATFORM_FRAME"/>
  <swe:coordinate name="qx">
    <swe:Quantity definition="urn:ogc:def:property:OGC::Coefficient" axisID="X">
      <swe:uom code="1"/>
      <swe:value>0.14</swe:value>
    </swe:Quantity>
  </swe:coordinate>
  <swe:coordinate name="qy">
    <swe:Quantity definition="urn:ogc:def:property:OGC::Coefficient" axisID="Y">
      <swe:uom code="1"/>
      <swe:value>0.22</swe:value>
    </swe:Quantity>
  </swe:coordinate>
  <swe:coordinate name="qz">
    <swe:Quantity definition="urn:ogc:def:property:OGC::Coefficient" axisID="Z">
      <swe:uom code="1"/>
      <swe:value>0.05</swe:value>
    </swe:Quantity>
  </swe:coordinate>
  <swe:coordinate name="qw">
    <swe:Quantity definition="urn:ogc:def:property:OGC::Coefficient" axisID="R">
      <swe:uom code="1"/>
      <swe:value>0.33</swe:value>
    </swe:Quantity>
  </swe:coordinate>
</swe:Vector>
```

This vector specifies the attitude of the local frame (i.e. attached to SPOT5 spacecraft) with respect to the ECI reference frame using a normalized quaternion. Note that quaternion coefficients are unit-less and normalized to 1.0 which is indicated by the UCUM code "1".

8.4 Requirements Class: Block Components Schema

XML Schema elements and types defined in the "*block_components.xsd*" schema implement all classes defined in the "Block Components" UML packages.

Req 69 **An implementation passing the "Block Components Schema" conformance test class shall first pass the "Aggregate Components Schema" and "Simple Encodings Schema" conformance test classes.**

Req 70 **A compliant XML instance shall be valid with respect to the grammar defined in the "block_components.xsd" XML schema as well as satisfy all Schematron patterns defined in "block_components.sch".**

8.4.1 DataArray Element

The "*DataArray*" element is the XML schema implementation of the "*DataArray*" UML class defined in clause §7.4.1. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="DataArray" type="swe:DataArrayType"
         substitutionGroup="swe:AbstractDataComponent"/>

<complexType name="DataArrayType">
  <complexContent>
    <extension base="swe:AbstractDataComponentType">
      <sequence>
        <element name="elementCount" type="swe:CountPropertyType"/>
        <element name="elementType">
          <complexType>
            <complexContent>
              <extension base="swe:AbstractDataComponentPropertyType">
                <attribute name="name" type="NCName" use="required"/>
              </extension>
            </complexContent>
          </complexType>
        </element>
        <element name="encoding" minOccurs="0">
          <complexType>
            <sequence>
              <element ref="swe:AbstractEncoding"/>
            </sequence>
          </complexType>
        </element>
        <element name="values" type="swe:EncodedValuesPropertyType" minOccurs="0"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

The size of the array is given by the "*elementCount*" property element which takes a "*Count*" data component. It can be used to construct both fixed size and variable size arrays. When the "*Count*" child element of the "*elementCount*" property includes an inline value, the array has a fixed size indicated by the value. When the "*Count*" child element has no inline value or when the "*elementCount*" has an "*xlink:href*" attribute, the array has a variable size.

The "*elementType*" property carries the definition of a single array element while the "*encoding*" and "*values*" properties allow including the array data inline as an efficient encoded data block. When present, this data block contains values for all elements of the array (the number of elements is given by the array size).

Req 71 **The encoded data block included either inline or by-reference in the "values" property of a "DataArray", "Matrix" or "DataStream" element shall be consistent with the definition of the element type, the element count and the encoding method.**

This first example shows how the "*DataArray*" element can be used to define a fixed size array of several measurement records and give their values inline as an encoded data block:

```
<swe:DataArray>
  <gml:description>Array of synchronous weather measurements</gml:description>
  <swe:elementCount>
    <swe:Count definition="urn:ogc:def:data:OGC::TemporalDimension">
      <swe:value>5</swe:value>
    </swe:Count>
  </swe:elementCount>
```

```
  <swe:elementType name="weather_measurement">
    <swe:DataRecord>
      <gml:name>Weather Data Record</gml:name>
      <swe:field name="time">
        <swe:Time definition="urn:ogc:def:property:OGC::SamplingTime">
          <gml:name>Sampling Time</gml:name>
          <swe:uom xlink:href="urn:ogc:def:unit:ISO:8601"/>
        </swe:Time>
      </swe:field>
      <swe:field name="temperature">
        <swe:Quantity definition="urn:ogc:def:property:OGC::AirTemperature">
          <gml:name>Air Temperature</gml:name>
          <swe:uom xlink:href="Cel"/>
        </swe:Quantity>
      </swe:field>
      <swe:field name="pressure">
        <swe:Quantity definition="urn:ogc:def:property:OGC::AtmosphericPressure">
          <gml:name>Atmospheric Pressure</gml:name>
          <swe:uom code="mbar"/>
        </swe:Quantity>
      </swe:field>
    </swe:DataRecord>
  </swe:elementType>
  <swe:encoding>
    <swe:TextEncoding blockSeparator="&#10;" tokenSeparator=","/>
  </swe:encoding>
  <swe:values>
    2009-02-10T10:42:56Z,25.4,1020
    2009-02-10T10:43:06Z,25.3,1021
    2009-02-10T10:44:16Z,25.3,1020
    2009-02-10T10:45:26Z,25.4,1022
    2009-02-10T10:46:36Z,25.4,1022
  </swe:values>
</swe:DataArray>
```

In this example, an array of 5 weather records is created. Each element of the array is a record of 3 values: the measurement sampling time, a temperature value and a pressure value. The array values are encoded as text tuples, and since the array size is 5, there are 5 tuples in the "*values*" element (in this case each line is a new tuple since the block separator is a 'new line' character. See clauses §8.5 and §8.6 for more information on "*TextEncoding*" and other encoding methods).

The next example illustrates how a dataset field containing variable length trajectory data can be defined:

```
<swe:DataArray definition="urn:ogc:def:data:OGC::Trajectory">
  <gml:description>Mobile Trajectory</gml:description>
  <swe:elementCount>
    <swe:Count definition="urn:ogc:def:data:OGC::SpatialDimension"/>
  </swe:elementCount>
  <swe:elementType name="point">
    <swe:Vector definition="urn:ogc:def:data:OGC:LocationVector"
                referenceFrame="urn:ogc:def:crs:EPSG:7.1:4326">
      <gml:name>Location Point</gml:name>
      <swe:coordinate name="lat">
        <swe:Quantity definition="urn:ogc:def:property:OGC::GeodeticLatitude" axisID="Lat">
          <gml:name>Latitude</gml:name>
          <swe:uom xlink:href="deg"/>
        </swe:Quantity>
      </swe:coordinate>
      <swe:coordinate name="lon">
        <swe:Quantity definition="urn:ogc:def:property:OGC::Longitude" axisID="Long">
          <gml:name>Longitude</gml:name>
          <swe:uom code="deg"/>
```

```
      </swe:Quantity>
    </swe:coordinate>
  </swe:Vector>
 </swe:elementType>
</swe:DataArray>
```

In this case, the "*elementCount*" value is not specified indicating that there will be an integer number specifying the array size in the data (corresponding to the "*Count*" representation) before the array values themselves. The array data will then contain several pairs of Lat/Lon values, each representing one array element. Note that neither the "*encoding*" or "*values*" properties are present in this example as the "*DataArray*" is used as a data descriptor. The "*definition*" attribute on the array gives useful information about its content.

Several "*DataArray*" elements can be nested to form multidimensional arrays. The following example shows how to fully define the structure of an image by using arrays:

```
<swe:DataArray definition="urn:ogc:def:data:OGC::Image">
  <swe:elementCount>
    <swe:Count definition="urn:ogc:def:data:OGC::ImageDimension">
      <swe:value>3000</swe:value>
    </swe:Count>
  </swe:elementCount>
  <swe:elementType name="row">
    <swe:DataArray definition="urn:ogc:def:data:OGC::Row">
      <swe:elementCount>
        <swe:Count definition="urn:ogc:def:data:OGC::ImageDimension">
          <swe:value>3000</swe:value>
        </swe:Count>
      </swe:elementCount>
      <swe:elementType name="pixel">
        <swe:DataRecord definition="urn:ogc:def:data:OGC::Pixel">
          <swe:field name="band1">
            <swe:Quantity definition="urn:ogc:def:property:OGC::Radiance">
              <gml:description>Radiance measured on band1</gml:description>
              <swe:uom code="W.m-2.Sr-1.um-1"/>
            </swe:Quantity>
          </swe:field>
          <swe:field name="band2">
            <swe:Quantity definition="urn:ogc:def:property:OGC::Radiance">
              <gml:description>Radiance measured on band2</gml:description>
              <swe:uom code="W.m-2.Sr-1.um-1"/>
            </swe:Quantity>
          </swe:field>
          <swe:field name="band3">
            <swe:Quantity definition="urn:ogc:def:property:OGC::Radiance">
              <gml:description>Radiance measured on band3</gml:description>
              <swe:uom code="W.m-2.Sr-1.um-1"/>
            </swe:Quantity>
          </swe:field>
        </swe:DataRecord>
      </swe:elementType>
    </swe:DataArray>
  </swe:elementType>
</swe:DataArray>
```

This example describes a 3000x3000 pixels image with three components. The image is organized by rows and the bands are interleaved by pixel. It is possible to describe different interleaving patterns by reversing the nesting order of the components.

8.4.2 Matrix Element

The "*Matrix*" element is the XML schema implementation of the "*Matrix*" UML class defined in clause §7.4.2. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="Matrix" type="swe:MatrixType" substitutionGroup="swe:AbstractDataComponent"/>

<complexType name="MatrixType">
  <complexContent>
    <extension base="swe:AbstractDataComponentType">
      <sequence>
        <element name="elementCount" type="swe:CountPropertyType"/>
        <element name="elementType">
          <complexType>
            <complexContent>
              <extension base="swe:AbstractDataComponentPropertyType">
                <attribute name="name" type="NCName" use="required"/>
              </extension>
            </complexContent>
          </complexType>
        </element>
        <element name="encoding" minOccurs="0">
          <complexType>
            <sequence>
              <element ref="swe:AbstractEncoding"/>
            </sequence>
          </complexType>
        </element>
        <element name="values" type="swe:EncodedValuesPropertyType" minOccurs="0"/>
      </sequence>
      <attribute name="referenceFrame" type="anyURI" use="required"/>
      <attribute name="localFrame" type="anyURI" use="optional"/>
    </extension>
  </complexContent>
</complexType>
```

The "*Matrix*" element is a special case of "*DataArray*" that adds "*referenceFrame*" and "*localFrame*" attributes for expressing the array components in a well defined reference frame. As opposed to the "*Vector*" component, the axis order is implied in a matrix because it is difficult to assign a frame axis to each individual element of an N-dimensional array. The array index in each dimension is thus used as the axis index in the ordered list provided by the reference frame.

The following example shows how to encode a rotation matrix:

```
<swe:Matrix definition="urn:ogc:def:data:OGC::RotationMatrix"
            referenceFrame="urn:ogc:def:crs:OGC::ECI">
  <swe:elementCount>
    <swe:Count definition="urn:ogc:def:data:OGC::SpatialDimension">
      <swe:value>3</swe:value>
    </swe:Count>
  </swe:elementCount>
  <swe:elementType name="row">
    <swe:DataArray>
      <swe:elementCount>
        <swe:Count definition="urn:ogc:def:data:OGC::SpatialDimension">
          <swe:value>3</swe:value>
        </swe:Count>
      </swe:elementCount>
      <swe:elementType name="coef">
        <swe:Quantity definition="urn:ogc:def:property:OGC::Coefficient">
          <swe:uom code="1"/>
```

```
      </swe:Quantity>
     </swe:elementType>
    </swe:DataArray>
  </swe:elementType>
  <swe:encoding>
   <swe:TextEncoding blockSeparator=" " tokenSeparator=","/>
  </swe:encoding>
  <swe:values>0.36,0.48,-0.8 -0.8,0.6,0 0.48,0.64,0.6</swe:values>
</swe:Matrix>
```

This example defines a 3x3 rotation matrix whose elements are expressed in the ECI coordinate reference system. It corresponds to the following matrix:

$$\begin{bmatrix} 0.36 & 0.48 & -0.8 \\ -0.8 & 0.60 & 0 \\ 0.48 & 0.64 & 0.60 \end{bmatrix}$$

Axes are assumed to be in the same order as specified in the reference frame definition, that is to say: 1st row/column = X, 2nd row/column = Y, 3rd row/column = Z

As with the "*Vector*" element, the "*localFrame*" attribute can be used to identify the frame of whose positioning information is specified by the matrix

8.4.3 DataStream Element

The "*DataStream*" element is the XML schema implementation of the "*DataStream*" UML class defined in clause §7.4.3. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="DataStream" type="swe:DataStreamType"
         substitutionGroup="swe:AbstractSWEIdentifiable">

<complexType name="DataStreamType">
  <complexContent>
    <extension base="swe:AbstractSWEIdentifiableType">
      <sequence>
        <element name="elementCount" minOccurs="0">
          <complexType>
            <sequence>
              <element ref="swe:Count"/>
            </sequence>
          </complexType>
        </element>
        <element name="elementType">
          <complexType>
            <complexContent>
              <extension base="swe:AbstractDataComponentPropertyType">
                <attribute name="name" type="NCName" use="required"/>
              </extension>
            </complexContent>
          </complexType>
        </element>
        <element name="encoding">
          <complexType>
            <sequence>
              <element ref="swe:AbstractEncoding"/>
            </sequence>
          </complexType>
        </element>
        <element name="values" type="swe:EncodedValuesPropertyType"/>
      </sequence>
```

```
    </extension>
  </complexContent>
</complexType>
```

This element is used to describe a data stream as a list of elements whose type is given by the element type. It is similar to a "*DataArray*" but the "*elementCount*" property is optional as the total number of elements composing the stream does not have to be specified. This is useful in particular to describe never-ending streams such as the ones used for delivering real time sensor data. Additionally, the "*DataStream*" element is not a data component and thus cannot be nested into other aggregates. It can only serve as a root object to represent the data stream as a whole.

The next example shows how it is used to describe a real time stream of aircraft navigaton data:

```
<swe:DataStream>
  <gml:name>Aircraft Navigation</gml:name>
  <swe:elementType name="navData">
    <swe:DataRecord>
      <swe:field name="time">
        <swe:Time definition="urn:ogc:def:property:OGC:SamplingTime"
                  referenceFrame="urn:ogc:def:crs:OGC::GPS"
                  referenceTime="1970-01-01T00:00:00Z">
          <swe:uom code="s"/>
        </swe:Time>
      </swe:field>
      <swe:field name="location">
        <swe:Vector definition="urn:ogc:property:OGC::LocationVector"
                    referenceFrame="urn:ogc:def:crs:EPSG:6.7:4979">
          <swe:coordinate name="lat">
            <swe:Quantity definition="urn:ogc:def:property:OGC:GeodeticLatitude" axisID="Lat">
              <swe:uom code="deg"/>
            </swe:Quantity>
          </swe:coordinate>
          <swe:coordinate name="lon">
            <swe:Quantity definition="urn:ogc:def:property:OGC:Longitude" axisID="Long">
              <swe:uom code="deg"/>
            </swe:Quantity>
          </swe:coordinate>
          <swe:coordinate name="alt">
            <swe:Quantity definition="urn:ogc:def:property:OGC:EllipsoidalHeight" axisID="h">
              <swe:uom code="m"/>
            </swe:Quantity>
          </swe:coordinate>
        </swe:Vector>
      </swe:field>
      <swe:field name="attitude">
        <swe:Vector definition="urn:ogc:property:OGC::EulerAngles"
                    referenceFrame="urn:ogc:def:crs:OGC::ENU">
          <swe:coordinate name="heading">
            <swe:Quantity definition="urn:ogc:def:property:OGC:TrueHeading" axisID="Z">
              <swe:uom code="deg"/>
            </swe:Quantity>
          </swe:coordinate>
          <swe:coordinate name="pitch">
            <swe:Quantity definition="urn:ogc:def:property:OGC:PitchAngle" axisID="X">
              <swe:uom code="deg"/>
            </swe:Quantity>
          </swe:coordinate>
          <swe:coordinate name="roll">
            <swe:Quantity definition="urn:ogc:def:property:OGC:RollAngle" axisID="Y">
              <swe:uom code="deg"/>
            </swe:Quantity>
          </swe:coordinate>
```

```
      </swe:Vector>
    </swe:field>
  </swe:DataRecord>
 </swe:elementType>
 <swe:encoding>
  <swe:TextEncoding tokenSeparator="," blockSeparator="&#10;" decimalSeparator="."/>
 </swe:encoding>
 <swe:values xlink:href="rtp://myserver:4563/navData"/>
</swe:DataStream>
```

This example defines a stream of homogeneous records, each of which is composed of a time stamp, 3D aircraft location expressed in the EPSG 4979 (WGS 84 Lat/Lon/Alt) coordinate reference system, and aircraft attitude expressed relative to the local ENU (East-North-Up) coordinate frame. The actual data values would then be sent via the RTP connection in the following text (CSV) format:

```
1257691405,41.55,13.61,325,90.5,1.2,1.1
1257691410,41.55,13.62,335,90.4,1.3,0.5
1257691415,41.55,13.63,345,90.5,1.3,0.1
1257691420,41.55,13.64,355,90.4,1.2,-1.1
1257691425,41.55,13.65,365,90.5,1.2,-0.5
...
```

Note that the "*encoding*" and "*values*" properties are mandatory on the "*DataStream*" element, indicating that it can only be used to describe the dataset as a whole, along with its encoding method. The "values" element is usually used to provide a reference to the actual data stream (i.e. the values).

Note that streams of heterogeneous records can also be described by using a "*DataChoice*" as the element type. This is shown below:

8.5 Requirements Class: Simple Encodings Schema

XML Schema elements and types defined in the "*simple_encodings.xsd*" schema implement all classes defined in the "Simple Encodings" UML packages.

Req 72 **An implementation passing the "Simple Encodings Schema" conformance test class shall first pass the "Basic Types and Simple Components Schemas" conformance test class.**

Req 73 **A compliant XML instance shall be valid with respect to the grammar defined in the "simple_encodings.xsd" XML schema as well as satisfy all Schematron patterns defined in "simple_encodings.sch".**

This requirement class defines a set of core encodings that have been chosen to cover the needs of simple applications that need to encode data as efficient data blocks. The "*TextEncoding*" method allows encodings datasets in a human readable textual format,

while the "*XMLEncoding*" method allows encoding data values light weight XML tagged values.

Note: It is not the intent of this standard to support legacy formats by simply wrapping them with an XML description. Implementations seeking comformance to this requirement class will most often have to re-encode existing data by following the encoding rules described in this clause. However the encoding model has been designed and tested so that re-encoding can be done very efficiently on-the-fly without requiring the pre-processing of large amounts of existing data.

8.5.1 General Encoding Rules

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.

8.5.1.1 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.

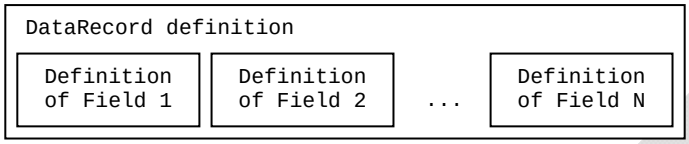
8.5.1.2 Rules for Range Components

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.

8.5.1.3 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 XML description and including the resulting values sequentially in the stream.

The definition of a "*DataRecord*" ("*Vector*") structure composed of N fields (coordinates) can be represented in the following way:

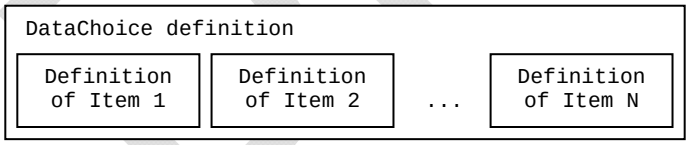


The data block corresponding to such a structure would sequentially include all values for field 1, then all values for field 2, etc. until the last field is reached. Each field may consist of a single value if it is a scalar but may also consist of multiple values if it is itself an aggregate or a range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

Req 74 **"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.**

8.5.1.4 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.



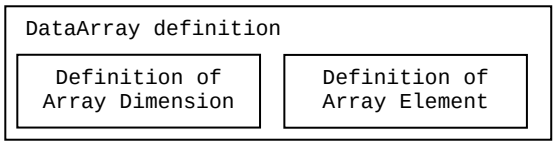
The data block corresponding to such a structure would then sequentially include the item identifier (i.e. the choice value) and then the value(s) for the selected item. The item may consist of a single value if it is a scalar or multiple values if it is itself an aggregate or a range component.

Req 75 **Encoded values for the selected item of a "DataChoice" shall be provided along with information that unambiguously identifies the selected item.**

8.5.1.5 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.

Req 76 **"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).**

Req 77 **Encoded data for a variable size "DataArray" shall include a number specifying the array size whatever the encoding method used.**

8.5.2 AbstractEncoding Element

The "*AbstractEncoding*" element is the XML schema implementation of the "*AbstractEncoding*" UML class defined in clause §0. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="AbstractEncoding" type="swe:AbstractEncodingType" abstract="true"
         substitutionGroup="swe:AbstractSWE"/>

<complexType name="AbstractEncodingType" abstract="true">
  <complexContent>
    <extension base="swe:AbstractSWEType"/>
  </complexContent>
</complexType>
```

This element serves as the substitution group for all XML elements that describe encoding methods in this standard or in extensions of this standard.

8.5.3 TextEncoding Element

The "*TextEncoding*" element is the XML schema implementation of the "*TextEncoding*" UML class defined in clause §7.5.1. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="TextEncoding" type="swe:TextEncodingType"
         substitutionGroup="swe:AbstractEncoding"/>

<complexType name="TextEncodingType">
  <complexContent>
    <extension base="swe:AbstractEncodingType">
      <attribute name="collapseWhiteSpaces" type="boolean" use="optional" default="true"/>
      <attribute name="decimalSeparator" type="string" use="optional" default="."/>
      <attribute name="tokenSeparator" type="string" use="required"/>
      <attribute name="blockSeparator" type="string" use="required"/>
    </extension>
  </complexContent>
</complexType>
```

This element is used to specify encoding of data values in a "Delimiter Separated Values" format (a generalization of CSV) that is parameterized by its 4 XML attributes. The following example shows a set of commonly used parameters:

```
<swe:TextEncoding tokenSeparator="," blockSeparator=" "/>
```

The "*decimalSeparator*" and "*collapseWhiteSpaces*" attributes have been omitted to indicate that their default values should be used. This leads to a data stream where individual tokens are separated by commas (i.e. the ',' character), while complete blocks are separated by spaces. It can for example be used to encode coordinate tuples of "lat,lon,lat" values in a very readable manner, such as:

```
25.41,10.23,320 25.43,10.23,300 25.39,11.51,310
```

Special characters such as carriage returns (CR) or line feeds (LF) can be used as block or token separators by using XML entities. For example new line characters are often used as block separators to cleanly separate blocks of values on successive lines:

```
<swe:TextEncoding tokenSeparator=";" blockSeparator="&#10;"/>
```

This corresponds to the following data block format:

```
25.41;10.23;320↵
25.43;10.23;300↵
25.39;11.51;310↵
```

This is compatible with the CSV format and is often used for compatibility with other software.

More than one character can be used as a separator in order to avoid conflicts with characters within the data values themselves. The following example shows this type of usage:

```
<swe:TextEncoding tokenSeparator="||" blockSeparator="@@&#10;"/>
```

This specifies the following data block format:

```
25.41||text with spaces||text with
carriage return||{special_chars}@@
25.42||text with spaces||text with
carriage return||{special_chars}@@
25.43||text with spaces||text with
carriage return||{special_chars}@@
```

A compliant parser can successfully parse such a data block because only sequences of characters that perfectly match the separator definition indicate the end of a token or block. Implementations are required to support sequences of characters of any length as separators but small ones (i.e. 1 to 3 characters) are more efficient and should be used whenever possible.

Both "*tokenSeparator*" and "*blockSeparator*" can have the same value but this is not recommended as it makes the data block less readable and makes block-level resynchronizations impossible in error prone transmissions.

8.5.4 Text Encoding Rules

The "*TextEncoding*" method encodes field values (especially numbers) by their text representation rather than their binary representation. Special characters provide a way to separate successive values. The EBNF syntax defined in ISO 14977 is used to formalize the encoding rules, and thus all EBNF snippet provided in this section are normative.

Req 78 Compliant encoding/decoding software shall implement the "TextEncoding" method by following the EBNF grammar defined in this clause.

8.5.4.1 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.

Req 79 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 and 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];
```

White spaces around separators are in fact only allowed when the "collapseWhiteSpaces" attribute is set to 'true' (which is the default).

8.5.4.2 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;
```

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.

8.5.4.3 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;
```

8.5.4.4 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”, any-field-value) | “N”;

field-value = mandatory-field-value | optional-field-value

record-values = field-value, <field-count> * (token-separator, field-value);
```

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.

Req 80 **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);
```

The following example shows how elements of an array defined as a “*DataRecord*” are encoded with the text method:

```
<swe:DataArray definition="urn:ogc:def:data:OGC::ErrorCurve">
  <gml:description>Measurement error vs. temperature</gml:description>
  <swe:elementCount>
    <swe:Count definition="urn:ogc:def:data:OGC::PhysicalDimension">
      <swe:value>5</swe:value>
    </swe:Count>
  </swe:elementCount>
  <swe:elementType name="point">
    <swe:DataRecord>
      <gml:name>Error vs. Temperature</gml:name>
      <swe:field name="temp">
        <swe:Quantity definition="urn:ogc:def:property:OGC::Temperature">
          <gml:name>Temperature</gml:name>
          <swe:uom code="Cel"/>
        </swe:Quantity>
      </swe:field>
      <swe:field name="error">
        <swe:Quantity definition="urn:ogc:def:property:OGC::RelativeError">
          <gml:name>Relative Error</gml:name>
          <swe:uom code="%"/>
        </swe:Quantity>
      </swe:field>
    </swe:DataRecord>
  </swe:elementType>
  <swe:encoding>
    <swe:TextEncoding blockSeparator=" " tokenSeparator=","/>
  </swe:encoding>
  <swe:values>0,5 10,2 50,2 80,5 100,15</swe:values>
</swe:DataArray>
```

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.

The following example shows the resulting encoded block when some of the fields are optional:

```
<swe:DataStream>
  <gml:name>Aircraft Navigation</gml:name>
  <swe:elementType name="navData">
    <swe:DataRecord>
      <swe:field name="time">
        <swe:Time definition="urn:ogc:def:property:OGC::SamplingTime"
                  referenceFrame="urn:ogc:def:crs:OGC::GPS">
          <swe:uom xlink:href="urn:ogc:def:unit:ISO:8601"/>
        </swe:Time>
      </swe:field>
      <swe:field name="speed">
        <swe:Quantity definition="urn:ogc:def:property:OGC::AirSpeed">
          <swe:uom code="m/s"/>
        </swe:Quantity >
      </swe:field>
      <swe:field name="location">
        <swe:Vector optional="true" referenceFrame="urn:ogc:def:crs:EPSG:6.7:4979">
          <swe:coordinate name="lat">
            <swe:Quantity definition="urn:ogc:def:property:OGC::GeodeticLatitude" axisID="Lat">
              <swe:uom code="deg"/>
            </swe:Quantity>
          </swe:coordinate>
          <swe:coordinate name="lon">
            <swe:Quantity definition="urn:ogc:def:property:OGC::Longitude" axisID="Long">
              <swe:uom code="deg"/>
            </swe:Quantity>
          </swe:coordinate>
          <swe:coordinate name="alt">
            <swe:Quantity definition="urn:ogc:def:property:OGC::EllipsoidalHeight" axisID="h">
              <swe:uom code="m"/>
            </swe:Quantity>
          </swe:coordinate>
        </swe:Vector>
      </swe:field>
    </swe:DataRecord>
  </swe:elementType>
  <swe:encoding>
    <swe:TextEncoding blockSeparator="&#10;" tokenSeparator=","/>
  </swe:encoding>
  <swe:values>
    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
  </swe:values>
</swe:DataStream>
```

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.

8.5.4.5 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;
```

Req 81 **The selected-item-name token shall correspond to the value of the "name" attribute of the "item" property element that represents the selected item.**

This is illustrated by the following example:

```
<swe:DataStream>
  <swe:elementType name="message">
    <swe:DataChoice>
      <swe:choiceValue>
        <swe:Category definition="urn:ogc:def:data:OGC::MessageType"/>
      </swe:choiceValue>
      <swe:item name="TEMP">
        <swe:DataRecord>
          <gml:name>Temperature Measurement</gml:name>
          <swe:field name="time">
            <swe:Time definition="urn:ogc:def:property:OGC::SamplingTime"
                      referenceFrame="urn:ogc:def:crs:OGC::GPS">
              <swe:uom xlink:href="urn:ogc:def:unit:ISO:8601"/>
            </swe:Time>
          </swe:field>
          <swe:field name="temp">
            <swe:Quantity definition="urn:ogc:def:property:OGC::Temperature">
              <swe:uom code="Cel"/>
            </swe:Quantity>
          </swe:field>
        </swe:DataRecord>
      </swe:item>
      <swe:item name="WIND">
        <swe:DataRecord>
          <gml:name>Wind Measurement</gml:name>
          <swe:field name="time">
            <swe:Time definition="urn:ogc:def:property:OGC::SamplingTime"
                      referenceFrame="urn:ogc:def:crs:OGC::GPS">
              <swe:uom xlink:href="urn:ogc:def:unit:ISO:8601"/>
            </swe:Time>
          </swe:field>
          <swe:field name="wind_speed">
            <swe:Quantity definition="urn:ogc:def:property:OGC::WindSpeed">
              <swe:uom code="km/h"/>
            </swe:Quantity>
          </swe:field>
          <swe:field name="wind_dir">
            <swe:Quantity definition="urn:ogc:def:property:OGC::WindDirectionToNorth">
              <swe:uom code="deg"/>
            </swe:Quantity>
          </swe:field>
        </swe:DataRecord>
      </swe:item>
    </swe:DataChoice>
  </swe:elementType>
  <swe:encoding>
    <swe:TextEncoding blockSeparator="&#10;" tokenSeparator=","/>
```

```
  </swe:encoding>
  <swe:values>
    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
  </swe:values>
</swe:DataStream>
```

This data stream interleaves different types of messages separated by the block separator character. The element type is a "*DataChoice*" which means that each block is composed of the item name 'TEMP' or 'WIND' (highlighted in yellow), followed by values of the item. This example also demonstrates that items of a choice can be of different types and length.

8.5.4.6 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*" or "*Matrix*" is nested in another block component (i.e. "*DataArray*", "*Matrix*" or "*DataStream*"), its elements are separated by token separators, otherwise its elements are separated by block 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;

array-values = fixed-size-array-values | variable-size-array-values;
```

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);
```

The following example illustrates how values of a fixed size 3x3 stress matrix can be text encoded:

```
<swe:Matrix definition="urn:ogc:def:data:OGC::StressMatrix" referenceFrame="#SAMPLE_FRAME">
  <swe:elementCount>
    <swe:Count definition="urn:ogc:def:data:OGC::SpatialDimension">
      <swe:value>3</swe:value>
    </swe:Count>
  </swe:elementCount>
  <swe:elementType name="row">
    <swe:DataArray definition="urn:ogc:def:data:OGC::Row">
```

```
      <swe:elementCount>
        <swe:Count definition="urn:ogc:def:data:OGC::SpatialDimension">
          <swe:value>3</swe:value>
        </swe:Count>
      </swe:elementCount>
      <swe:elementType name="coef">
        <swe:Quantity definition="urn:ogc:def:property:OGC::Stress">
          <swe:uom code="MPa"/>
        </swe:Quantity>
      </swe:elementType>
    </swe:DataArray>
  </swe:elementType>
  <swe:encoding>
    <swe:TextEncoding blockSeparator=" " tokenSeparator=","/>
  </swe:encoding>
  <swe:values>0.36,0.48,-0.8 -0.8,0.6,0.0 0.48,0.64,0.6</swe:values>
</swe:Matrix>
```

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.

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);
```

The following example shows how SWE Common can be used to encode a series of irregular length profiles by using a variable size array:

```
<swe:DataStream>
  <swe:elementType name="profileData">
    <swe:DataRecord>
      <swe:field name="time">
        <swe:Time definition="urn:ogc:def:property:OGC:SamplingTime">
          <gml:name>Sampling Time</gml:name>
          <swe:uom xlink:href="urn:ogc:def:unit:ISO:8601"/>
        </swe:Time>
      </swe:field>
      <swe:field name="profilePoints">
        <swe:DataArray>
          <swe:elementCount>
            <swe:Count definition="urn:ogc:def:data:OGC::SpatialDimension"/>
          </swe:elementCount>
          <swe:elementType name="point">
            <swe:DataRecord>
              <swe:field name="depth">
                <swe:Quantity definition="urn:ogc:def:property:OGC:EllipsoidalHeight"
                              referenceFrame="urn:ogc:def:crs:EPSG:7.1:4979" axisID="Z">
                  <gml:name>Sampling Point Vertical Location</gml:name>
                  <swe:uom code="m"/>
                </swe:Quantity>
              </swe:field>
              <swe:field name="salinity">
                <swe:Time definition="http://mmisw.org/ont/cf/parameter#sea_water_salinity">
                  <gml:name>Salinity</gml:name>
                  <swe:uom code="[ppth]"/>
                </swe:Time>
              </swe:field>
```

```
          </swe:DataRecord>
        </swe:elementType>
      </swe:DataArray>
    </swe:field>
  </swe:DataRecord>
 </swe:elementType>
 <swe:encoding>
  <swe:TextEncoding blockSeparator="@@&#10;" tokenSeparator=","/>
 </swe:encoding>
 <swe:values>
  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@@
  2005-05-17T00:45:22Z,6,0,45,10,20,20,30,30,35,40,40,50,45@@
 </swe:values>
</swe:DataStream>
```

The example shows data for 4 profiles with a variable number of measurements along the vertical dimension. The number of measurements is indicated by a number in the data block (highlighted in yellow) that is inserted before the measurements themelves. Since the array is itself the element of a "*DataStream*", elements of the array are separated by token separators.

8.5.4.7 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);
```

Examples of "*DataStream*" with "*TextEncoding*" have already been given in previous sections.

8.5.5 XMLEncoding Element

The "*XMLEncoding*" element is the XML schema implementation of the "*XMLEncoding*" UML class defined in clause §7.5.2. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="XMLEncoding" type="swe:XMLEncodingType"
         substitutionGroup="swe:AbstractEncoding"/>

<complexType name="XMLEncodingType">
  <complexContent>
    <extension base="swe:AbstractEncodingType">
      <attribute name="defaultNamespace" type="anyURI" use="optional"/>
    </extension>
  </complexContent>
</complexType>
```

The XML Block encoding method is used when data values are to be encoded as light weight XML elements. The way the XML elements are named and structured are tied to the data structure specified using a hierarchy of data components.

This encoding method defines only one parameter: the "*defaultNamespace*" attribute indicates the namespace URI to use for XML elements in the data stream. There is no restriction on what this namespace can be but it is recommended to use a different namespace than the one used by this standard and its dependencies. Ideally this namespace should be unique to the dataset whose values are encoded using this method. The declaration of such a namespace is shown below:

```
<swe:XMLEncoding defaultNamespace="http://www.epa.gov/swe_datasets/023451"/>
```

8.5.6 XML Encoding rules

The "*XMLEncoding*" method encodes field values (especially numbers) by their text representation according to XML schema data types and wraps them with XML tags carrying the name of the corresponding field. The hierarchy of components is fully represented by XML tags, which makes this encoding more verbose but also well suited for processing and validation with existing XML frameworks.

8.5.6.1 XML element names

Each data component of the tree is represented by an XML tag whose element name corresponds to the "*name*" attribute of the soft-typed property containing the component description. This property is most often "*field*", "*coordinate*" or "*elementType*", depending on the parent aggregate.

Req 82 All data components shall be XML encoded with an element whose local name shall correspond to the "name" attribute of the soft-typed property containing the data component.

Scalar components are thus encoded by an XML element with a text value whereas aggregate components are encoded by an XML element itself containing sub-elements representing the aggregate's children.

8.5.6.2 Rules for Scalar Components

Scalar components are encoded by an XML element whose name corresponds to the soft-typed property containing the component.

Req 83 Scalar components values shall be XML encoded with a single element containing the value as its text content and no other child element.

Examples of scalar values encoded in XML are given below:

```
<ns:status>OFF</ns:status>

<ns:time>2009-01-02T23:45:12Z</ns:time>

<ns:temp>25.5</ns:temp>
```

NIL values are included as the text content of the XML element representing scalar components, in the same way regular scalar values would be included.

8.5.6.3 Rules for Range Components

Range components are encoded by an XML element whose name corresponds to the soft-typed property containing the component which itself contain two min/max elements carrying the range extreme values.

Req 84 **Range components values shall be XML encoded with an element containing two sub-elements with local names "min" and "max" which respectively contain the lower and upper values of the range as their text content.**

Let us consider the example of "*TimeRange*" below:

```
<swe:field name="SurveyPeriod">
  <swe:TimeRange definition="urn:ogc:def:property:CEOS:eop:SurveyPeriod" referenceFrame="…">
    <swe:uom xlink:href="urn:ogc:def:unit:ISO:8601"/>
  </swe:TimeRange>
</swe:field>
```

Following Req 84, this component values are encoded as XML as shown below:

```
<ns:SurveyPeriod>
  <ns:min>2009-01-02T23:45:12Z</ns:min>
  <ns:max>2009-01-02T23:45:12Z</ns:max>
</ns:SurveyPeriod>
```

8.5.6.4 Rules for DataRecord and Vector

Aggregate components are encoded by using a parent element with the proper local name as enforced by Req 82 to which elements for sub-components are appended (recursively). Elements normally corresponding to record fields marked as optional can be completely omitted since parsers can use element names to unambiguously know the ones that are missing.

Req 85 **"DataRecord" values shall be XML encoded with an element which contains one sub-element for each "field" that is not omitted.**

Req 86 "Vector" values shall be XML encoded with an element which contains one sub-element for each "coordinate" of the aggregate.

The curve data example introduced in section §8.5.4.4 would be encoded in XML as shown below:

```
<swe:encoding>
  <swe:XMLEncoding defaultNamespace="http://www.myorg.com/datasets/id"/>
</swe:encoding>

<swe:values xmlns:ns="http://www.myorg.com/datasets/id">
  <ns:point>
    <ns:temp>0</ns:temp>
    <ns:error>5</ns:error>
  </ns:point>
  <ns:point>
    <ns:temp>10</ns:temp>
    <ns:error>2</ns:error>
  </ns:point>
  <ns:point>
    <ns:temp>50</ns:temp>
    <ns:error>2</ns:error>
  </ns:point>
  <ns:point>
    <ns:temp>80</ns:temp>
    <ns:error>5</ns:error>
  </ns:point>
  <ns:point>
    <ns:temp>100</ns:temp>
    <ns:error>15</ns:error>
  </ns:point>
</swe:values>
```

In this example, the array element type is called 'point' and is defined as a "*DataRecord*" that contains two scalar fields called 'temp' and 'error'. These soft-typed property names are thus used as the element local names of encoded values.

The following example shows how the second sample dataset from section §8.5.4.4 that makes use of optional fields is encoded with the "*XMLEncoding*" method:

```
<swe:encoding>
  <swe:XMLEncoding defaultNamespace="urn:myorg:dataset:X156822"/>
</swe:encoding>

<swe:values xmlns:ns="urn:myorg:dataset:X156822">
  <ns:navData>
    <ns:time>2007-10-23T15:46:12Z</ns:time>
    <ns:speed>15.3</ns:speed>
    <ns:location>
      <ns:lat>45.3</ns:lat>
      <ns:lon>-90.5</ns:lon>
      <ns:alt>311</ns:alt>
    </ns:location>
  </ns:navData>
  <ns:navData>
    <ns:time>2007-10-23T15:46:22Z</ns:time>
    <ns:speed>25.3</ns:speed>
  </ns:navData>
  <ns:navData>
    <ns:time>2007-10-23T15:46:32Z</ns:time>
    <ns:speed>20.6</ns:speed>
    <ns:location>
```

```
      <ns:lat>45.3</ns:lat>
      <ns:lon>-90.6</ns:lon>
      <ns:alt>312</ns:alt>
    </ns:location>
  </ns:navData>
</swe:values>
```

The missing 'location' value in the second stream element has been completely omitted.

8.5.6.5 Rules for DataArray, Matrix and DataStream

Block components are slightly different because they can either include the encoded data block in their "*values*" element or be nested into another block component which includes the encoded data block.

In the case of all "*DataStream*" instances or when the "*DataArray*" or "*Matrix*" includes its own encoded values, only the array elements are actually encoded within the "*values*" XML element. The two previous examples of this section illustrate this case.

Req 87 **Values of each element of a "DataArray", "Matrix" or "DataStream" shall be encapsulated in a separate XML element whose local name shall be the value of the "name" attribute of its "elementType" element.**

When a "*DataArray*" or "*Matrix*" is nested in a parent block component (and thus does not encapsulate encoded values itself), array elements are encoded as defined above but are also wrapped in an element carrying the array name.

Req 88 **All elements of each nested "DataArray" and "Matrix" shall be encapsulated in a parent element as specified in Req 82 and this element shall also have an "elementCount" attribute that specifies the array size.**

The following example builds on the sample profile series dataset introduced in clause §8.5.4.6 and shows how the same values could be encoded with the "*XMLEncoding*" method:

```
<swe:encoding>
  <swe:XMLEncoding defaultNamespace="urn:myorg:dataset:PS3658"/>
</swe:encoding>

<swe:values xmlns:ns="urn:myorg:dataset:PS3658">
  <ns:profileData>
    <ns:time>2005-05-16T21:47:12Z</ns:time>
    <ns:profilePoints elementCount="5">
      <ns:point>
        <ns:depth>0</ns:depth>
        <ns:salinity>45</ns:salinity>
      </ns:point>
      <ns:point>
        <ns:depth>10</ns:depth>
        <ns:salinity>20</ns:salinity>
      </ns:point>
      <ns:point>
```

```
      <ns:depth>20</ns:depth>
      <ns:salinity>30</ns:salinity>
    </ns:point>
    <ns:profilePoint>
      <ns:depth>30</ns:depth>
      <ns:salinity>35</ns:salinity>
    </ns:point>
    <ns:profilePoint>
      <ns:depth>40</ns:depth>
      <ns:salinity>40</ns:salinity>
    </ns:point>
  </ns:profilePoints>
</ns:profileData>
<ns:profileData>
  <ns:time>2005-05-16T22:43:05Z</ns:time>
  <ns:profilePoints elementCount="4">
    <ns:point>
      <ns:depth>0</ns:depth>
      <ns:salinity>45</ns:salinity>
    </ns:point>
    <ns:point>
      <ns:depth>10</ns:depth>
      <ns:salinity>20</ns:salinity>
    </ns:point>
    <ns:point>
      <ns:depth>20</ns:depth>
      <ns:salinity>30</ns:salinity>
    </ns:point>
    <ns:point>
      <ns:depth>30</ns:depth>
      <ns:salinity>35</ns:salinity>
    </ns:point>
  </ns:profilePoints>
</ns:profileData>
    ...
</swe:values>
```

This example shows how the array size is specified on the 'profilePoints' element corresponding to each nested array, and how element local names correspond to the "name" attributes of each component's parent property.

8.6 Requirements Class: Advanced Encodings Schema

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

Req 89 **An implementation passing the "Advanced Encodings Schema" conformance test class shall first pass the "Simple Encodings Schema" conformance test class.**

Req 90 **A compliant XML instance shall be valid with respect to the grammar defined in the "advanced_encodings.xsd" XML schema as well as satisfy all Schematron patterns defined in "advanced_encodings.sch".**

Note: The raw binary encoding option is not usable within an XML document since it makes use of characters not allowed in XML. Raw binary data can only be provided

separately from the XML document and eventually referenced via an xlink. If there is a requirement for binary data to be included as text content of an XML element, the base64 option should be used.

8.6.1 BinaryEncoding Element

The "*BinaryEncoding*" element is the XML schema implementation of the "*BinaryEncoding*" UML class defined in clause §7.6.1. The schema snippet for this element and its corresponding complex type is shown below:

```
<element name="BinaryEncoding" type="swe:BinaryEncodingType"
         substitutionGroup="swe:AbstractEncoding"/>

<complexType name="BinaryEncodingType">
  <complexContent>
    <extension base="swe:AbstractEncodingType">
      <sequence>
        <element name="member" maxOccurs="unbounded">
          <complexType>
            <sequence>
              <group ref="swe:ComponentOrBlock"/>
            </sequence>
          </complexType>
        </element>
      </sequence>
      <attribute name="byteOrder" type="swe:ByteOrderType" use="required"/>
      <attribute name="byteEncoding" type="swe:ByteEncodingType" use="required"/>
      <attribute name="byteLength" type="integer" use="optional"/>
    </extension>
  </complexContent>
</complexType>
```

This element makes use of two simple types implementing the "*ByteEncoding*" and "*ByteOrder*" UML enumerations respectively:

```
<simpleType name="ByteEncodingType">
  <restriction base="string">
    <enumeration value="base64"/>
    <enumeration value="raw"/>
  </restriction>
</simpleType>

<simpleType name="ByteOrderType">
  <restriction base="string">
    <enumeration value="bigEndian"/>
    <enumeration value="littleEndian"/>
  </restriction>
</simpleType>
```

The member property allow a choice of "*Component*" or "*Block*" sub-elements as defined below:

```
<group name="ComponentOrBlock">
  <choice>
    <element ref="swe:Component"/>
    <element ref="swe:Block"/>
  </choice>
</group>
```

The "*Component*" element implements the UML class with the same name. It is used to specify encoding parameters of scalar components and is shown below:

```
<element name="Component" type="swe:ComponentType" substitutionGroup="gml:AbstractObject"/>

<complexType name="ComponentType">
  <sequence>
    <element ref="swe:ComponentExtensibilityPoint" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="encryption" type="anyURI" use="optional"/>
  <attribute name="significantBits" type="integer" use="optional"/>
  <attribute name="bitLength" type="integer" use="optional"/>
  <attribute name="byteLength" type="integer" use="optional"/>
  <attribute name="dataType" type="anyURI" use="required"/>
  <attribute name="ref" type="string" use="required"/>
  <attribute ref="gml:id" use="optional"/>
</complexType>
```

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 and is shown below:

```
<element name="Block" type="swe:BlockType" substitutionGroup="gml:AbstractObject"/>

<complexType name="BlockType">
  <sequence>
    <element ref="swe:BlockExtensibilityPoint" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="compression" type="anyURI" use="optional"/>
  <attribute name="encryption" type="anyURI" use="optional"/>
  <attribute name="paddingBytes-after" type="integer" use="optional"/>
  <attribute name="paddingBytes-before" type="integer" use="optional"/>
  <attribute name="byteLength" type="integer" use="optional"/>
  <attribute name="ref" type="string" use="required"/>
  <attribute ref="gml:id" use="optional"/>
</complexType>
```

These elements allow for the detailed specification of the encoding parameters associated to components of the data description tree as discussed in clause §7.6.1. 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.

Req 91 **The "ref" attribute of the "Component" and "Block" elements 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 while it should point to an aggregate component when used on the "*Block*" element.

Req 92 **The "ref" attribute shall reference a scalar component when used on the "Component" element and an aggregate component when used on the "Block" element.**

The following binary encoded image data illustrates how this path like syntax is used:

```
<swe:DataArray definition="urn:ogc:def:data:OGC::Image">
  <swe:elementCount>
```

```
    <swe:Count definition="urn:ogc:def:data:OGC::ImageDimension">
      <swe:value>256</swe:value>
    </swe:Count>
  </swe:elementCount>
  <swe:elementType name="row">
    <swe:DataArray definition="urn:ogc:def:data:OGC::Row">
      <swe:elementCount>
        <swe:Count definition="urn:ogc:def:data:OGC::ImageDimension">
          <swe:value>256</swe:value>
        </swe:Count>
      </swe:elementCount>
      <swe:elementType name="pixel">
        <swe:DataRecord definition="urn:ogc:def:data:OGC::Pixel">
          <swe:field name="red">
            <swe:Quantity definition="urn:ogc:def:property:OGC::Radiance">
              <gml:description>Radiance measured on band1</gml:description>
              <swe:uom code="W.m-2.Sr-1.um-1"/>
            </swe:Quantity>
          </swe:field>
          <swe:field name="green">
            <swe:Quantity definition="urn:ogc:def:property:OGC::Radiance">
              <gml:description>Radiance measured on band2</gml:description>
              <swe:uom code="W.m-2.Sr-1.um-1"/>
            </swe:Quantity>
          </swe:field>
          <swe:field name="blue">
            <swe:Quantity definition="urn:ogc:def:property:OGC::Radiance">
              <gml:description>Radiance measured on band3</gml:description>
              <swe:uom code="W.m-2.Sr-1.um-1"/>
            </swe:Quantity>
          </swe:field>
        </swe:DataRecord>
      </swe:elementType>
      <swe:encoding>
        <swe:BinaryEncoding byteOrder="bigEndian" byteEncoding="base64">
          <swe:member>
            <swe:Component dataType="urn:ogc:def:data:OGC:unsignedByte" ref="row/pixel/red"/>
          </swe:member>
          <swe:member>
            <swe:Component dataType="urn:ogc:def:data:OGC:unsignedByte" ref="row/pixel/green"/>
          </swe:member>
          <swe:member>
            <swe:Component dataType="urn:ogc:def:data:OGC:unsignedByte" ref="row/pixel/blue"/>
          </swe:member>
        </swe:BinaryEncoding>
      </swe:encoding>
      <swe:values>
        FZEFZE564864HGZ6RG54Z684F86R7H4Z84FR8Z4685E468GTA4E8G4A6...
      </swe:values>
    </swe:DataArray>
  </swe:elementType>
</swe:DataArray>
```

In this example the root component is the element type of the array in which the values are embedded (i.e. the outer array). All paths used in the encoding section thus start with this component name (i.e. ‘row’) and then hierarchically list the names that lead to the scalar component whose data type is being defined.

8.6.2 Binary Encoding Rules

The “*BinaryEncoding*” method encodes field values by their binary representation. The EBNF syntax defined in ISO 14977 is used to formalize the encoding rules, and thus all EBNF snippet provided in this section are normative.

Req 93 Compliant encoding/decoding software shall implement the "BinaryEncoding" method by following the EBNF 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).

8.6.2.1 Rules for Scalar Components

8.6.2.1.1 Binary Data Types

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 definition and byte order ?;
```

This standard defines the list of data types that are allowed for scalar values and the corresponding URNs to use in an XML instance of the "*BinaryEncoding*" element.

Req 94 The value of the "dataType" XML attribute of the "Component" element shall be one of the URNs listed in Table 8.1.

These data types are specified in the normative table below:

Common Name	URN to use in "dataType" attribute	Description
Signed Byte	`urn:ogc:def:data:OGC:signedByte`	8-bits signed binary integer. Range: −128 to +127
Unsigned Byte	`urn:ogc:def:data:OGC:unsignedByte`	8-bits unsigned binary integer. Range: 0 to +255
Signed Short	`urn:ogc:def:data:OGC:signedShort`	16-bits signed binary integer. Range: −32,768 to +32,767
Unsigned Short	`urn:ogc:def:data:OGC:unsignedShort`	16-bits unsigned binary integer. Range: 0 to +65,535
Signed Int	`urn:ogc:def:data:OGC:signedInt`	32-bits signed binary integer. Range: −2,147,483,648 to +2,147,483,647
Unsigned Int	`urn:ogc:def:data:OGC:unsignedInt`	32-bits unsigned binary integer. Range: 0 to +4,294,967,295
Signed Long	`urn:ogc:def:data:OGC:signedLong`	64-bits signed binary integer. Range: −2^63 to +2^63 - 1
Unsigned Long	`urn:ogc:def:data:OGC:unsignedLong`	64-bits unsigned binary integer. Range: 0 to +2^64 - 1

Custom Integer*	`urn:ogc:def:data:OGC:integer`	Custom size integer (the actual length is specified by the "*bitLength*" or "*byteLength*" attribute).
Half Precision Float	`urn:ogc:def:data:OGC:float16`	16-bits single precision floating point number as defined in IEEE 754.
Float	`urn:ogc:def:data:OGC:float32`	32-bits single precision floating point number as defined in IEEE 754.
Double	`urn:ogc:def:data:OGC:double` or `urn:ogc:def:data:OGC:float64`	64-bits double precision floating point number as defined in IEEE 754.
Long Double	`urn:ogc:def:data:OGC:float128`	128-bits quadruple precision floating point number as defined in IEEE 754.
UTF-8 String (Variable Length)	`urn:ogc:def:data:OGC: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)	`urn:ogc:def:data:OGC: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.

Table 8.1 – Allowed Binary Data Types

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 insure that all future values for the component will "fit" in the data type.

Req 95 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 the description column (in the case of a variable length string, the size is included in the stream). This is enforced by a Schematron pattern.

Req 96 The "bitLength" and "byteLength" XML attribute shall not be set when a fixed size data type is used.

The last column of the table 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 muti-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.

Req 97 Binary data types in Table 8.1 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.

The value of the "*byteEncoding*" XML attribute allows the selection of either the 'raw' or 'base64' encoding methods. When the 'raw' option is selected, bytes resulting from the data type encoding process defined above are inserted in the binary stream directly. This is refered to as 'raw binary' encoding. When the 'base64' option is selected, each byte resulting from this encoding process is also encoded in Base64 before being included in the stream. Scalar values can be Base64 encoded one by one or by blocks as long as the resulting stream is compatible with requirements of IETF RFC 2045.

Req 98 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.

8.6.2.2 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;
```

Values are always included in the same order: The lower bound of the range first, followed by the upper bound.

8.6.2.3 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;
```

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.

Req 99 The 'Y' or 'N' 1-byte token shall be inserted in a binary encoded data block for all "DataRecord" 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;
```

Vector coordinates cannot be optional.

8.6.2.4 Rules for DataChoice

A "*DataChoice*" is encoded with the binary 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, and be encoded as a variable length string datatype.

```
selected-item-name = ? Value of the "name" attribute of the item selected ?;

selected-item-value = scalar-value | range-values | record-values | choice-values | array-values;

choice-values = selected-item-name, selected-item-value;
```

Req 100 The selected-item-name token shall correspond to the value of the "name" attribute of the "item" property element that represents the selected item.

8.6.2.5 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;
```

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;
```

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;
```

When the array size is 0, only this number is encoded and no element values are included in the data block.

8.6.2.6 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;
```

A data stream usually contains at least one value but could be empty.

8.6.2.7 Block encoded components

Binary encoding parameters can be specified for aggregate components in order to insert padding or achieve compression or encryption of whole or part of a dataset. This is achieved by using a "*Block*" element with its "*ref*" attribute pointing to an aggregate component in the data description and setting one or more of the "*compression*", "*encryption*" or "*padding*" attributes.

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 the block length is specified.

Annex A
(normative)

Abstract Conformance Test Suite

A.1 Conformance Test Class: Core Concepts

Tests described in this section shall be used to test conformance of software and encoding models implementing the Requirements Class: Core Concepts **(normative core)**.

A.1.1 Core concepts are the base of all derived models

A conformant model or software shall implement the concepts defined in the core of this standard in a way that is consistent with their definition.

a) Reference: Clause 6, Req 1

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.1.2 A boolean representation consists of a boolean value

A boolean representation shall at least consist of a boolean value.

a) Reference: Clause 6.2.1, Req 2

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.1.3 A categorical representation consists of a token with a code space

A categorical representation shall at least consist of a category identifier and information describing the value space of this identifier.

a) Reference: Clause 6.2.2, Req 3

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.1.4 A continuous numerical representation consists of a number with a scale

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

a) Reference: Clause 6.2.3, Req 4

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.1.5 A countable representation consists of an integer number

A countable representation shall at least consist of an integer number.

a) Reference: Clause 6.2.4, Req 5

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.1.6 A textual representation is implemented as a character string

A textual representation shall at least consist of a character string.

a) Reference: Clause 6.2.5, Req 6

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.1.7 Semantic definition of each measured property shall be provided

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

a) Reference: Clause 6.3.2, Req 7

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.1.8 References to semantical information shall be resolvable

If robust semantics are provided by referencing out-of-band information, the locators or identifiers used to point to this information shall be resolvable by some well-defined method.

a) Reference: Clause 6.3.2, Req 8

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.1.9 A temporal quantity is associated to a temporal reference frame

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

a) Reference: Clause 6.3.3, Req 9

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.1.10 A spatial quantity is associated to an axis of a spatial reference frame

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

a) Reference: Clause 6.3.3, Req 10

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.1.11 A NIL value maps a reserved value to a reason

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

a) Reference: Clause 6.4.2, Req 11

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.1.12 Aggregate data types are modeled according to ISO 11404

A conformant model or software shall implement aggregate data structures in a way that is consistent with definitions of ISO 11404.

a) Reference: Clause 6.5, Req 12

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.1.13 Encoding methods shall be defined for all possible data structures

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 6.5.

a) Reference: Clause 6.6, Req 13

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.2 Conformance Test Class: Simple Components UML Package

Tests described in this section shall be used to test conformance of software and encoding models implementing the Requirements Class: Basic Types and Simple Components Packages.

Software implementations shall at least be tested against this test class to claim conformance to this standard. Additionally, conformance of XML documents ingested and generated by the software shall be tested by using conformance test classes A.7 to A.12 when seeking compliance with the XML encodings defined in this standard.

A.2.1 Dependency on core

An implementation passing the “Simple Components UML Package” conformance test class shall first pass the core conformance test class.

d) Reference: Clause 7.2, Req 14

e) Test Type: Conformance

f) Test Method: Apply all tests described in section A.1

A.2.2 Compliance with UML models defined in this package

A compliant encoding or software shall correctly implement all UML classes defined in the "Simple Components" and "Basic Types" packages.

a) Reference: Clause 7.2, Req 15

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.2.3 Compliance with UML models defined in ISO 19103

A compliant encoding or software shall correctly implement all UML classes defined in ISO 19103 that are used in this standard.

a) Reference: Clause 7.2, Req 16

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.2.4 Compliance with UML models defined in ISO 19136

A compliant encoding or software shall correctly implement all UML classes defined in ISO 19136 (GML) that are used in this standard.

a) Reference: Clause 7.2, Req 17

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.2.5 Unknown extensions shall be ignored gracefully

A compliant implementation shall not generate errors when the content of an "extension" attribute is unknown.

a) Reference: Clause 7.2.3, Req 18

b) Test Type: Conformance

c) Test Method: Verify that the implementation is able to handle extensions by, at the minimum, ignoring them without triggering errors. If extensions are supported check that they are made available via the generic "extension" property.

A.2.6 A definition URI is mandatory on all simple components

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

a) Reference: Clause 7.2.4, Req 19

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.2.7 Reference frames are described using ISO 19111 models

The URI used as the value of the "referenceFrame" attribute shall identify a coordinate reference system as defined by ISO 19111.

a) Reference: Clause 7.2.4, Req 20

b) Test Type: Conformance

c) Test Method: Check that the CRS identifier can be mapped to an ISO 19111 Coordinate Reference System definition. This definition can be obtained from a well known registry (e.g. EPSG), the local machine, or a remote location if the identifier (URI) can be dynamically resolved to it.

A.2.8 The value of the axisID and axisAbbrev attributes match

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.

a) Reference: Clause 7.2.4, Req 21

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.2.9 The axis ID is always specified on scalar spatial properties

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

a) Reference: Clause 7.2.4, Req 22

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.2.10 The reference frame is specified on scalar spatial properties not part of a vector

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).

a) Reference: Clause 7.2.4, Req 23

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.2.11 The value of a component satisfies the constraints

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.

a) Reference: Clause 7.2.4, Req 24

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.2.12 All derived simple components have an optional value attribute

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.

a) Reference: Clause 7.2.4, Req 25

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.2.13 The list of values allowed in a Category component is a subset of the code space

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.

a) Reference: Clause 7.2.7, Req 26

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.2.14 A Category component always specifies a list of possible values

An instance of the "Category" class shall either specify a code space or an enumerated list of allowed tokens, or both.

a) Reference: Clause 7.2.7, Req 27

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.2.15 The value of a Category component is one defined in the code space

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.

a) Reference: Clause 7.2.7, Req 28

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.2.16 A reference frame is always specified on a Time component

The "referenceFrame" attribute inherited from "AbstractSimpleComponent" shall be set on all instances of the "Time" class.

a) Reference: Clause 7.2.10, Req 29

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.2.17 The time of reference is expressed relative to the origin of the reference frame

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

a) Reference: Clause 7.2.10, Req 30

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.2.18 The local and reference frames of a Time component are different

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

a) Reference: Clause 7.2.10, Req 31

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.2.19 The scale of a Time component is always a temporal unit

The "uom" attribute of an instance of the "Time" class shall specify a base or derived time unit.

a) Reference: Clause 7.2.10, Req 32

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.2.20 Values of range components satisfy the same requirements as scalar values

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.

a) Reference: Clause 7.2.11, Req 33

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the conceptual model has constraints that enforce the above.

A.2.21 CategoryRange components satisfy all requirements of a Category component

All requirements associated to the "Category" class defined in clause §7.2.7 apply to the "CategoryRange" class.

a) Reference: Clause 7.2.12, Req 34

b) Test Type: Conformance

c) Test Method: Apply conformance tests A.2.13 to A.2.15 to the "CategoryRange" class.

A.2.22 The code space of a CategoryRange component is well-ordered

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

a) Reference: Clause 7.2.12, Req 35

b) Test Type: Conformance

c) Test Method: Inspect the information defining the code space to verify the above.

A.2.23 TimeRange components satisfy all requirements of the Time class

All requirements associated to the "Time" class defined in clause §7.2.10 apply to the "TimeRange" class.

a) Reference: Clause 7.2.15, Req 36

b) Test Type: Conformance

c) Test Method: Apply conformance tests A.2.16 to A.2.19 to the "CategoryRange" class.

A.2.24 The reason attribute is a URI that is resolvable to a definition

The "reason" attribute of an instance of the "NilValue" class shall contain a URI that can be resolved to the complete human readable definition of the reason associated with the NIL value.

a) Reference: Clause 7.2.17, Req 37

b) Test Type: Conformance

c) Test Method: Check that the NIL reason identifier corresponds to either a well known reason code defined by OGC or can be resolved to the textual description of a custom reason.

A.2.25 Values reserved for NIL reasons are compatible with the component data type

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.

a) Reference: Clause 7.2.17, Req 38

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.2.26 The scale of constraints is the same as the scale of the component value

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.

a) Reference: Clause 7.2.19, Req 39

b) Test Type: Conformance

c) Test Method: Inspect instances generated by the implementation of the "Quantity", "Count" and "Time" classes including an "AllowedValues" constraint to verify the above.

A.3 Conformance Test Class: Aggregate Components UML Package

A.3.1 Dependency on Simple Components package

An implementation passing the "Aggregate Components UML Package" conformance test class shall first pass the "Basic Types and Simple Components UML Packages" conformance test class.

d) Reference: Clause 7.3, Req 40

e) Test Type: Conformance

f) Test Method: Apply all tests described in section A.2.

A.3.2 Compliance with UML models defined in this package

A compliant encoding or software shall correctly implement all UML classes defined in the "Aggregate Components" package.

a) Reference: Clause 7.3, Req 41

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.3.3 Each DataRecord field has a unique name

Each "field" attribute in a given instance of the "DataRecord" class shall be identified by a name that is unique to this instance.

a) Reference: Clause 7.3.1, Req 42

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the "DataRecord" class has a constraint that enforces the above.

A.3.4 Each DataChoice item has a unique name

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

a) Reference: Clause 7.3.2, Req 43

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the "DataChoice" class has a constraint that enforces the above.

A.3.5 The reference frame is not specified on individual coordinates of a Vector

The "referenceFrame" attribute shall be ommited from all data components used to define coordinates of a "Vector" instance.

a) Reference: Clause 7.3.3, Req 44

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.3.6 The axis ID is specified on all coordinates of a Vector

The "axisID" attribute shall be specified on all data components used to define coordinates of a "Vector" instance.

a) Reference: Clause 7.3.3, Req 45

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.3.7 The local and reference frames of a Vector component are different

The "localFrame" attribute of an instance of the "Vector" class shall have a different value than the "referenceFrame" attribute.

d) Reference: Clause 7.3.3, Req 46

e) Test Type: Conformance

f) Test Method: Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.4 Conformance Test Class: Block Components UML Package

A.4.1 Dependency on Aggregate Components package

An implementation passing the "Block Components UML Package" conformance test class shall first pass the "Aggregate Components UML Package" and "Simple Encodings UML Package" conformance test classes.

g) Reference: Clause 7.4, Req 47

h) Test Type: Conformance

i) Test Method: Apply all tests described in sections A.3 and A.5.

A.4.2 Compliance with UML models defined in this package

A compliant encoding or software shall correctly implement all UML classes defined in the "Block Components" package.

a) Reference: Clause 7.4, Req 48

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.4.3 Components nested in a block component are data descriptors

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.

a) Reference: Clause 7.4.1, Req 49

b) Test Type: Conformance

c) Test Method: Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.4.4 An encoding method is specified whenever an encoded data block is included

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.

a) Reference: Clause 7.4.1, Req 50

b) Test Type: Conformance

c) Test Method: Inspect block components instances ("DataArray", "DataStream" and "Matrix") generated by the implementation to verify that an encoding method is specified and properly used. (Note that detailed requirements for encoding data are only defined in section 8 XML Implementation **(normative)**, but these requirements are also applicable when data streams are not wrapped in XML).

A.5 Conformance Test Class: Simple Encodings UML Package

A.5.1 Dependency on Basic Types package

An implementation passing the "Simple Encodings UML Package" conformance test class shall first pass "Basic Types and Simple Components UML Package" conformance test class.

a) Reference: Clause 7.5, Req 51

b) Test Type: Conformance

c) Test Method: Apply all tests described in section A.2.

A.5.2 Compliance with UML models defined in this package

A compliant encoding or software shall correctly implement all UML classes defined in the "Simple Encodings" package.

a) Reference: Clause 7.5, Req 52

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.6 Conformance Test Class: Advanced Encodings UML Package

A.6.1 Dependency on Simple Encodings package

An implementation passing the "Advanced Encodings UML Package" conformance test class shall first pass the "Simple Encodings UML Package" conformance test class.

a) Reference: Clause 7.6, Req 53

b) Test Type: Conformance

c) Test Method: Apply all tests described in section A.5.

A.6.2 Compliance with UML models defined in this package

A compliant encoding or software shall correctly implement all UML classes defined in the "Advanced Encodings" package.

a) Reference: Clause 7.6, Req 54

b) Test Type: Conformance

c) Test Method: Inspect the model or software implementation to verify the above.

A.7 Conformance Test Class: XML Encoding Principles

All tests in this conformance test class and in the following shall be used to check conformance of XML instances created according to the schemas defined in this standard. They shall also be used to check conformance of software implementations that output XML instances.

For all software implementations that provide reading functionality of the SWE Common XML format, the behaviour of the software when ingesting invalid XML instances shall be tested as well. This shall be done by running the tests described in this section and making sure that at least one of the tests fails.

A.7.1 Dependency on Core

An implementation passing the "XML Encoding Principles" conformance test class shall first pass the core conformance test classes.

a) Reference: Clause 8.1, Req 55

b) Test Type: Conformance

c) Test Method: Apply all tests described in section A.1.

A.7.2 XML property values are included inline or by reference

A property element supporting the “gml:AssociationAttributeGroup” shall contain the value inline or populate the “xlink:href” attribute with a valid reference but shall not be empty.

a) Reference: Clause 8.1.2, Req 56

b) Test Type: Conformance

c) Test Method: Check that all properties either include an inline value or an “xlink:href” attribute.

A.7.3 Each extension uses a different namespace

All extensions of the XML schemas described in this standard shall be defined in a new unique namespace.

a) Reference: Clause 8.1.3, Req 57

b) Test Type: Conformance

c) Test Method: If the standardization target is an extension of the XML schema defined in this standard, inspect the XML schema of the extension to verify the above.

A.7.4 Extensions do not redefine XML elements or types

Extensions of this standard shall not redefine or change the meaning or behavior of XML elements and types defined in this standard.

a) Reference: Clause 8.1.3, Req 58

b) Test Type: Conformance

c) Test Method: If the standardization target is an extension of the XML schema defined in this standard, inspect the XML schema of the extension to verify the above.

A.8 Conformance Test Class: Basic Types and Simple Components Schemas

A.8.1 Dependency on XML Encoding Principles

An implementation passing the "Basic Types and Simple Components Schemas" conformance test class shall first pass the "XML Encoding Principles" and core conformance test classes.

a) Reference: Clause 8.2, Req 59

b) Test Type: Conformance

c) Test Method: Apply all tests described in section A.7.

A.8.2 Dependency on GML

An implementation passing the "Basic Types and Simple Components Schemas" conformance test class shall first pass the "Abstract test suite for GML documents" conformance test class of the GML 3.2.1 standard.

a) Reference: Clause 8.2, Req 60

b) Test Type: Conformance

c) Test Method: Apply all tests defined in the GML 3.2.1 standard that are applicable to XML elements used in this standard.

A.8.3 Compliance with XML schemas and Schematron patterns

A compliant XML instance shall be valid with respect to the XML grammar defined in the "basic_types.xsd" and "simple_components.xsd" XML as well as satisfy all Schematron patterns defined in "simple_components.sch".

a) Reference: Clause 8.2, Req 61

b) Test Type: Conformance

c) Test Method: Validate the XML instance containing simple data components with the "swe.xsd" XML schema file and the Schematron patterns in "simple_components.sch".

A.8.4 The value of the definition attribute is a resolvable URI

The "definition" attribute shall contain a URI that can be resolved to the complete human readable definition of the property that is represented by the data component.

d) Reference: Clause 8.2.1, Req 62

e) Test Type: Conformance

f) Test Method: Verify that the URI can be resolved to an online document (or a document fragment if the URI includes a fragment) describing the type of property. In the case of a URL, check that connecting to the specified address results in the successful retrieval of the document. In the case of a URN check that a registry is available to resolve it to a URL that behaves as specified above or directly to retrieve the document.

A.8.5 Data component inline value satisfies the constraints

The inline value included in an instance of a simple data component shall satisfy the constraints specified by this instance.

g) Reference: Clause 8.2.1, Req 63

h) Test Type: Conformance

i) Test Method: This test is run only on instances of simple data components that include a constraint (i.e. using one of "AllowedValues", "AllowedTimes" or "AllowedTokens" elements) and an inline value. For such instances, verify that the inline value is valid with respect to the specified constraint(s).

A.8.6 UCUM is used whenever possible

The UCUM code for a unit of measure shall be used as the value of the "code" XML attribute whenever it can be constructed using the UCUM 1.8 specification. Otherwise the "href" XML attribute shall be used to reference an external unit definition.

a) Reference: Clause 8.2.6, Req 64

b) Test Type: Conformance

c) Test Method: Verify that in all instances of the "Quantity" class, values of the "code" attribute on the "uom" element are valid UCUM expressions. When the "code" attribute is not used verify that the "href" attribute is present and that it is only used to reference units of measure that cannot be expressed using UCUM.

A.8.7 URI to use for specifying ISO 8601 encoding

When ISO 8601 notation is used to express the measurement value associated to a "Time" element, the URI "urn:ogc:def:unit:ISO:8601" shall be used as the value of the "xlink:href" XML attribute on the "uom" element.

a) Reference: Clause 8.2.7, Req 65

b) Test Type: Conformance

c) Test Method: Verify that in all instances of the "Time" class including a temporal value encoded as ISO 8601 (either inline or in a block encoded data stream) the proper URN is used as the unit.

A.8.8 Pattern constraints are expressed using Unicode regular expressions

The "pattern" child element of the "AllowedTokens" element shall be a regular expression valid with respect to Unicode Technical Standard #18, Version 13.

a) Reference: Clause 8.2.14, Req 66

b) Test Type: Conformance

c) Test Method: Verify that all character strings used as regular expressions are valid according to the Unicode standard.

A.9 Conformance Test Class: Aggregate Components Schema

A.9.1 Dependency on Basic Types and Simple Components schemas

An implementation passing the "Aggregate Components Schema" conformance test class shall first pass the "Basic Types and Simple Components Schemas" conformance test class.

a) Reference: Clause 8.3, Req 67

b) Test Type: Conformance

c) Test Method: Apply all tests described in section A.8.

A.9.2 Compliance with XML schema and Schematron patterns

A compliant XML instance shall be valid with respect to the XML grammar defined in the "aggregate_components.xsd" XML schema as well as satisfy all Schematron patterns defined in "aggregate_components.sch".

a) Reference: Clause 8.3, Req 68

b) Test Type: Conformance

c) Test Method: Validate the XML instance containing aggregate components with the "swe.xsd" XML schema file and the Schematron patterns in "aggregate_components.sch".

A.10 Conformance Test Class: Block Components Schema

A.10.1 Dependency on Aggregate Components and Simple Encodings schemas

An implementation passing the "Block Components Schema" conformance test class shall first pass the "Aggregate Components Schema" and "Simple Encodings Schema" conformance test classes.

a) Reference: Clause 8.4, Req 69

b) Test Type: Conformance

c) Test Method: Apply all tests described in sections A.9 and A.11.

A.10.2 Compliance with XML schema and Schematron patterns

A compliant XML instance shall be valid with respect to the grammar defined in the "block_components.xsd" XML schema as well as satisfy all Schematron patterns defined in "block_components.sch".

a) Reference: Clause 8.4, Req 70

b) Test Type: Conformance

c) Test Method: Validate the XML instance containing block components with the "swe.xsd" XML schema file and the Schematron patterns in "block_components.sch".

A.10.3 Encoding of array elements is consistent with the DataArray definition

The encoded data block included either inline or by-reference in the "values" property of a "DataArray", "Matrix" or "DataStream" element shall be consistent with the definition of the element type, the element count and the encoding method.

a) Reference: Clause 8.4.1, Req 71

b) Test Type: Conformance

c) Test Method: Verify that the data block is effectively encoded with the specified encoding method. Decode the data block as specified by this standard and verify that the decoded data is actually a sequence of values that is consistent with the element type definition: For each decoded value in the sequence, verify that it is consistent with the data type and constraints (including the code space for a "Count" component) of the corresponding data component. Verify that the total number of decoded elements is equal to the element count.

A.11 Conformance Test Class: Simple Encodings Schema

A.11.1 Dependency on Basic Types and Simple Components schema

An implementation passing the "Simple Encodings Schema" conformance test class shall first pass the "Basic Types and Simple Components Schemas" conformance test class.

a) Reference: Clause 8.5, Req 72

b) Test Type: Conformance

c) Test Method: Apply all tests described in section A.8.

A.11.2 Compliance with XML schema and Schematron patterns

A compliant XML instance shall be valid with respect to the grammar defined in the "simple_encodings.xsd" XML schema as well as satisfy all Schematron patterns defined in "simple_encodings.sch".

a) Reference: Clause 8.5, Req 73

b) Test Type: Conformance

c) Test Method: Validate the XML instance containing definitions of simple encodings with the "swe.xsd" XML schema file and the Schematron patterns in "simple_encodings.sch".

A.11.3 DataRecord fields and Vector coordinates are encoded recursively

"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.

a) Reference: Clause 8.5.1.3, Req 74

b) Test Type: Conformance

c) 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.

A.11.4 DataChoice items are encoded recursively

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

a) Reference: Clause 8.5.1.4, Req 75

b) Test Type: Conformance

c) 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.

A.11.5 DataArray elements are encoded recursively

"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).

a) Reference: Clause 8.5.1.5, Req 76

b) Test Type: Conformance

c) 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.

A.11.6 The length of variable size arrays is encoded in the data block

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

a) Reference: Clause 8.5.1.5, Req 77

b) Test Type: Conformance

c) 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.11.7 Text Encoding: Compliance with EBNF grammar

Compliant encoding/decoding software shall implement the "TextEncoding" method by following the EBNF grammar defined in this clause.

a) Reference: Clause 8.5.4, Req 78

b) Test Type: Conformance

c) Test Method: Verify that the text encoded data block is correct with respect to the EBNF grammar corresponding to the particular dataset (The complete EBNF grammar of the dataset should be logically constructed from the EBNF snippets provided in the specification).

A.11.8 Text Encoding: Separator characters are well chosen

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.

a) Reference: Clause 8.5.4.1, Req 79

b) Test Type: Conformance

c) 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.

A.11.9 Text Encoding: Special flags are inserted before optional component values

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'.

a) Reference: Clause 8.5.4.4, Req 80

b) Test Type: Conformance

c) Test Method: Verify that the sequence of values in the section of the data block corresponding to the optional value starts with the 'Y' or 'N' flag.

A.11.10 Text Encoding: The name of a selected choice item is inserted in the stream

The selected-item-name token shall correspond to the value of the "name" attribute of the "item" property element that represents the selected item.

a) Reference: Clause 8.5.4.5, Req 81

b) Test Type: Conformance

c) Test Method: Verify that the sequence of values in the section of the data block corresponding to the "DataChoice" starts with a character string matching the name of one item of the choice.

A.11.11 XML Encoding: Element local names are derived from name attribute

All data components shall be XML encoded with an element whose local name shall correspond to the "name" attribute of the soft-typed property containing the data component.

a) Reference: Clause 8.5.6.1, Req 82

b) Test Type: Conformance

c) Test Method: Inspect the XML of the encoded data block to verify the above.

A.11.12 XML Encoding: Scalar components are encoded with an XML element with text content

Scalar components values shall be XML encoded with a single element containing the value as its text content and no other child element.

a) Reference: Clause 8.5.6.2, Req 83

b) Test Type: Conformance

c) Test Method: Inspect the XML of the encoded data block to verify the above.

A.11.13 XML Encoding: Range components are encoded as a group of two XML elements

Range components values shall be XML encoded with an element containing two sub-elements with local names "min" and "max" which respectively contain the lower and upper values of the range as their text content.

a) Reference: Clause 8.5.6.3, Req 84

b) Test Type: Conformance

c) Test Method: Inspect the XML of the encoded data block to verify the above.

A.11.14 XML Encoding: DataRecord components are encoded as an XML element with complex content

"DataRecord" values shall be XML encoded with an element which contains one sub-element for each "field" that is not omitted.

a) Reference: Clause 8.5.6.4, Req 85

b) Test Type: Conformance

c) Test Method: Inspect the XML of the encoded data block to verify the above.

A.11.15 XML Encoding: Vectors components are encoded as an XML element with complex content

"Vector" values shall be XML encoded with an element which contains one sub-element for each "coordinate" of the aggregate.

a) Reference: Clause 8.5.6.4, Req 86

b) Test Type: Conformance

c) Test Method: Inspect the XML of the encoded data block to verify the above.

A.11.16 XML Encoding: Array elements are encoded as an XML element with complex content

Values of each element of a "DataArray", "Matrix" or "DataStream" shall be encapsulated in a separate XML element whose local name shall be the value of the "name" attribute of its "elementType" element.

a) Reference: Clause 8.5.6.5, Req 87

b) Test Type: Conformance

c) Test Method: Inspect the XML of the encoded data block to verify the above.

A.11.17 XML Encoding: Nested arrays are encoded with an XML element with a size

All elements of each nested "DataArray" and "Matrix" shall be encapsulated in a parent element as specified in Req 82 and this element shall also have an "elementCount" attribute that specifies the array size.

a) Reference: Clause 8.5.6.5, Req 88

b) Test Type: Conformance

c) Test Method: Inspect the XML of the encoded data block to verify the above.

A.12 Conformance Test Class: Advanced Encodings Schema

A.12.1 Dependency on Simple Encodings Schema

An implementation passing the "Advanced Encodings Schema" conformance test class shall first pass the "Simple Encodings Schema" conformance test class.

a) Reference: Clause 8.6, Req 89

b) Test Type: Conformance

c) Test Method: Apply all tests described in section A.11.

A.12.2 Compliance with XML schema and Schematron patterns

A compliant XML instance shall be valid with respect to the grammar defined in the "advanced_encodings.xsd" XML schema as well as satisfy all Schematron patterns defined in "advanced_encodings.sch".

a) Reference: Clause 8.6, Req 90

b) Test Type: Conformance

c) Test Method: Validate the XML instance containing definitions of simple encodings with the "swe.xsd" XML schema file and the Schematron patterns in "simple_encodings.sch".

A.12.3 The path value in the ref attribute has the correct syntax

The "ref" attribute of the "Component" and "Block" elements shall contain a hierarchical '/' separated list of data component names.

a) Reference: Clause 8.6.1, Req 91

b) Test Type: Conformance

c) Test Method: Inspect the section of the XML instance describing the binary encoding options. Check that the path formed by the '/' separated list of component names actually points to a component of the dataset definition tree.

A.12.4 The path value in the ref attribute points to a valid component

The "ref" attribute shall reference a scalar component when used on the "Component" element and an aggregate component when used on the "Block" element.

a) Reference: Clause 8.6.1, Req 92

b) Test Type: Conformance

c) Test Method: Inspect the section of the XML instance describing the binary encoding options. Resolve the path to a component of the dataset definition tree and check that this component is of the right type.

A.12.5 Compliance with EBNF grammar

Compliant encoding/decoding software shall implement the "BinaryEncoding" method by following the EBNF grammar defined in this clause.

a) Reference: Clause 8.6.2, Req 93

b) Test Type: Conformance

c) Test Method: Verify that the binary encoded data block is correct with respect to the EBNF grammar of the particular dataset (The complete EBNF grammar of the dataset should be logically constructed from the EBNF snippets provided in the specification).

A.12.6 The chosen datatype is one of the possible options

The value of the "dataType" XML attribute of the "Component" element shall be one of the URNs listed in Table 8.1.

a) Reference: Clause 8.6.2.1.1, Req 94

b) Test Type: Conformance

c) Test Method: Verify that the URN used to specify the binary data type is in the list.

A.12.7 The chosen datatype is compatible with the associated component

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

a) Reference: Clause 8.6.2.1.1, Req 95

b) Test Type: Conformance

c) 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 (urn:ogc:def:data:OGC:unsignedByte)

A.12.8 The length of a datatype is specified only when appropriate

The “bitLength” and “byteLength” XML attribute shall not be set when a fixed size data type is used.

a) Reference: Clause 8.6.2.1.1, Req 96

b) Test Type: Conformance

c) Test Method: Verify that these attributes are used only when one of the UTF-8 String or Custom Integer data types is selected.

A.12.9 Data types are encoded as specified in this standard

Binary data types in Table 8.1 shall be encoded according to their definition in the description column and the value of the "byteOrder" attribute.

a) Reference: Clause 8.6.2.1.1, Req 97

b) Test Type: Conformance

c) 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, abberant values (such as -65502 for a temperature field, etc…) are usually obtained, which can be easily detected.

A.12.10 Base64 encoding is implemented as defined by IETF

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.

a) Reference: Clause 8.6.2.1.1, Req 98

b) Test Type: Conformance

c) Test Method: 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 has been decoded into a raw binary data stream.

A.12.11 Special flags are inserted before optional component values

The 'Y' or 'N' 1-byte token shall be inserted in a binary encoded data block for all "DataRecord" fields that have the "optional" attribute set to 'true'.

a) Reference: 8.6.2.3, Req 99

b) Test Type: Conformance

c) Test Method: 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 has been decoded into a raw binary data stream.

A.12.12 The name of a selected choice item is inserted in the stream

The selected-item-name token shall correspond to the value of the "name" attribute of the "item" property element that represents the selected item.

a) Reference: Clause 8.6.2.4, Req 100

b) Test Type: Conformance

c) Test Method: 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 has been decoded into a raw binary data stream.

Annex B
(informative)

Relationship with other ISO models

B.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, and a special meaning is associated to these objects.

A good example is a set of weather observations obtained from different sensors that may be grouped into a single "*Record*" in SWE Common.

B.2 Geometry model

SWE provides lower level data from which GML form, canonical XML representation of ISO19107 geometries, can be derived

B.3 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 ISO "*Coverage*" model 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 ISO model.

Table of Requirements