OGC® DOCUMENT: 23-000

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

Open Geospatial Consortium

OGC SENSORML ENCODING STANDARD

STANDARD
Implementation

APPROVED

Version: 3.0.0
Submission Date: 2025-03-19
Approval Date: 2025-06-02
Publication Date: 2025-07-16
Editor: Alexandre Robin

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF RECOMMENDATIONS

I ABSTRACT

The primary focus of the Sensor Model Language (SensorML) is to provide a robust and semantically-tied means of defining processes and processing components associated with the measurement and post-measurement transformation of observations. This includes sensors and actuators as well as computational processes applied pre- and post-measurement.

The main objective is to enable interoperability, first at the syntactic level and later at the semantic level (by using ontologies and semantic mediation), so that sensors and processes can be better understood by machines, utilized automatically in complex workflows, and easily shared between intelligent sensor web nodes.

This standard is one of several implementation standards produced under OGC's Sensor Web Enablement (SWE) activity. This standard is a revision of content that was previously integrated in the SensorML version 1.0 standard (OGC 07-000), version 2.0 (OGC 12-000), and version 2.1 (OGC 12-000r2).

II KEYWORDS

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

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

III PREFACE

This Standard arises from work undertaken by the **OGC API — Connected Systems Standards Working Group** of the OGC, with the aim of modernizing the Sensor Web Enablement (SWE) suite of Standards. The working group is concerned with establishing interfaces and encodings that will enable a “Sensor Web” through which applications and services will be able to access connected systems of all types (e.g., sensors, actuators, robots), the observations generated by them, as well as provide command and control functionalities.

This Standard specifies models and a JSON implementation for the SensorML.

This document supersedes and replaces OGC® Sensor Model Language (SensorML) Specification version 2.1 (OGC 12-000r2).

The main changes of SensorML 3.0 from SensorML version 2.1 are:

- Addition of the JSON encodings and schemas
- Addition of the Deployment class
- Addition of the Derived Property class
- Removal of the XML encodings

This release is fully backward compatible with version 2.1.

SensorML is well-suited for describing sensor model imaging geometries – the SensorML 2.0 RFC contains examples of a frame camera sensor model based on the Community Standard Model from NGA (NGA.SIG.0002_2.1). Additional (and more complete) sensor model descriptions are being compiled into a sensor model repository by the OGC Naming Authority, based on work by Gobe Hobona [OGC 18-042r3 (unpublished)]. In addition, work to connect OGC grid coverages to SensorML 2 that began in 2013 is now completed, which involved extending CIS 1.0 [OGC 09-146r2] via the ReferenceableGridCoverage Extension [OGC 16-083r3] to support SensorML 2 descriptions. Version 2.1 of the GMLJP2 imagery standard [OGC 08-083r8] takes advantage of this coverage extension standard to support embedded and externally located SensorML 2 descriptions, thereby giving GMLJP2 the ability to support “raw” sensor model imagery.

IV SECURITY CONSIDERATIONS

SensorML documents will often be used to transmit confidential or sensitive data. Encryption in-transit using HTTPS (i.e., HTTP over TLS/SSL) is thus highly recommended and is now very common practice on the web.

In addition, implementations of this standard may also store confidential or sensitive data (e.g., in a database) for extended periods of time. In this case, encryption at rest is also recommended, especially if data is hosted on a shared infrastructure (e.g., public clouds).

Security constraints for individual documents may be defined as described in Clause 8.2.2.5.

V SUBMITTING ORGANIZATIONS

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

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

VI SUBMITTERS

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

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

Additional contributors to this Standard include the following:

NAME	AFFILIATION
Arne Broering	52° North Initiative
Eric Hirschon	Eric Hirschon
Ingo Simonis	iGSI

NAME	AFFILIATION
Johannes Echterhoff	iGSI
Luis Bermudez	SURA

1

SCOPE

1 SCOPE

This Standard defines conceptual models and JSON encodings for SensorML. The primary focus of SensorML is to provide a framework for defining processes and processing components associated with the measurement and post-measurement transformation of observations. Thus, SensorML has more of a focus on the process of measurement and observation, rather than on sensor hardware, yet still provides a robust means of defining the physical characteristics and functional capabilities of physical processes such as sensors and actuators.

The aims of SensorML are to:

- Provide descriptions of sensors and sensor systems for inventory management;
- Provide sensor and process information in support of asset and observation discovery;
- Support the processing and analysis of the sensor observations;
- Support the geolocation of observed values (measured data);
- Provide performance and quality of measurement characteristics (e.g., accuracy, threshold, etc.);
- Provide general descriptions of components (e.g., a particular model or type of a sensor) as well as the specific configuration of that component when its deployed;
- Provide a machine interpretable description of the interfaces and data streams flowing in and out of a component;
- Provide an explicit description of the process by which an observation was obtained (i.e., its lineage);
- Provide an executable aggregate process for deriving new data products on demand (i.e., derivable products); and
- Archive fundamental properties and assumptions regarding sensor systems and computational processes.

SensorML provides a common framework for any process, but is particularly well-suited for the description of sensor and systems and the processes surrounding sensor observations. Within SensorML, sensor and transducer components (detectors, transmitters, actuators, and filters) are all modeled as physical processes that can be connected and participate equally within a process network or system, and which utilize the same model framework as any other process.

Processes are entities that take one or more inputs and through the application of well-defined methods and configurable parameters and produce one or more outputs. The process model defined in SensorML can be used to describe a wide variety of processes, including not only sensors, but also actuators and data processes, to name a few. SensorML also supports explicit linking between processes and thus supports the concept of process chains, networks, or workflows, which are themselves defined as processes using a composite pattern.

SensorML provides a framework within which the geometric, dynamic, and observational characteristics of sensors and sensor systems can be defined. There are a great variety of sensor types, from simple thermometers to complex electron microscopes and earth observing satellites. These can all be supported through the definition of simple and aggregate processes.

The models and schema within the core SensorML specification provide a “skeletal” framework for describing processes, aggregate processes, and sensor systems. Interoperability within and between various sensor communities, is greatly improved through the definition of shared community-specific semantics (within online dictionaries or ontologies) that can be utilized within the framework. In addition, the profiling of small, general-use, atomic processes that can serve as components within aggregate processes and systems is envisioned.

2

CONFORMANCE

2 CONFORMANCE

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

This Standard defines conceptual models and a JSON implementation of these models for describing non-physical and physical processes surrounding the act of measurement and subsequent processing of observations. The conceptual models are described using UML while the implementation is described using the JSON Schema language.

This Standard defines the following requirements classes and standardization targets:

Table 1 — Requirements Classes

REQUIREMENTS CLASS	STANDARDIZATION TARGET
Core	Derived Models and Software Implementations
Clause 7, Requirements Class: Core Concepts (normative core)	
UML Models	Software Implementation or Encoding of the Conceptual Models
Clause 8.2, Requirements Class: Core Abstract Process	
Clause 8.3, Requirements Class: Simple Process	
Clause 8.4, Requirements Class: Aggregate Process	
Clause 8.5, Requirements Class: Physical Component	
Clause 8.6, Requirements Class: Physical System	
Clause 8.7, Requirements Class: Processes with Advanced Data Types	
Clause 8.8, Requirements Class: Configurable Processes	
Clause 8.9, Requirements Class: Deployment	
Clause 8.10, Requirements Class: Derived Property	

REQUIREMENTS CLASS	STANDARDIZATION TARGET
JSON Encodings	
Clause 9.1, Requirements Class: Core Schema	JSON Document
Clause 9.2, Requirements Class: Simple Process Schema	
Clause 9.3, Requirements Class: Aggregate Process Schema	
Clause 9.4, Requirements Class: Physical Component Schema	
Clause 9.5, Requirements Class: Physical System Schema	
Clause 9.6, Requirements Class: Deployment Schema	
Clause 9.7, Requirements Class: Derived Property Schema	

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

- An implementation that defines a new data model shall at least conform with the core requirements class.
- An encoding of the conceptual models (e.g., a protobuf encoding) shall implement at least one of the requirements classes listed in the “UML Models” section of the table.
- An implementation that produces or consumes SensorML descriptions encoded in JSON shall implement at least one of the requirements classes listed in the “JSON Encodings” section of the table.

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

3

NORMATIVE REFERENCES

3 NORMATIVE REFERENCES

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

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

Alexandre Robin: OGC 24-014, OGC SWE Common Data Model Encoding Standard, version 3.0 (2025). https://docs.ogc.org/is/24-014/24-014.html

Carl Stephen Smyth: OGC 21-056r11, *OGC GeoPose 1.0 Data Exchange Standard*. Open Geospatial Consortium (2023). http://www.opengis.net/doc/IS/geopose/1.0.0.

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

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

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

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

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

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

ISO: ISO 19115:2003/Cor 1:2006, *Geographic information — Metadata — Technical Corrigendum 1*. International Organization for Standardization, Geneva (2003). https://www.iso.org/standard/44361.html.

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

T. Bray (ed.): IETF RFC 8259, *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC Publisher (2017). https://www.rfc-editor.org/info/rfc8259.

M. Nottingham: IETF RFC 8288, *Web Linking*. RFC Publisher (2017). https://www.rfc-editor.org/info/rfc8288.

JSON Schema Validation: A Vocabulary for Structural Validation of JSON, Version 2020-12, https://json-schema.org/draft/2020-12/json-schema-validation.html

H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, T. Schaub: IETF RFC 7946, *The GeoJSON Format*. RFC Publisher (2016). https://www.rfc-editor.org/info/rfc7946.

4

TERMS AND DEFINITIONS

4 TERMS AND DEFINITIONS

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

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

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

4.1. Actuator

A type of transducer that converts a signal to some real-world action or phenomenon.

4.2. Aggregate Process

Composite process consisting of interconnected sub-processes, which can in turn be Simple Processes or themselves Aggregate Processes. An aggregate process can include possible data sources. A description of an aggregate process should explicitly define connections that link input and output signals of sub-processes together. Since it is a process itself, an aggregate process also has its own inputs, outputs and parameters.

4.3. Coordinate Reference System (CRS)

A spatial or temporal framework within which a position and/or time can be defined. According to ISO 19111, a coordinate system that is related to the real world by a datum.

4.4. Coordinate System (CS)

According to ISO19111, a set of (mathematical) rules for specifying how coordinates are assigned to points. In this document, a Coordinate System is extended to be defined as a set of axes with which location and orientation can be defined.

4.5. Data Component

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

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

4.6. Datum

Undefined in ISO 19111. Defined here as a means of relating a coordinate system to the real world by specifying the physical location of the coordinate system and the orientation of the axes relative to the physical object. For a geodetic datum, the definition also includes a reference ellipsoid that approximates the physical or gravitational surface of the planetary body.

4.7. Detector

Atomic part of a composite Measurement System defining sampling and response characteristic of a simple detection device. A detector has only one input and one output, both being scalar quantities. More complex Sensors, such as a frame camera, which are composed of multiple detectors, can be described as a detector group or array using a System or Sensor model.

4.8. Determinand

A Parameter or a characteristic of a phenomenon subject to observation. Synonym for observable.

OGC 20-082r4

4.9. Feature

Abstraction of real-world phenomena

ISO 19101:2002, definition 4.11

Note 1 to entry: A feature may occur as a type or an instance. Feature type or feature instance should be used when only one is meant.

4.10. Location

A point or extent in space relative to a coordinate system. For point-based systems, this is typically expressed as a set of n-dimensional coordinates within the coordinate system. For bodies, this is typically expressed by relating the translation of the origin of an object's local coordinate system with respect to the origin of an external reference coordinate system.

4.11. Location Model

A model that allows one to locate objects in one local reference frame relative to another reference frame.

4.12. Measurand

Physical parameter or a characteristic of a phenomenon subject to a measurement,whose value is described using a Measure (ISO 19103). Subset of determinand or observable.

OGC 20-082r4

4.13. Measure (noun)

Value described using a numeric amount with a scale or using a scalar reference system [ISO/TS 19103]. When used as a noun, measure is a synonym for physical quantity

4.14. **Measurement (noun)**

An observation whose result is a measure.

OGC 20-082r4

4.15. **Measurement (verb)**

An instance of a procedure to estimate the value of a natural phenomenon, typically involving an instrument or sensor. This is implemented as a dynamic feature type, which has a property containing the result of the measurement. The measurement feature also has a location, time, and reference to the method used to determine the value. A measurement feature effectively binds a value to a location and to a method or instrument.

4.16. **Muliplexed Data Stream**

A data stream that consists of disparate but well-defined data packets within the same stream.

4.17. **Observable, Observable Property (noun)**

A parameter or a characteristic of a phenomenon subject to observation. Synonym for determinand. A physical property of a phenomenon that can be observed and measured (e.g., temperature, gravitational force, position, chemical concentration, orientation, number-of-individuals, physical switch status, etc.), or a characteristic of one or more feature types, the value for which must be estimated by application of some procedure in an observation. It is thus a physical stimulus that can be sensed by a detector or created by an actuator.

Note 1 to entry: definition includes content from [OGC20-082r4].

4.18. **Observation**

Act of measuring or otherwise determining the value of a property.

ISO 19156:2011, definition 4.11

Note 1 to entry: The goal of an observation may be to measure, estimate or otherwise determine the value of a property.

4.19. Observation Procedure

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

ISO 19156:2011, definition 4.12

Note 1 to entry: In the context of the sensor web, an observation procedure is often composed of one or more sensors that transform a real world phenomenon into digital information, plus additional processing steps.

4.20. Observed Value

A value describing a natural phenomenon, which may use one of a variety of scales including nominal, ordinal, ratio and interval. The term is used regardless of whether the value is due to an instrumental observation, a subjective assignment or some other method of estimation or assignment.

OGC 20-082r4

4.21. Orientation

The rotational relationship of an object relative to an external coordinate system. Typically expressed by relating the rotation of an object's local coordinate axes relative to those axes of an external reference coordinate system.

4.22. Phenomenon

A physical state that can be observed and its properties measured.

4.23. **Physical System**

An aggregate model of a group or array of process components, which can include detectors, actuators, or sub-systems. A Physical System relates an Aggregate Process to the real world and therefore provides additional definitions regarding relative positions of its components and communication interfaces.

4.24. **Position**

The location and orientation of an object relative to an external coordinate system. For body-based systems (in lieu of point-based systems) is typically expressed by relating the object's local coordinate system to an external reference coordinate system. This definition is in contrast to some definitions (e.g., ISO 19107) which equate position to location.

4.25. **Process**

An operation that takes one or more inputs, and based on a set of parameters, and a methodology generates one or more outputs.

4.26. **Process Method**

Definition of the algorithm, behavior, and interface of a Process.

4.27. **Property**

Facet or attribute of an object referenced by a name.

EXAMPLE: Abby's car has the color red, where "color" is a property of the car instance, and "red" is the value of that property.

ISO 19143/2010

4.28. Reference Frame

A coordinate system by which the position (location and orientation) of an object can be referenced.

4.29. Result

An estimate of the value of some property generated by a known procedure.

OGC 20-082r4

4.30. Sample

A representative subset of the physical entity on which an observation is made.

4.31. Sensor

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

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

4.32. Sensor Model

In line with traditional definitions of the remote sensing community, a sensor model is a type of Location Model that allows one to georegister or co-register observations from a sensor (particularly remote sensors).

4.33. Sensor Data

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

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

4.34. Sensor-Related Data

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

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

4.35. (Sensor) Platform

An entity to which can be attached sensors or other platforms. A platform has an associated local coordinate reference frame that can be referenced relative to an external coordinate reference frame and to which the reference frames of attached sensors

4.36. Transducer

An entity that receives a signal as input and generates a modified signal as output. Includes detectors, actuators, and filters.

4.37. Value

A member of the value-space of a datatype. A value may use one of a variety of scales including nominal, ordinal, ratio and interval, spatial and temporal. Primitive datatypes may be combined to form aggregate datatypes with aggregate values, including vectors, tensors and images.

[ISO_11404]

6

CONVENTIONS

6 CONVENTIONS

This sections provides details and examples for any conventions used in the document. Examples of conventions are symbols, abbreviations, use of XML schema, or special notes regarding how to read the document.

6.1. Identifiers

The normative provisions in this standard are denoted by the URI

`http://www.opengis.net/spec/sensorML/3.0`

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

6.2. Abbreviated terms

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

- CRS: Coordinate Reference System
- DN: Digital Number
- ECEF: Earth-Centered Earth-Fixed
- ECI: Earth Centered Inertial
- GPS: Global Positioning System
- ISO: International Organization for Standardization
- MISB: Motion Imagery Standards Board
- OGC: Open Geospatial Consortium
- SAS: Sensor Alert Service
- SensorML: Sensor Model Language
- SI: Système International (International System of Units)
- SOS: Sensor Observation Service
- SPS: Sensor Planning Service

- SWE: Sensor Web Enablement
- TAI: Temps Atomique International (International Atomic Time)
- uom: Unit(s) of measure
- UCUM: Unified Code for Units of Measure
- UML: Unified Modeling Language
- UTC: Coordinated Universal Time
- XML: eXtensible Markup Language
- 1D: One Dimensional
- 2D: Two Dimensional
- 3D: Three Dimensional

6.3. UML notation

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

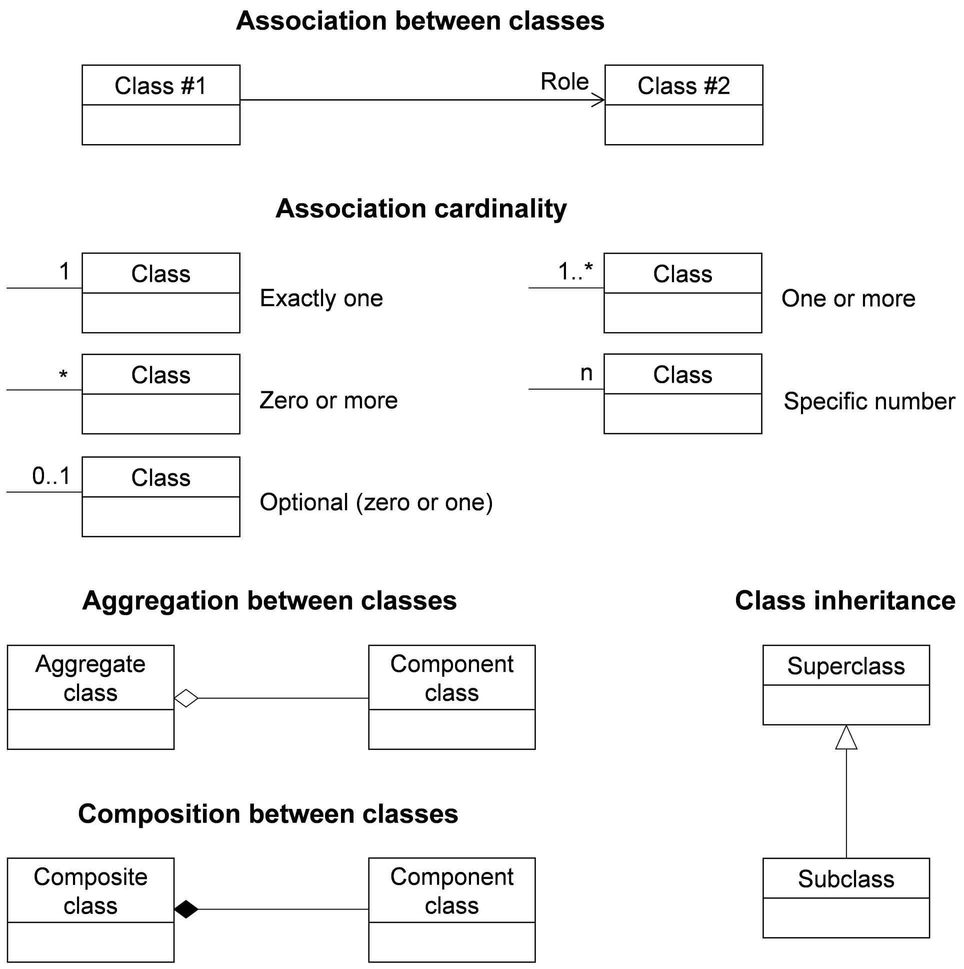


Figure 1 — UML Notation

7

REQUIREMENTS CLASS: CORE CONCEPTS (NORMATIVE CORE)

7 REQUIREMENTS CLASS: CORE CONCEPTS (NORMATIVE CORE)

REQUIREMENTS CLASS 1: CORE CONCEPTS	
IDENTIFIER	`/req/core`
TARGET TYPE	Derived Model, Encoding, and Software Implementation
CONFORMANCE CLASS	Conformance class A.1: `/conf/core`
NORMATIVE STATEMENTS	Requirement 1: `/req/core/concepts-used` Requirement 2: `/req/core/processes` Requirement 3: `/req/core/uniqueID` Requirement 4: `/req/core/metadata` Requirement 5: `/req/core/execution`

7.1. Introduction

In SensorML, all components are modeled as processes. This includes components normally viewed as hardware, such as detectors, actuators, and physical processors (which are viewed as physical components) and sensors and platforms (which are viewed as physical systems). All components are modeled as processes that receive input and through the application of an algorithm defined by a method and set parameter values, generate output. All such components can therefore participate in process networks (or aggregate processes). Aggregate processes are themselves processes with their own inputs, outputs, and parameters.

Hence, SensorML can be viewed as a specialized process description language with an emphasis on application to sensor data. Process descriptions in SensorML are agnostic of the environment in which they might be executed, or the protocol by which data is exchanged between process execution modules.

In order to support the use of SensorML within specialized applications (e.g., processing centers or image processing software), the SensorML models and encodings have been divided into several conformance classes. Thus, if one wishes to use SensorML for computation processes only, the software only needs to conform to the requirements for non-physical processes. Similarly, by only adhering to the Simple Process conformance class, a piece of software can describe internal processes using SensorML while supporting chaining of these processes in a proprietary way.

However, all derived model and encodings based on SensorML must implement the core concepts of SensorML, regardless of whether they deal strictly with non-physical computational processes or sensor systems.

REQUIREMENT 1	
IDENTIFIER	`/req/core/concepts-used`
INCLUDED IN	Requirements class 1: `/req/core`
STATEMENT	Any derived model or encoding shall correctly implement the modeling concepts defined in the core of this specification.

7.2. Process Definitions

In SensorML, all relevant components are modeled as processes, including both computation and physical processes (e.g., detectors, actuators, and sensor systems). Processes in SensorML are conceptually divided into two types: (1) those that are physical processes, such as detectors, actuators, and sensor systems, where information regarding their positions may be relevant, and (2) non-physical or "pure" processes which can be treated as merely mathematical operations or functions.

Example: Examples
For a process representing the standard linear equation, x would be the input, m and b the parameters, y the output, and the equation y = mx + b would define the methodology.

For a detector, the input would typically be a physical stimulus (or observable property), the parameters might include a calibration curve and other factors that affect the measurement, and the output would be a digital number representing some quantity representation of that observed property.

Fundamentally, a process is a physical or computational operation that may receive input and based on configurable parameters and a methodology, generate output.

Inputs and outputs may be digital numbers or physical stimuli (i.e., observable properties of the environment). Parameters can be variable or constant, but they don't typically vary at the same frequency as the input values. In essence, however, parameters can be viewed as just another input into the process that is either fixed or changes less frequently than inputs

A process can consist of a single atomic operation, or an explicitly defined network of operations (e.g., an aggregate process or system).

Any process must have a definable method of operation. In the case of an aggregate process or physical system, the explicit description of the process components and the flow of data between them will itself serve as the process methodology.

REQUIREMENT 2	
IDENTIFIER	`/req/core/processes`
INCLUDED IN	Requirements class 1: `/req/core`
STATEMENT	The core model for a process shall define inputs, outputs, parameters, and methodology of that process.

Any process description must provide a unique ID that can be used for discovery of that process and for retrieving the definition of that process.

REQUIREMENT 3	
IDENTIFIER	`/req/core/uniqueID`
INCLUDED IN	Requirements class 1: `/req/core`
STATEMENT	The core model for a process shall include a unique ID for distinguishing that process from all others

To be useful, the core process model shall include metadata about the process that aid in identification, discovery, and qualification of the process but do not themselves affect the execution of the process.

REQUIREMENT 4	
IDENTIFIER	`/req/core/metadata`
INCLUDED IN	Requirements class 1: `/req/core`
STATEMENT	The core model for a process shall include metadata that support identification, discovery, and qualification of the process.

REQUIREMENT 5	
IDENTIFIER	`/req/core/execution`
INCLUDED IN	Requirements class 1: `/req/core`

REQUIREMENT 5	
STATEMENT	The metadata descriptions for a process shall not be required for successful execution of that process. All information required for execution of a simple process shall be contained within the inputs, outputs, parameters, and methodology descriptions of the process.

Process definitions can support general representations of a process or a specific instance of a process.

Example: Examples
A general process for the linear equation would define the allowable inputs, outputs, and parameters. A specific instance of the process might define constant values for the parameters.

An example of a general physical process would be the manufacturer's description of the characteristics and configurable options for a particular model of a sensor (i.e., one that describes the common characteristics of all instances of that model of sensor). The description of a specific instance of that model of sensor would include information that is relevant to that particular instance of the sensor (e.g., serial number, owner's name, location, etc.).

8

UML CONCEPTUAL MODELS (NORMATIVE)

8 UML CONCEPTUAL MODELS (NORMATIVE)

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

8.1. Package Dependencies

The packages defined by the SensorML Model and their dependencies are shown in the figure below:

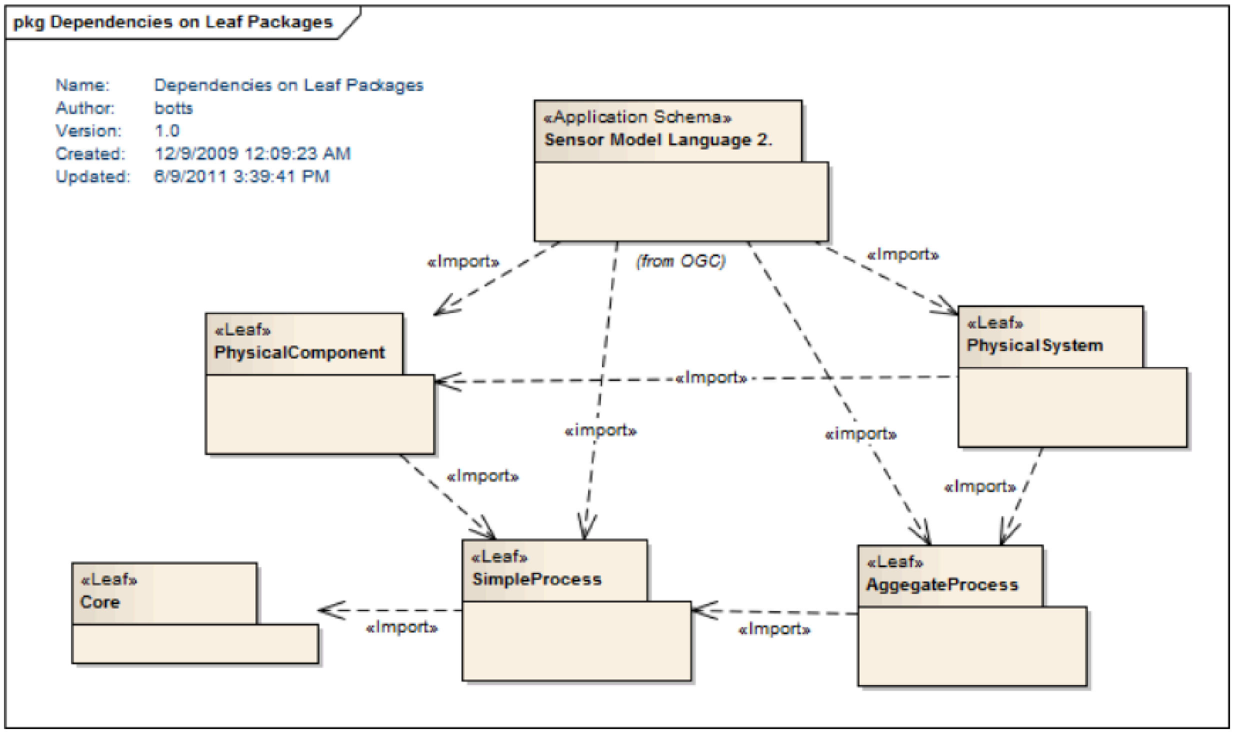

pkg Dependencies on Leaf Packages
Name: Dependencies on Leaf Packages
Author: botts
Version: 1.0
Created: 12/9/2009 12:09:23 AM
Updated: 6/9/2011 3:39:41 PM
«Application Schema» Sensor Model Language 2.
(from OGC)
«Leaf» PhysicalComponent
«Leaf» PhysicalSystem
«Leaf» SimpleProcess
«Leaf» AggegateProcess
«Leaf» Core
«Import»
«import»

Figure 2 — Internal Package Dependencies

SensorML also has dependencies on several external packages defined within other standards, ISO 19103, ISO 19108, ISO 19111, and ISO 19115, as described below.

8.1.1. Dependency on ISO TC 211 Models

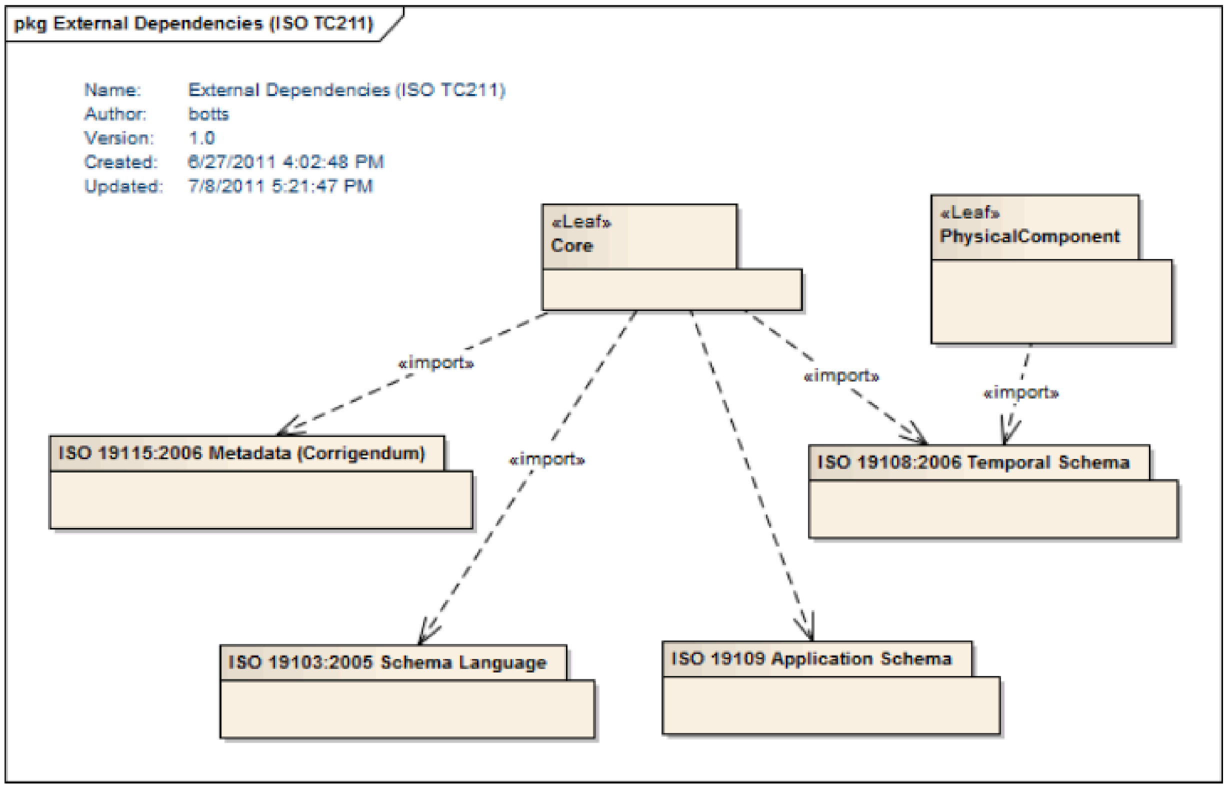
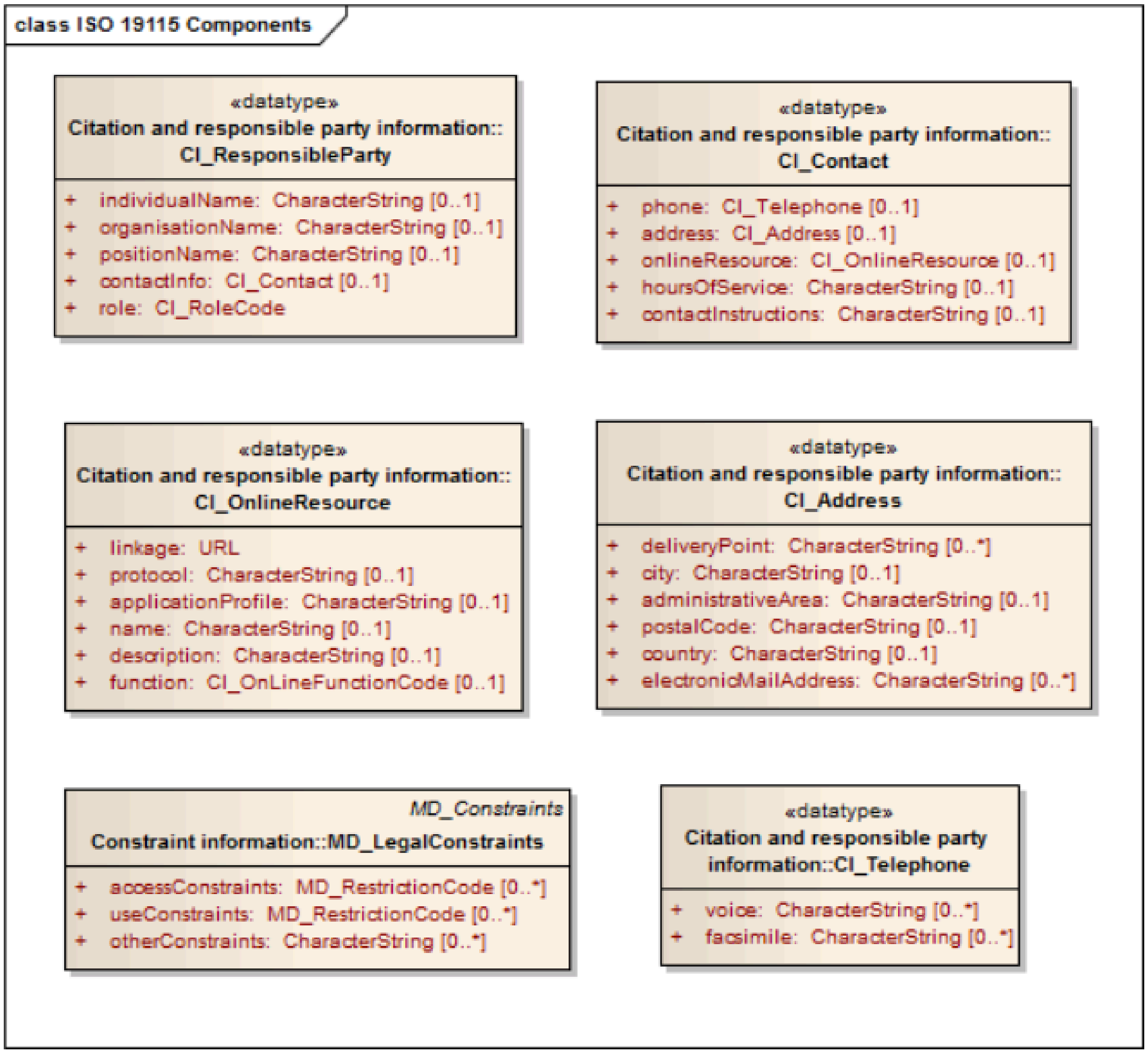

pkg External Dependencies (ISO TC211)
Name: External Dependencies (ISO TC211)
Author: botts
Version: 1.0
Created: 6/27/2011 4:02:48 PM
Updated: 7/8/2011 5:21:47 PM
«Leaf» Core
«Leaf» PhysicalComponent
«import»
«import»
«import»
«import»
ISO 19115:2006 Metadata (Corrigendum)
ISO 19108:2006 Temporal Schema
ISO 19103:2005 Schema Language
ISO 19109 Application Schema

Figure 3 — External Package Dependencies – ISO TC 211

The SensorML standard utilizes the ISO 19115 models for common metadata properties such as citations, online resources, responsible party, and constraints. While Version 1.0 of SensorML defined encoding based on the ISO 19115 models, this version utilizes these models directly.



class ISO 19115 Components
«datatype»
Citation and responsible party information::CI_ResponsibleParty
+ individualName: CharacterString [0..1]
+ organisationName: CharacterString [0..1]
+ positionName: CharacterString [0..1]
+ contactInfo: CI_Contact [0..1]
+ role: CI_RoleCode
«datatype»
Citation and responsible party information::CI_Contact
+ phone: CI_Telephone [0..1]
+ address: CI_Address [0..1]
+ onlineResource: CI_OnlineResource [0..1]
+ hoursOfService: CharacterString [0..1]
+ contactInstructions: CharacterString [0..1]
«datatype»
Citation and responsible party information::CI_OnlineResource
+ linkage: URL
+ protocol: CharacterString [0..1]
+ applicationProfile: CharacterString [0..1]
+ name: CharacterString [0..1]
+ description: CharacterString [0..1]
+ function: CI_OnLineFunctionCode [0..1]
«datatype»
Citation and responsible party information::CI_Address
+ deliveryPoint: CharacterString [0..*]
+ city: CharacterString [0..1]
+ administrativeArea: CharacterString [0..1]
+ postalCode: CharacterString [0..1]
+ country: CharacterString [0..1]
+ electronicMailAddress: CharacterString [0..*]
MD_Constraints
Constraint information::MD_LegalConstraints
+ accessConstraints: MD_RestrictionCode [0..*]
+ useConstraints: MD_RestrictionCode [0..*]
+ otherConstraints: CharacterString [0..*]
«datatype»
Citation and responsible party information::CI_Telephone
+ voice: CharacterString [0..*]
+ facsimile: CharacterString [0..*]

Figure 4 — ISO 19115 Models for dependent classes.

8.1.2. Dependency on SWE Common Data Models

In particular, SensorML is heavily dependent on the SWE Common Data Model standard for defining inputs, outputs, and parameters, as well as for specifying characteristics, capabilities, interfaces, and event properties. The SWE Common Data Models, which were originally defined within the version 1.0 SensorML specification, are since version 2.0 defined as a separate specification and are utilized throughout the SWE family of encoding and web service specifications.

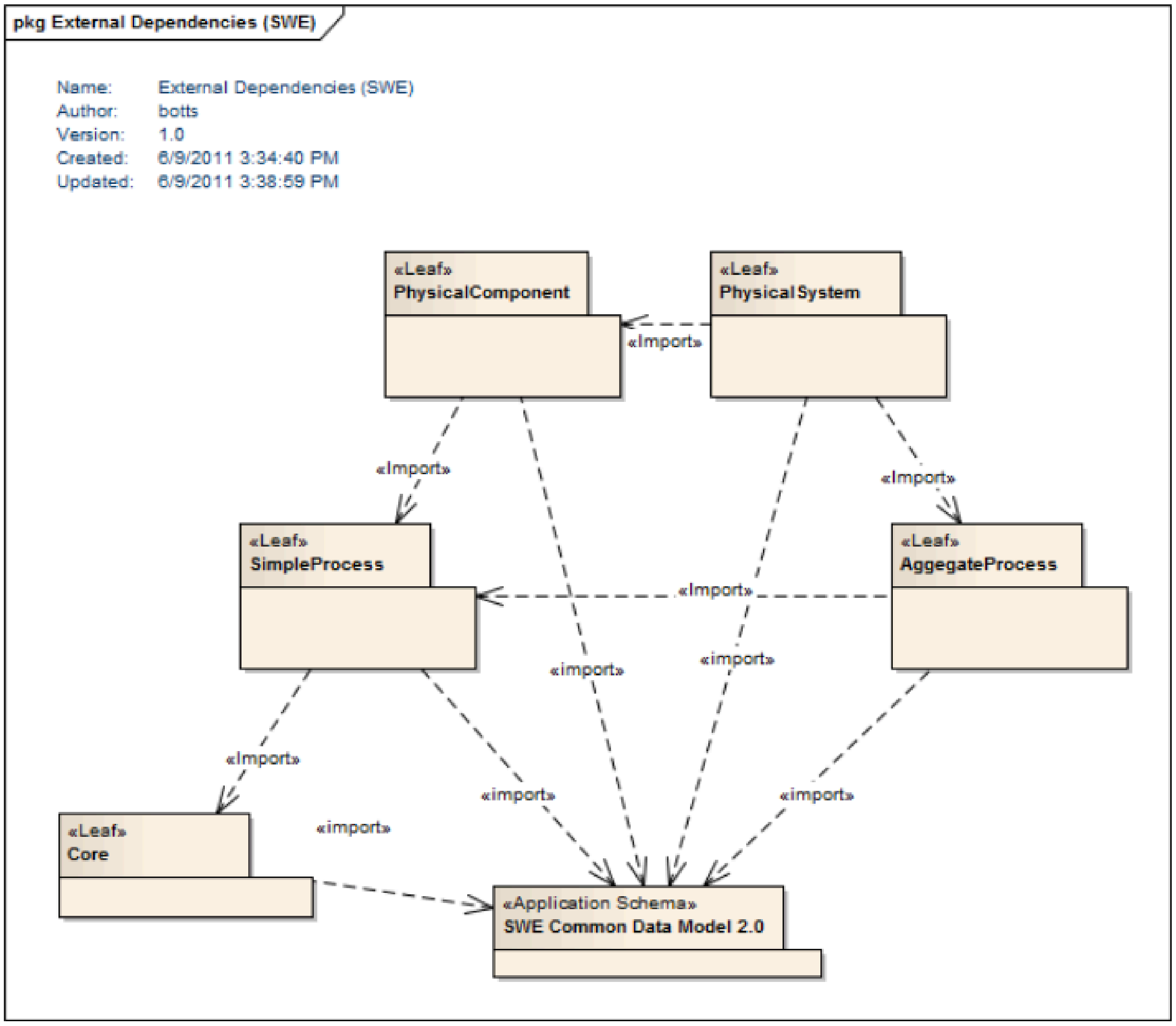

pkg External Dependencies (SWE)
Name: External Dependencies (SWE)
Author: botts
Version: 1.0
Created: 6/9/2011 3:34:40 PM
Updated: 6/9/2011 3:38:59 PM
«Leaf» PhysicalComponent
«Leaf» PhysicalSystem
«Leaf» SimpleProcess
«Leaf» AggegateProcess
«Leaf» Core
«Application Schema» SWE Common Data Model 2.0
«Import»
«import»

Figure 5 — External Package Dependencies - SWE Common Data

The SWE Common specification provides a flexible yet robust means of defining data types and data values, including support for simple data types such as Quantity, Boolean, Category, Count, Text, and Time, as well as aggregate data such as DataRecord, DataArray, Vector, and Matrix. Additionally, SWE Common supports the concept of DataChoice, which will be utilized by SensorML for providing multiplexed messages in data streams and configurable options for processes and physical systems.

The data models in SWE Common provide additional properties than are provided by basic data types, including for example, units of measure (uom), quality indications, allowable constraints, significant digit counts, and in particular, the meaning and semantics of a data component. Both simple and aggregate data components in SWE Common allow for unambiguous definition of that data component through a resolvable link to an online dictionary or ontology. The definition of the SWE Common Data Models can be found in OGC 08-094r1.

The main objective of SWE Common Data Models is to achieve interoperability, first at the syntactic level, and later at the semantic level (by using ontologies and 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.

SensorML depends heavily on the AbstractDataComponent element defined in SWE Common. This element serves as the base component from which all relevant data types in SWE Common are derived, including Quantity, Count, Category, Boolean, Text, DataRecord, DataArray, Vector, Matrix, and DataChoice. AbstractDataComponent thus serves as a substitution group that any of these data types can satisfy. AbstractSWEIdentifiable will serve as the basis for the ObservableProperty element defined in this specification (Clause 8.2.1).

The model for the SWE Common AbstractDataComponent is given in the figure below:

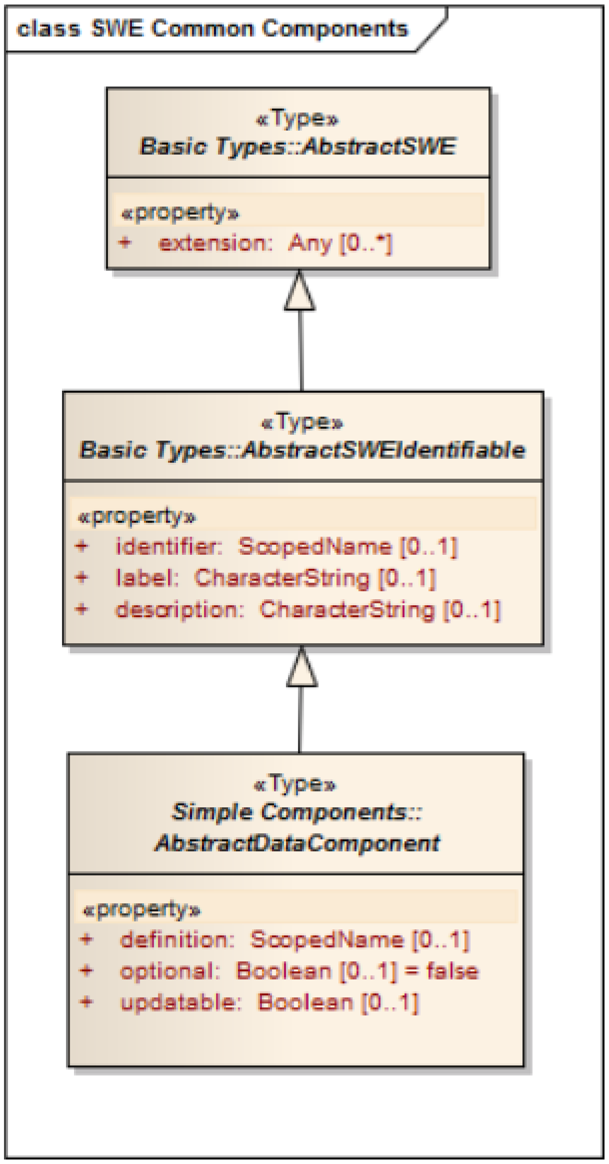

class SWE Common Components
«Type»
Basic Types::AbstractSWE
«property»
+ extension: Any [0..*]
«Type»
Basic Types::AbstractSWEIdentifiable
«property»
+ identifier: ScopedName [0..1]
+ label: CharacterString [0..1]
+ description: CharacterString [0..1]
«Type»
Simple Components::
AbstractDataComponent
«property»
+ definition: ScopedName [0..1]
+ optional: Boolean [0..1] = false
+ updatable: Boolean [0..1]

Figure 6 — Models for dependent SWE Common AbstractDataComponent class.

8.1.3. Relationship to Observations and Measurements (O&M)

Conceptual models for Observations and Measurements are provided by ISO 19156, which also provides models for sampling feature types. XML Schema encodings of these models are provided by the OGC Observations and Measurements XML Implementation Document (OGC 10-025). The model for Observation defines a procedure of type AbstractFeature which references or describes the origin of the observation (i.e., how the observation came to be).

SensorML has an association to the O&M models but no direct dependencies on them. The result of a SensorML process is typically considered to be an observation result if it is measuring or deriving some value of a physical property or phenomenon. Thus, the output values described in SensorML and resulting from a sensor or process may be packaged in an O&M Observation object or provided as a SWE Common DataStream. Inversely, the procedure property within an Observation instance may reference a SensorML description of the measurement process.

8.1.4. Relationship to OGC API — Processes

The OGC API — Processes — Part 1: Core standard (OGC 18-062r2) supports the wrapping of computational tasks into executable processes that can be offered by a server through a Web API and be invoked by a client application. The standard specifies a processing interface to communicate over a RESTful protocol using JSON encodings.

Even though the the standard recommends implementations to support its own encoding for process descriptions, the OGC Process Description, SensorML Simple Processes and Aggregate Processes are a suitable substitution. While the OGC Process Description allows for the definition of inputs and outputs using JSON Schema, a SensorML process description would allow for a definition using SWE Common.

Physical Components and Physical Systems are also suitable surrogates for the OGC Process Description would for example allow OGC API — Processes to becoming a tasking interface for real world sensors.

8.2. Requirements Class: Core Abstract Process

REQUIREMENTS CLASS 2: CORE ABSTRACT PROCESS	
IDENTIFIER	`/req/model/coreProcess`
TARGET TYPE	Derived Encoding or Software Implementation
CONFORMANCE CLASS	Conformance class A.2: `/conf/model/coreProcess`

REQUIREMENTS CLASS 2: CORE ABSTRACT PROCESS	
PREREQUISITES	Requirements class 1: `/req/core` http://www.opengis.net/spec/SWE/3.0/req/uml-record-components ISO 19115:2003/Cor.1:2006 (All Metadata)
NORMATIVE STATEMENTS	Requirement 6: `/req/model/coreProcess/dependency-core` Requirement 7: `/req/model/coreProcess/package-fully-implemented` Requirement 8: `/req/model/coreProcess/uniqueID` Requirement 9: `/req/model/coreProcess/extensionIndependence` Requirement 10: `/req/model/coreProcess/extensionRestrictions` Requirement 11: `/req/model/coreProcess/SWE-Common-dependency1` Requirement 12: `/req/model/coreProcess/aggregateData` Requirement 13: `/req/model/coreProcess/typeOf` Requirement 14: `/req/model/coreProcess/simpleInheritance` Requirement 15: `/req/model/coreProcess/configuration` Requirement 16: `/req/model/coreProcess/SWE-Common-dependency2`

All major classes in SensorML are based on a process model, as presented in the core concepts. Processes are features as defined in ISO 19109:2006. SensorML also supports interoperable discovery, identification, and qualification of these processes through the definition of a standard collection of metadata.

REQUIREMENT 6	
IDENTIFIER	`/req/model/coreProcess/dependency-core`
INCLUDED IN	Requirements class 2: `/req/model/coreProcess`
STATEMENT	An encoding or software passing the “Core Abstract Process” model conformance class shall first pass the “Core Concepts” conformance test class.

REQUIREMENT 7	
IDENTIFIER	`/req/model/coreProcess/package-fully-implemented`
INCLUDED IN	Requirements class 2: `/req/model/coreProcess`
STATEMENT	An encoding or software shall correctly implement all UML classes defined in the “Core” package and described in this section

8.2.1. ObservableProperty

An ObservableProperty is a physical property of a phenomenon that can be observed and measured (e.g., temperature, gravitational force, position, chemical concentration, orientation, number-of-individuals, physical switch status, etc.), or a characteristic of one or more feature types, the value for which must be estimated by application of some procedure in an observation. It is thus a physical stimulus that can be sensed by a detector or created by an actuator.

Example: Examples
The ObservableProperty element allows one to reference a measurable property of a phenomenon or feature for detector inputs or actuator outputs. For example, the temperature of the atmosphere is an ObservableProperty. Before measurement, it is simply a property of the atmosphere that can be defined and measured. After measurement by a detector, the temperature may be represented as a Quantity with units of measure, a value, and an indication of our degree of confidence in the measurement.

ObservableProperty is derived as a concrete instance of the SWE Common AbstractSWEIdentifiable and adds the definition property to this model. It will be used as a potential input (e.g., for detectors), output (e.g., for actuators), and for parameters (e.g., for a sensor whose measurement varies with fluctuations of atmospheric pressure on a diaphragm).

In ObservableProperty the phenomenon property will be defined by reference using the definition attribute. The definition attribute value will reference a property defined within a dictionary or ontology. An ObservableProperty may also include a name and a description. However, unlike the simple data types in SWE Common, an ObservableProperty does NOT include the properties uom, quality, or constraints, since these are typically characteristics of the measuring procedure and not properties of the observable phenomenon itself.

8.2.2. DescribedObject

As shown in the UML model below, the DescribedObject class provides a specific set of metadata for all process classes in SensorML. The DescribedObject provides a unique ID, and support for a label and a description. The unique ID in SensorML will be supported by a single uniqueId property.

REQUIREMENT 8	
IDENTIFIER	`/req/model/coreProcess/uniqueID`
INCLUDED IN	Requirements class 2: `/req/model/coreProcess`
STATEMENT	A single, required uniqueId property shall be used to provide a unique ID for the DescribedObject.

Metadata about each process is essential to supporting identification, discovery, and qualification of the process. Metadata is provided by the base class, DescribedObject, from which AbstractProcess is derived. While these metadata may provide relevant information to understand quality of output from the process, the values of properties within the DescribedObject should not be required for execution of the process. The model for the DescribedObject is shown in Figure 7, while the models for the individual property values are provided in either Figure 8 or in the ISO 19115 models in Figure 9.

The DescribedObject includes several descriptive properties that support rapid discovery (keywords, identification, and classification), constraints (validTime, securityConstraints, legalConstraints), qualification (characteristics and capabilities), references (contacts and documentation), and history. These are each grouped in lists, which provide for easy separation and parsing of these properties.

8.2.2.1. Extension Property

The extension property allows one to add domain or community-specific content to a DescribedObject instance. This might include, for example, security taggings, vendor or community-specific metadata, or information encoded in other models or schema. Extension properties must exist in a separate namespace and SensorML-compliant software is not required to understand or utilize the information contained within the extension property.

The constraints on the extension property include: a) the extension model must be defined in a separate namespace, b) the information added by the extension model must not be required for execution of the process, and c) SensorML-compliant parsers may parse and utilize the information within these extensions but they are not required to do so in order to be compliant to the SensorML standard.

REQUIREMENT 9	
IDENTIFIER	`/req/model/coreProcess/extensionIndependence`
INCLUDED IN	Requirements class 2: `/req/model/coreProcess`
STATEMENT	Models inside of the extension property must exist within a namespace other than SensorML.

REQUIREMENT 10	
IDENTIFIER	`/req/model/coreProcess/extensionRestrictions`
INCLUDED IN	Requirements class 2: `/req/model/coreProcess`
STATEMENT	Information provided inside of the extension property must not be required for execution of the process and shall not alter the execution of the process.

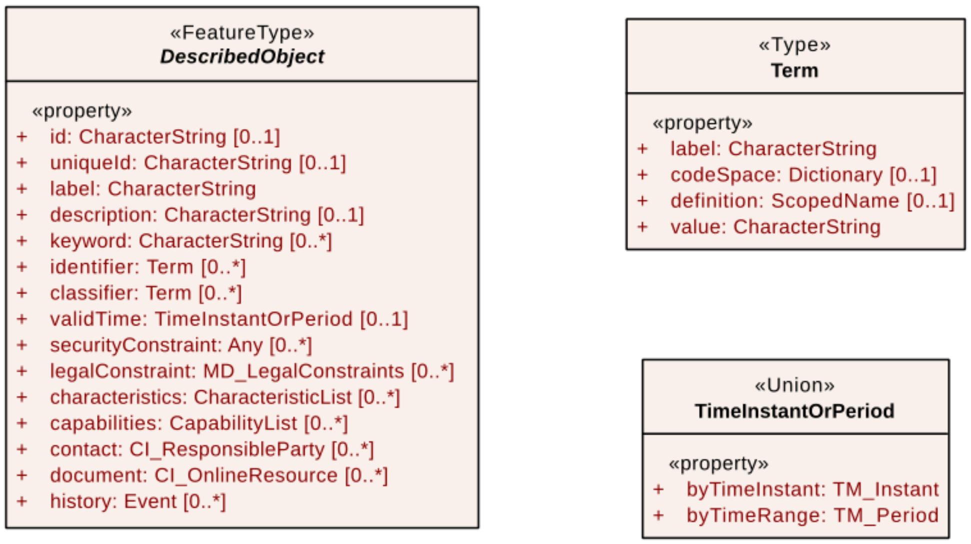

«FeatureType»
DescribedObject
«property»
+ id: CharacterString [0..1]
+ uniqueId: CharacterString [0..1]
+ label: CharacterString
+ description: CharacterString [0..1]
+ keyword: CharacterString [0..*]
+ identifier: Term [0..*]
+ classifier: Term [0..*]
+ validTime: TimeInstantOrPeriod [0..1]
+ securityConstraint: Any [0..*]
+ legalConstraint: MD_LegalConstraints [0..*]
+ characteristics: CharacteristicList [0..*]
+ capabilities: CapabilityList [0..*]
+ contact: CI_ResponsibleParty [0..*]
+ document: CI_OnlineResource [0..*]
+ history: Event [0..*]
«Type»
Term
«property»
+ label: CharacterString
+ codeSpace: Dictionary [0..1]
+ definition: ScopedName [0..1]
+ value: CharacterString
«Union»
TimeInstantOrPeriod
«property»
+ byTimeInstant: TM_Instant
+ byTimeRange: TM_Period

Figure 7 — DescribedObject with Metadata Properties

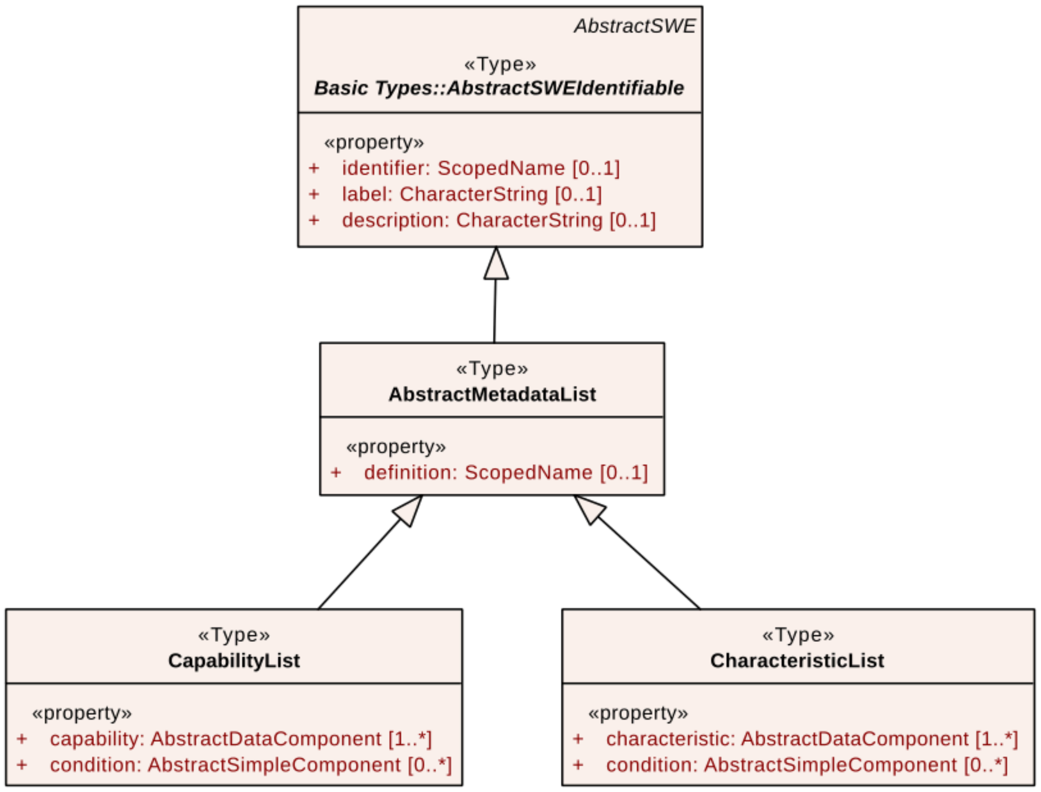

AbstractSWE
«Type»
Basic Types::AbstractSWEIdentifiable
«property»
+ identifier: ScopedName [0..1]
+ label: CharacterString [0..1]
+ description: CharacterString [0..1]
«Type»
AbstractMetadataList
«property»
+ definition: ScopedName [0..1]
«Type»
CapabilityList
«property»
+ capability: AbstractDataComponent [1..*]
+ condition: AbstractSimpleComponent [0..*]
«Type»
CharacteristicList
«property»
+ characteristic: AbstractDataComponent [1..*]
+ condition: AbstractSimpleComponent [0..*]

Figure 8 — Models for Metadata Elements

8.2.2.2. Keywords

Keywords provide a simple means of discovery using short tokens that may be recognized by the general audience or specific communities. Keywords are unqualified terms in that they are not necessarily required to be related to a specific codespace or ontology, as are classifiers and identifiers.

8.2.2.3. Identifiers

The identifier property takes a Term as its value. The Term has a definition attribute that specifies in this case the type of identifier, while the codeSpace attribute specifies that the value of the identifier is according to the rules or enumerations of a particular authority.

Example: Examples
An identifier with a definition of "http://sensors.ws/def/tailNumber" might take "N291PV" as its value based on the codespace of a US Air Force rules dictionary. Other possible definitions for identifiers might include, for example, shortName, longName, acronym, missionID, processorID, serialNumber, manufacturerID, or partNumber.

The identification properties should be considered as information suitable for the discovery applications.

8.2.2.4. Classifiers

The classifier property provides a list of possible classifiers that might aid in the rapid discovery or organization of processes, sensors, or sensor systems. The classifier properties should be considered as information suitable for the discovery and categorization applications.

Example: Examples
Definitions for a classifier Term might include, for instance, sensorType, observableType, processType, intendedApplication, or missionType.

8.2.2.5. Security Constraints

The model for specification of security constraints shall be based on external security models, such as the Security Banner Marking model of the Intelligence Community Information Security Marking (IC ISM) Standard. The securityConstraints property takes an any value which allows various communities and countries to utilize their standard encoding for security tags. This security constraint is for the overall document. As will be discussed in the JSON encoding, extension points provided with SWE Common Data elements will allow security tagging for individual properties or property aggregates.

Example: Examples
One can specify the overall security classification of the entire document using the Intelligence Community Information Security Banner Marking (IC ISM) standard or using ISO 19115 MD_Constraints. For tagging individual sections in the document, the SensorML standard allows for security tagging of properties using an extension property, as describe in later sections of the standard.

8.2.2.6. Valid Time Constraint

The validTime property indicates the time instance or time range over which this process description is valid. Time constraints are important for processes in which parameter values or operation modes may change with time, or instrument deployment times change.

Example: Examples
Several SensorML documents can exist for the same sensor or system description but with different validity periods. This allows for capturing the configuration of a sensor at different times and, along with the history section, is the basis for maintaining history of the sensor's description. Alternately, parameter values can be provided as a time-tagged series of values accounting for changes.

8.2.3. Legal Constraint

The legalConstraints property is based on ISO 19115 and specifies whether such legal and ethical considerations as privacy acts, intellectual property rights, copyrights, or scientific publication ethics apply to the content of the process description and its use.

8.2.4. Capabilities

The capabilities property is intended for the definition of properties that further qualify the input or output of the process, component, or system for the purpose of discovery. These properties are defined using one or more SWE Common DataRecord elements.

Once a user has identified candidate sensors or processes based on the classifiers described above, the capabilities parameters might prove useful for further filtering of processes or sensor system during this discovery stage. Thus, the capabilities properties should be considered as information suitable for the discovery process.

Example: Examples
A particular remote sensor on a satellite might measure radiation between a certain spectral range (e.g., 700 to 900 nanometers) at a particular ground resolution (e.g., 5 meter), and with a typical spatial repeat period (e.g., 3.25 – 4.3 days). Alternatively, a particular process might have certain quality constraints. Any process may have certain limits (e.g., operational and survivable limits), based on physical or mathematical conditions. These properties do affect the output of the process and should be considered as capabilities.

8.2.5. Characteristics

A physical or non-physical process may have characteristics that may not directly qualify the output. These properties are defined using one or more SWE Common DataRecord elements.

Example: Examples

A component may have certain physical measurements such as dimensions and weight and be constructed of a particular material. A component may have particular power demands, or anticipated lifetime. These are characteristics of the component that may not directly affect the output of the component or system.

The characteristics properties may or may not be considered as information suitable for the discovery process.

8.2.6. Contacts

Contact information can provide access to manufacturers, system experts, equipment owners, or any other persons responsible in some way for design, deployment, maintenance, or additional information regarding the DescribedObject. The contact property within the ContactList takes the ISO 19115 classes CI_ResponsibleParty as its values.

8.2.7. Documentation

Documentation can be provided which provides further clarification about the DescribedObject. This might include technical manuals, manufacturer brochures, journal references, or theoretical-basis documents. The DocumentList document property takes the ISO 19115 CI_OnlineResource as its value.

8.2.8. History

Within SensorML, the history of a process can be provided through a collection of Event objects. These are provided within an EventList that serves as the value of the history property. Events might for instance, specify calibration or maintenance history of a sensor, changes to an algorithm or parameter within a computational process, or deployment and maintenance events.

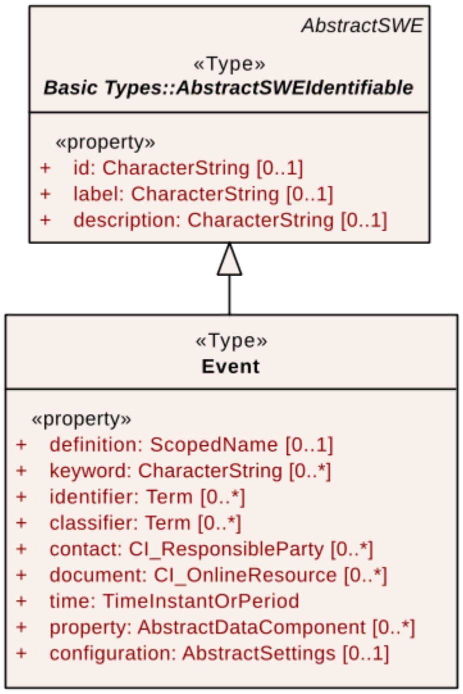

AbstractSWE
«Type»
Basic Types::AbstractSWEIdentifiable
«property»
+ id: CharacterString [0..1]
+ label: CharacterString [0..1]
+ description: CharacterString [0..1]
«Type»
Event
«property»
+ definition: ScopedName [0..1]
+ keyword: CharacterString [0..*]
+ identifier: Term [0..*]
+ classifier: Term [0..*]
+ contact: CI_ResponsibleParty [0..*]
+ document: CI_OnlineResource [0..*]
+ time: TimeInstantOrPeriod
+ property: AbstractDataComponent [0..*]
+ configuration: AbstractSettings [0..1]

Figure 9 — Model for history events

8.2.9. AbstractProcess

As discussed in the Core Concepts, the major elements of SensorML are modeled as physical and non-physical processes. All SensorML process elements shall derive from AbstractProcess,shown in Figure 10. The class AbstractProcess itself derives from the DescribedObject class and thus inherits a wide range of optional metadata supporting discovery, identification, and qualification and an option for domain and community-specific extensions. In addition to the metadata provided by DescribedObject, the AbstractProcess includes the properties of inputs, outputs, and parameters, as required by the process model defined in the Core Concepts, as well as the properties typeOf, featureOfInterest, configuration, and modes which will be discussed below.

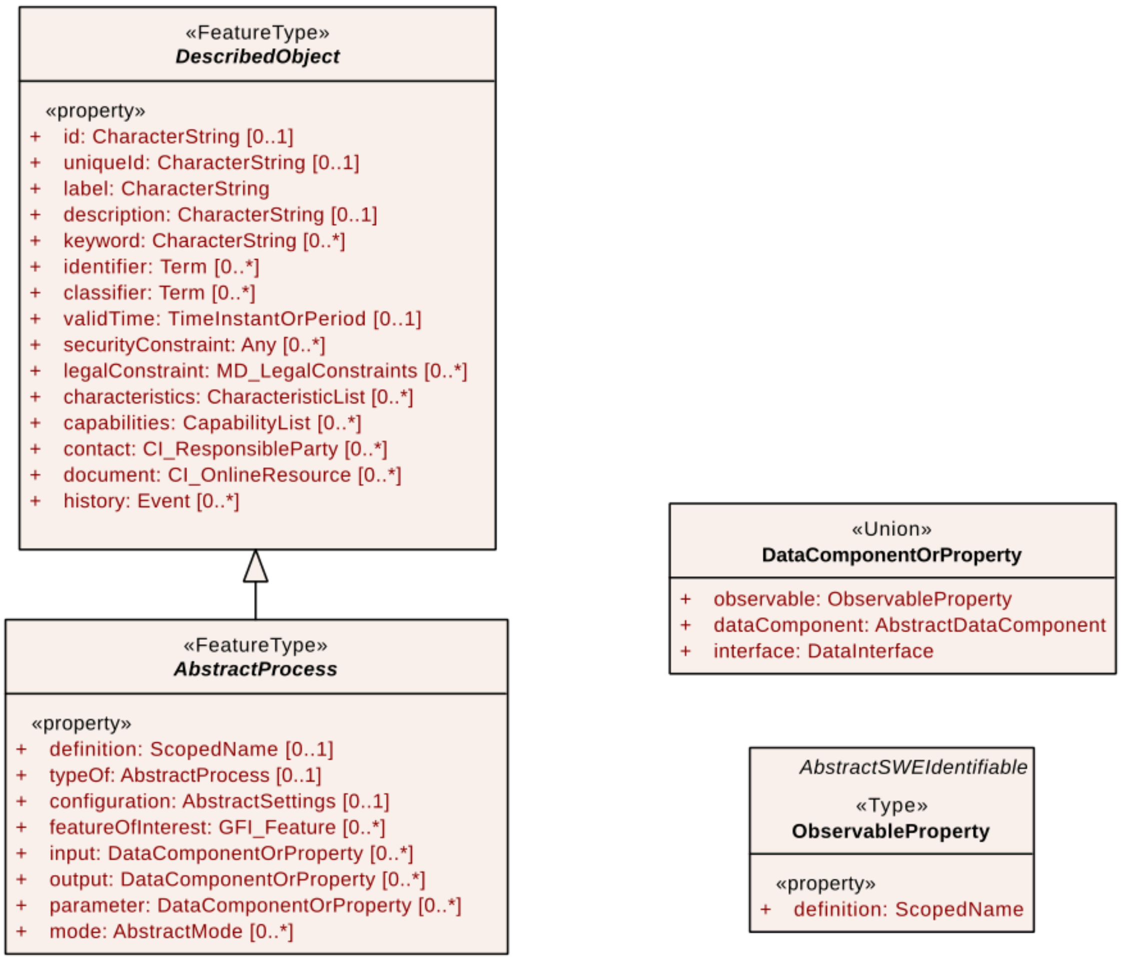

«FeatureType»
DescribedObject
«property»
+ id: CharacterString [0..1]
+ uniqueId: CharacterString [0..1]
+ label: CharacterString
+ description: CharacterString [0..1]
+ keyword: CharacterString [0..*]
+ identifier: Term [0..*]
+ classifier: Term [0..*]
+ validTime: TimeInstantOrPeriod [0..1]
+ securityConstraint: Any [0..*]
+ legalConstraint: MD_LegalConstraints [0..*]
+ characteristics: CharacteristicList [0..*]
+ capabilities: CapabilityList [0..*]
+ contact: CI_ResponsibleParty [0..*]
+ document: CI_OnlineResource [0..*]
+ history: Event [0..*]
«FeatureType»
AbstractProcess
«property»
+ definition: ScopedName [0..1]
+ typeOf: AbstractProcess [0..1]
+ configuration: AbstractSettings [0..1]
+ featureOfInterest: GFI_Feature [0..*]
+ input: DataComponentOrProperty [0..*]
+ output: DataComponentOrProperty [0..*]
+ parameter: DataComponentOrProperty [0..*]
+ mode: AbstractMode [0..*]
«Union»
DataComponentOrProperty
+ observable: ObservableProperty
+ dataComponent: AbstractDataComponent
+ interface: DataInterface
AbstractSWEIdentifiable
«Type»
ObservableProperty
«property»
+ definition: ScopedName

Figure 10 — UML models for DescribedObject and AbstractProcess

8.2.9.1. Inputs, Outputs, and Parameters

As discussed in the Core Concepts, any process can have inputs, outputs, and parameters. Processes typically receive input and based on the parameter settings and methodology, generate output. Some processes, such as detectors, receive physical stimulus as input and generate digital numbers as output. In such cases, the input would be represented as an ObservableProperty, and the output as a DataComponent (e.g., a Quantity). If this output is encoded and accessible directly, then the output can be represented as a DataInterface.

Example: Examples
A digital thermometer is stimulated by an observable property of the environment (temperature), which is modelled as its input (ObservableProperty), and outputs a digital number (Quantity) that represents a measure of that property.

Thus, an AbstractProcess model supports the inputs, outputs, and parameters properties in conformance with the Core Concepts. These properties can accept ObservableProperty or SWE Common elements AbstractDataComponent or DataStream as their values. Classes derived from

AbstractDataComponent include Quantity, Count, Category, Boolean, Text, and Time, as well as ranges and aggregates of these simple data types.

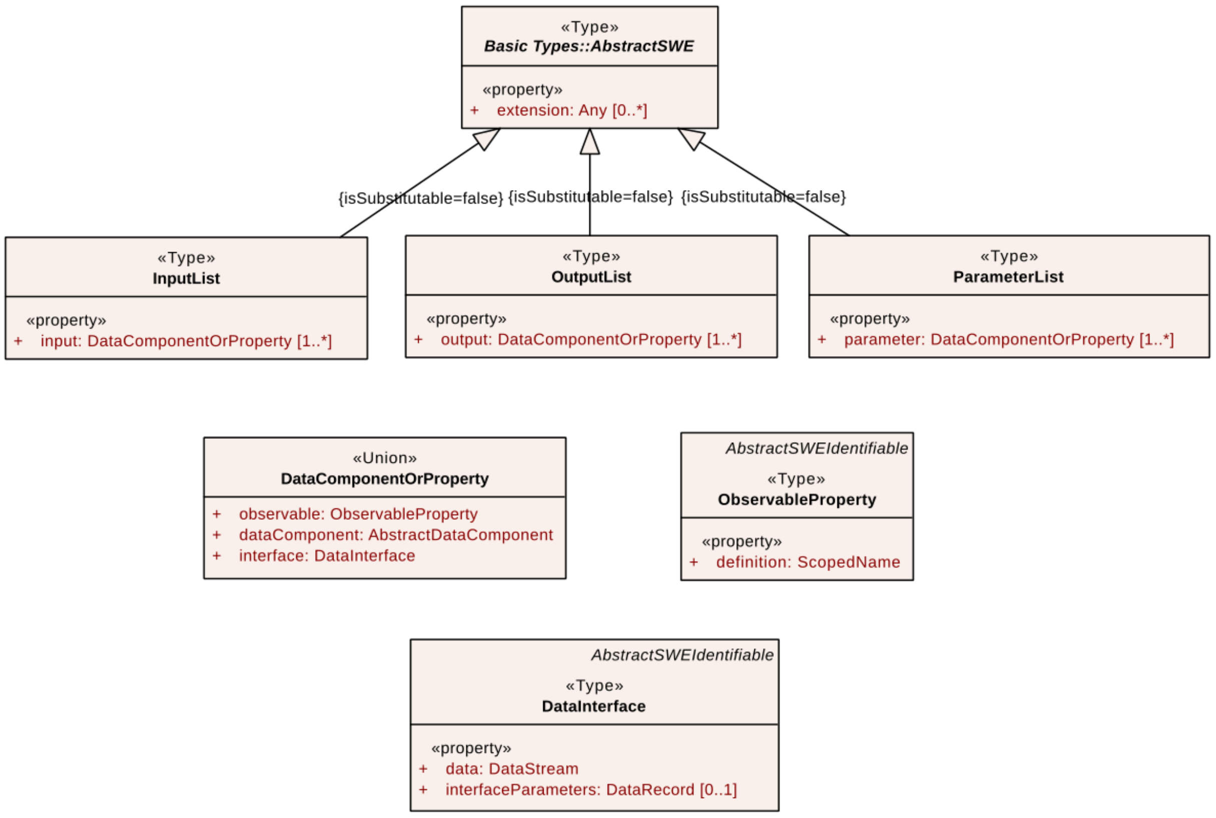


Figure 11 — UML models for process inputs, outputs, and parameters

The core process model will utilize the SWE Common Data Models for defining inputs, outputs, and parameters, as well as for other metadata properties. SensorML models are required to support the SWE Common Data Model up to the Block Components Requirements Class, but many instances of SensorML will find ALL conformance levels of SWE Common Data to be useful, including binary encodings.

REQUIREMENT 11	
IDENTIFIER	`/req/model/coreProcess/SWE-Common-dependency1`
INCLUDED IN	Requirements class 2: `/req/model/coreProcess`
STATEMENT	Any derived model or encoding for process shall utilize ObservableProperty or SWE Common Data Components as values for inputs, outputs, and parameters, and shall at a minimum conform to the SWE Common Data “Block Components Package” class (http://www.opengis.net/spec/SWE/3.0/req/uml-block-components).

The input, output, or parameters of many processes include multiple values, possibly of different data types, that are tightly related to one another. Sometimes referred to as tuples or records, these data aggregates can consist of values that are perhaps meaningless without the other associated values (e.g., the coordinates within a spatial reference system), or provide a more complete understanding because of their association with one another (e.g., a set of measured values taken by a sensor at a given time). Such data shall be modelled using the aggregate data types defined by the SWE Common Data standard.

Example: Examples
The location of a dynamic object can be specified through the aggregate values of time, latitude, longitude, and altitude. In such cases, the expression of one of the values separate from the others is meaningless or less complete than the expression of these values as a set or aggregate. These four values should be encapsulated in a Vector data type that also identifies the reference frame in which the latitude, longitude and altitude coordinates are expressed.

Weather stations often express a set of measurements of the atmosphere as a single record that might include for instance temperature, pressure, relative humidity, cloudiness, wind speed, and wind direction. These would be considered a tuple of values that provides a more complete picture of the environment at a particular time. This tuple should be modeled as a DataRecord with 7 fields (one for each measured parameters listed above + one time stamp) to indicate that the sampling time applies to all observable values included in the record.

REQUIREMENT 12	
IDENTIFIER	`/req/model/coreProcess/aggregateData`
INCLUDED IN	Requirements class 2: `/req/model/coreProcess`
STATEMENT	Multiple input, output, and parameter values that are tightly related with one another shall be modelled as a SWE Common Data aggregate data type.

8.2.9.2. Feature of Interest

Most sensors and many non-physical processes have been deployed or implemented with a focus on one or more features of interest. Within SensorML, the primary purpose of including a FeatureOfInterest property for AbstractProcess is to support discovery as well as to further clarify the intended purpose of the physical or non-physical process.

Example: Examples
The features of interest of an installed web camera might include a particular building, a particular street, or a general area of observation surrounding the camera. Features of interest for other sensors might include the Gulf of Mexico, a particular drilling well, the atmosphere surrounding a particular weather station, a particular patient, or a particular automobile. Features of interest for a model or other process might include a particular river basin, a particular toxic plume release, or a particular metropolitan area.

8.2.9.3. Inheritance, Extension, and Configuration

SensorML supports the concepts of inheritance, extension, and configuration. In other words, generalized base processes can be described in SensorML and then that description can be augmented or further constrained by one or more separate descriptions. Thus, a single, generalized description of a physical or non-physical process can serve as a basis for one or many more specific process descriptions. This provides support for more simple and concise process descriptions while also providing the ability for the user or application to "drill down" to greater and greater detail as desired.

The inheritance model will support two cases:

- Simple inheritance – the specific process description provides only additional information to the description of the general process, without modifying or restricting any property values of the general process; and
- Configuration – the specific process description is able to set or restrict property values within the allowable range provided by the general process description, as well as provide additional information.

The key to inheritance, extension, and configuration of a process lies in the typeOf property, by which a specific process can reference its more general base process. The typeOf property takes as its value any process model derived from AbstractProcess. This will be "by-reference-only" meaning that the value must be in the form of a resolvable link to another process instance.

REQUIREMENT 13	
IDENTIFIER	`/req/model/coreProcess/typeOf`
INCLUDED IN	Requirements class 2: `/req/model/coreProcess`
STATEMENT	A process that is a specific instance of another process shall reference the more general process through its typeOf property. The value of the typeOf property shall be a resolvable link to an instance of a process derived from AbstractProcess.

8.2.9.3.1. Simple Inheritance

In the simple inheritance model, a process (referred to as the "specific process") inherits and augments information from another process (referred to as the "general process").

Example: Examples
An Original Equipment Manufacturer (OEM) provides a description of a particular model of their sensor that would define inputs, outputs, and parameters, as well as perhaps capabilities, characteristics, manufacturer contact information and documentation relevant to that model. Thousands of sensors of this model type may of course be manufactured and sold by the OEM.

When one purchases and deploys an instance of that model of sensor, the owner can then reference the OEM's description of the model and provide additional information that's specific to his specific instance of the sensor. Additional information might include, for example, serial number, owner's contact information, the sensor's location, calibration data, and the interface description for accessing the data.

The simple inheritance model is fully supported in the Core Process conformance class and will be supported solely through the use of the typeOf property within the specific process. The typeOf property within the specific process will reference the general process through a resolvable reference.

REQUIREMENT 14	
IDENTIFIER	`/req/model/coreProcess/simpleInheritance`
INCLUDED IN	Requirements class 2: `/req/model/coreProcess`
STATEMENT	A process instance that references another process through the typeOf property, but does not include the configuration property, shall inherit properties of the referenced process through simple inheritance. The complete description of that process is thus the addition of information from both process descriptions.

8.2.9.3.2. Support for Configurable Processes

A configurable process is one that includes options or choices that can be selected, restricted, or enabled during deployment, operation, or execution of that process.

Example: Examples
An Original Equipment Manufacturer (OEM) can provide a description of a particular model of their sensor that would define inputs, outputs, and parameters, as well as perhaps capabilities, characteristics, manufacturer contact information and documentation relevant to that model. In addition, the OEM enables an individual instance of that model of sensor to be configured by providing options for setting parameter values, setting modes, or choosing a particular interface. Thousands of sensors of this model type may of course be manufactured and sold by the OEM.

When one purchases and deploys an instance of that model of sensor, the owner can then reference the OEM's description of the model and provide additional information that's specific to that particular instance of the sensor. In addition, the owner can configure the sensor by setting values, selecting modes, and enabling particular interfaces. These settings would be provided in the instance description.

The configuration model will utilize both the typeOf and configuration properties. The typeOf property references the more general process as with simple inheritance, while the configuration property provides a means to further restrict the options and allowed values for the specific process. The configuration property in the AbstractProcess takes an AbstractSettings class as its value.

REQUIREMENT 15	
IDENTIFIER	`/req/model/coreProcess/configuration`
INCLUDED IN	Requirements class 2: `/req/model/coreProcess`
STATEMENT	A process instance that references another process through the typeOf property, and further restricts options or allowed values provided in the referenced process, shall specify those restrictions through the configuration property.

A concrete implementation of a Settings class will be provided in a later Conformance Clause.

8.2.10. SWE Common Data Types

Many properties in the DescribedObject and AbstractProcess classes described above are of type AbstractDataComponent as defined in the SWE Common Data Model standard. This data type is used for defining inputs, outputs and parameters, as well as for other metadata properties.

This requirements class only mandates the support of the “Simple Components” and “Record Components” as defined in the SWE Common Data Model standard. These include the scalar data types Boolean, Text, Count, Quantity, Category, Time and their range equivalents, as well as DataRecord and Vector.

REQUIREMENT 16	
IDENTIFIER	`/req/model/coreProcess/SWE-Common-dependency2`
INCLUDED IN	Requirements class 2: `/req/model/coreProcess`
STATEMENT	Contents of all properties of type AbstractDataComponents shall pass the SWE Common Data Model “Records Components Package” conformance test class.

However, many implementations of SensorML will find ALL conformance levels of the SWE Common Data Model to be useful, including arrays, choices and encodings. An implementation claiming support for more than the record components can pass the “Processes with Advanced Data Types” conformance test class of this standard.

8.3. Requirements Class: Simple Process

REQUIREMENTS CLASS 3: SIMPLE PROCESS	
IDENTIFIER	`/req/model/simpleProcess`
TARGET TYPE	Derived Encoding or Software Implementation
CONFORMANCE CLASS	Conformance class A.3: `/conf/model/simpleProcess`
PREREQUISITES	Requirements class 2: `/req/model/coreProcess` ISO 19115:2003/Cor.1:2006 (All Metadata)
NORMATIVE STATEMENTS	Requirement 17: `/req/model/simpleProcess/dependency-core` Requirement 18: `/req/model/simpleProcess/package-fully-implemented` Requirement 19: `/req/model/simpleProcess/definition` Requirement 20: `/req/model/simpleProcess/method`

A simple process is derived from abstract process model, as presented in Clause 8.2.

REQUIREMENT 17	
IDENTIFIER	`/req/model/simpleProcess/dependency-core`
INCLUDED IN	Requirements class 3: `/req/model/simpleProcess`
STATEMENT	An encoding or software passing the “Simple Process” model conformance class shall first pass the “Abstract Process” requirements test class.

REQUIREMENT 18	
IDENTIFIER	`/req/model/simpleProcess/package-fully-implemented`
INCLUDED IN	Requirements class 3: `/req/model/simpleProcess`
STATEMENT	The encoding or software shall correctly implement all UML classes defined in the “SimpleProcess” package described in this section.

8.3.1. Simple Process Definition

A simple process is a process that, for whatever reason, is considered indivisible. That is, there is no intent to further divide the process description into an aggregation of sub-processes. While the process method may describe several steps within the algorithm, the actual execution of this process is expected to occur as a single modular unit.

Often simple processes are computational processes that can be executed with an associated piece of software. Simple processes are often one component of a physical or non-physical aggregate process.

REQUIREMENT 19	
IDENTIFIER	`/req/model/simpleProcess/definition`
INCLUDED IN	Requirements class 3: `/req/model/simpleProcess`
STATEMENT	A process shall be modeled a “Simple Process” if it provides a processing function with well-defined inputs and outputs, if there is no intent to further divide the process description into sub-process components, and if knowledge of its physical location is of no importance.

The SimpleProcess model, as shown in Figure 12, is a concrete instantiation of the AbstractProcess model. The SimpleProcess requires a method description.

REQUIREMENT 20	
IDENTIFIER	`/req/model/simpleProcess/method`
INCLUDED IN	Requirements class 3: `/req/model/simpleProcess`
STATEMENT	An encoding or software implementation of the SimpleProcess class shall support the definition of the method.

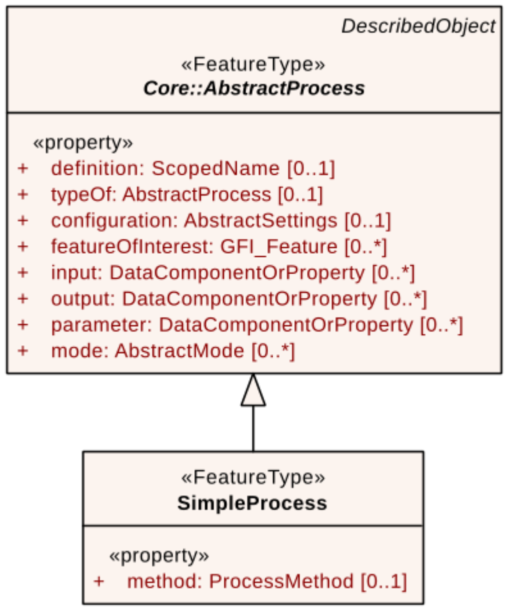

DescribedObject
«FeatureType»
Core::AbstractProcess
«property»
+ definition: ScopedName [0..1]
+ typeOf: AbstractProcess [0..1]
+ configuration: AbstractSettings [0..1]
+ featureOfInterest: GFI_Feature [0..*]
+ input: DataComponentOrProperty [0..*]
+ output: DataComponentOrProperty [0..*]
+ parameter: DataComponentOrProperty [0..*]
+ mode: AbstractMode [0..*]
«FeatureType»
SimpleProcess
«property»
+ method: ProcessMethod [0..1]

Figure 12 — Model for Simple Process

Example: Examples
A process computing a simple mathematical function such as sine, cosine or square root is usually modeled as a SimpleProcess instance. However, even more complex processes can be modeled this way if there is no intent to break down the implementation of the process into sub-processes.

8.3.2. Process Method Definition

The ProcessMethod provides a description of the methodology used by the process to execute and generate output based on the input and parameter values. This includes a textual description, as well as an optional description of the algorithm in an appropriate format (e.g., mathML) and optional references to particular executable implementations.

The ProcessMethod definition should be sufficient to allow one to understand how input values are converted to output values, given a particular set of parameter values, and be able to write software that is capable of executing this process.

A ProcessMethod description may be protected by security or legal constraints, which would purposely prevent access to the method description as well as restrict knowledge of the methodology to authorized personnel only. However, regardless of access restrictions, a ProcessMethod should always be able to be referenced and identified by a unique identifier.

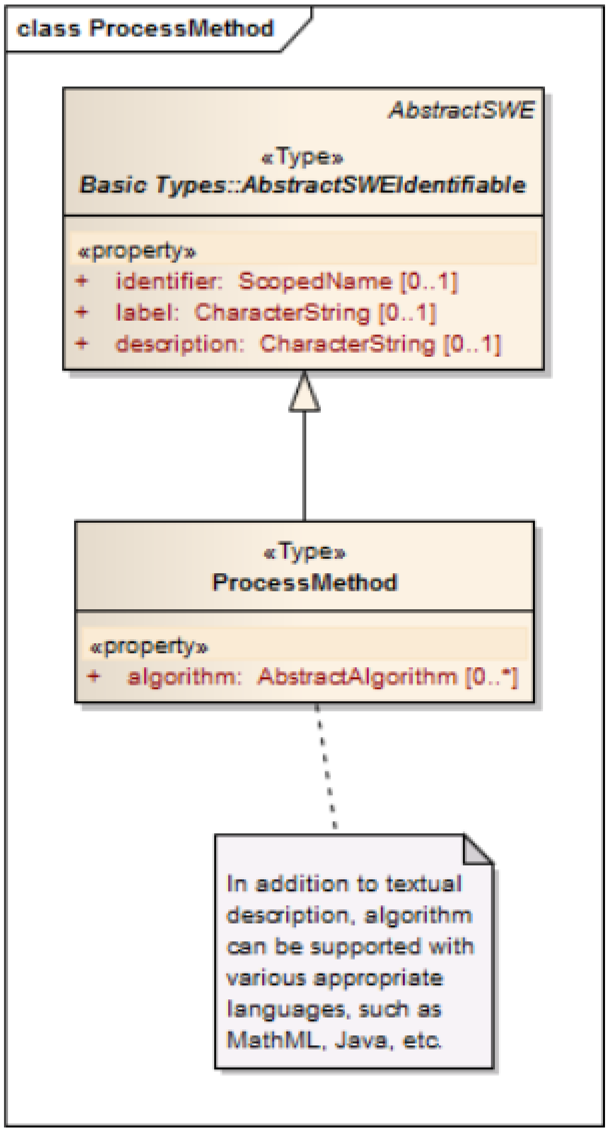

class ProcessMethod
AbstractSWE
«Type»
Basic Types::AbstractSWEIdentifiable
«property»
+ identifier: ScopedName [0..1]
+ label: CharacterString [0..1]
+ description: CharacterString [0..1]
«Type»
ProcessMethod
«property»
+ algorithm: AbstractAlgorithm [0..*]
In addition to textual description, algorithm can be supported with various appropriate languages, such as MathML, Java, etc.

Figure 13 — Model for ProcessMethod

8.4. Requirements Class: Aggregate Process

REQUIREMENTS CLASS 4: AGGREGATE PROCESS	
IDENTIFIER	`/req/model/aggregateProcess`
TARGET TYPE	Derived Encoding or Software Implementation

REQUIREMENTS CLASS 4: AGGREGATE PROCESS	
CONFORMANCE CLASS	Conformance class A.4: `/conf/model/aggregateProcess`
PREREQUISITE	Requirements class 3: `/req/model/simpleProcess`
NORMATIVE STATEMENTS	Requirement 21: `/req/model/aggregateProcess/dependency-core` Requirement 22: `/req/model/aggregateProcess/package-fully-implemented` Requirement 23: `/req/model/aggregateProcess/definition` Requirement 24: `/req/model/aggregateProcess/components`

An aggregate process is derived from abstract process model, as presented in Clause 8.2.

REQUIREMENT 21	
IDENTIFIER	`/req/model/aggregateProcess/dependency-core`
INCLUDED IN	Requirements class 4: `/req/model/aggregateProcess`
STATEMENT	An encoding or software passing the "Aggregate Process" conformance test class shall first pass the "Simple Process" conformance test class.

REQUIREMENT 22	
IDENTIFIER	`/req/model/aggregateProcess/package-fully-implemented`
INCLUDED IN	Requirements class 4: `/req/model/aggregateProcess`
STATEMENT	The encoding or software shall correctly implement all UML classes defined in the "Aggregate Process" package described in this section.

8.4.1. Aggregate Process Definition

An aggregate process is a collection of autonomous component processes with an explicit mapping of the data flow between these processes. Components of an aggregate process can be simple processes (i.e., atomic) or be aggregate process themselves. Aggregate processes can include both physical and non-physical (i.e., logical) components.

REQUIREMENT 23	
IDENTIFIER	`/req/model/aggregateProcess/definition`
INCLUDED IN	Requirements class 4: `/req/model/aggregateProcess`
STATEMENT	A process shall be modeled as an “aggregate process” if it provides a processing function with well-defined inputs and outputs, if there is intent to further divide the process description into sub-processes, and if knowledge of its physical location is of no importance.

REQUIREMENT 24	
IDENTIFIER	`/req/model/aggregateProcess/components`
INCLUDED IN	Requirements class 4: `/req/model/aggregateProcess`
STATEMENT	An encoding or software implementation of the AggregateProcess class shall support the inclusion of one or more component processes and a means for explicitly specifying data flow between these components.

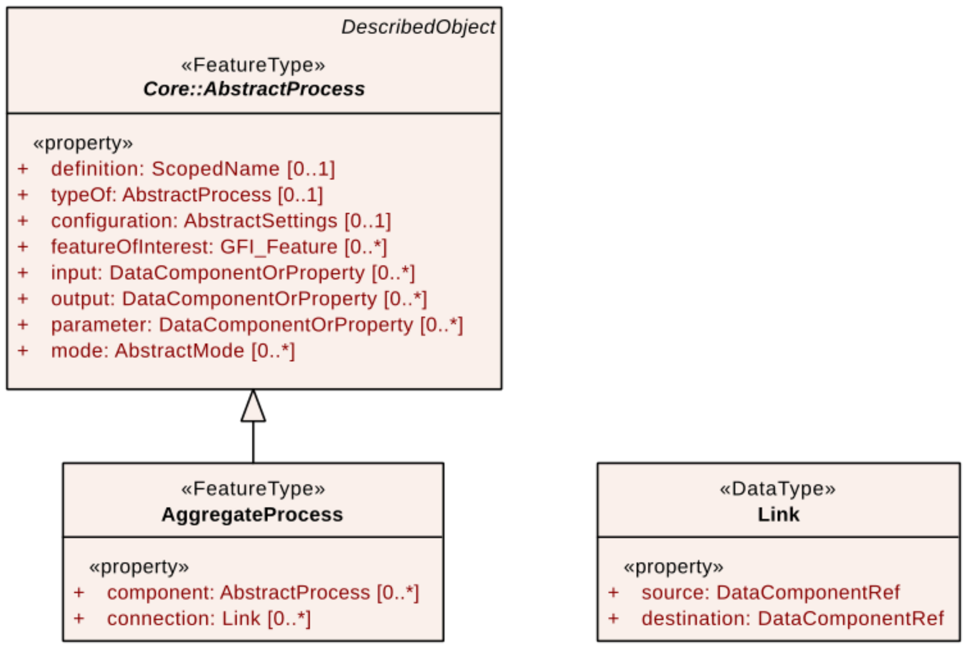
In SensorML, an aggregate process is agnostic to the execution engine that may perform the actual execution of individual sub-processes and manage the execution sequencing and the flow of data between the components. Also, while it is possible in SensorML to more explicitly define the data encoding if desired by using the encoding specifications defined in the SWE Common Data Specification, SensorML is typically agnostic to the protocol and format of data flowing between logical processes.

This provides significant flexibility as to where and how a SensorML-defined aggregate process is executed. While the ProcessMethod explicitly defines the algorithm for executing an atomic process, the actual execution of that algorithm and the management of data flow between processes can be handled by any software system able to parse a SensorML-defined aggregate process and sequence the execution of the processes.

A SensorML-defined process component or aggregate process can be executed through web services, within the CPU of a laptop, mobile device, or supercomputer, or a mix of these. Furthermore, a SensorML-defined aggregate process can be executed wherever desired, be it at a large data or computation center, within a visualization and analysis client on a laptop, or on-board a sensor or actuator system. Thus, SensorML provides the choice to either bring the process to the data or bring the data to the process.

The model for AggregateProcess is shown in the figure below. AggregateProcess extends the AbstractProcess model and adds one or more process components and explicit linking of data flow between these components. The component property takes any component derived from AbstractProcess as its value. Component process descriptions can be provided inline or by reference.

The derivation from AbstractProcess means that an AggregateProcess instance itself has its own inputs, outputs, and parameters, as well as identification and possible metadata.



DescribedObject
«FeatureType»
Core::AbstractProcess
«property»
+ definition: ScopedName [0..1]
+ typeOf: AbstractProcess [0..1]
+ configuration: AbstractSettings [0..1]
+ featureOfInterest: GFI_Feature [0..*]
+ input: DataComponentOrProperty [0..*]
+ output: DataComponentOrProperty [0..*]
+ parameter: DataComponentOrProperty [0..*]
+ mode: AbstractMode [0..*]
«FeatureType»
AggregateProcess
«property»
+ component: AbstractProcess [0..*]
+ connection: Link [0..*]
«DataType»
Link
«property»
+ source: DataComponentRef
+ destination: DataComponentRef

Figure 14 — Model for Aggregate Process

8.5. Requirements Class: Physical Component

REQUIREMENTS CLASS 5: PHYSICAL COMPONENT	
IDENTIFIER	`/req/model/physicalComponent`
TARGET TYPE	Derived Encoding or Software Implementation
CONFORMANCE CLASS	Conformance class A.5: `/conf/model/physicalComponent`
PREREQUISITES	Requirements class 2: `/req/model/coreProcess` http://www.opengis.net/spec/GeoPose/1.0/req/basic-ypr http://www.opengis.net/spec/GeoPose/1.0/req/basic-quaternion
NORMATIVE STATEMENTS	Requirement 25: `/req/model/physicalComponent/package-fully-implemented` Requirement 26: `/req/model/physicalComponent/dependency-core` Requirement 27: `/req/model/physicalComponent/byPointOrLocation` Requirement 28: `/req/model/physicalComponent/byPosition` Requirement 29: `/req/model/physicalComponent/byTrajectory` Requirement 30: `/req/model/physicalComponent/byProcess`

REQUIREMENTS CLASS 5: PHYSICAL COMPONENT
Requirement 31: `/req/model/physicalComponent/definition`

In the context of SensorML, physical processes represent real processing devices whose spatio-temporal position is important. Physical processes include detectors, actuators, sensor systems, and actuator systems. Such processes typically, but not always, involve interactions between a real-world domain (or environment) and a digital domain.

Example: Examples
A detector or sensor system typically senses an environmental stimulus and provides a digital number representing the measure of a property of that environment (e.g., temperature). Likewise, an actuator receives a digital number and based on its values causes an action in the real-world environment. Both devices interact with the real world and their position is usually of importance to the end-user. These should usually be modelled as physical components.

Because physical processes typically interact with the real-world environment, the position (location and orientation), as well as perhaps the dynamic state (velocity and acceleration), are usually of importance. We wish to either measure an observable property at a particular location in the environment or we wish to affect a physical action at a particular place in the environment. Thus, the position where the physical process measures or acts becomes important.

REQUIREMENT 25	
IDENTIFIER	`/req/model/physicalComponent/package-fully-implemented`
INCLUDED IN	Requirements class 5: `/req/model/physicalComponent`
STATEMENT	The encoding or software shall correctly implement all UML classes defined in the “Physical Component” package described in this section

8.5.1. Abstract Physical Process Defined

The AbstractPhysicalProcess model is derived from AbstractProcess and thus includes the metadata and properties of a core process. Additionally, AbstractPhysicalProcess supports additional properties that allow one to define spatial and temporal coordinates for the physical process device.

REQUIREMENT 26	
IDENTIFIER	`/req/model/physicalComponent/dependency-core`
INCLUDED IN	Requirements class 5: `/req/model/physicalComponent`

REQUIREMENT 26	
STATEMENT	An encoding or software passing the “Physical Component” conformance class shall first pass the “Core Abstract Process” conformance test class.

The model for AbstractPhysicalProcess is shown in Figure 15 below. The additional properties of the AbstractPhysicalProcess will be discussed in subsequent clauses.

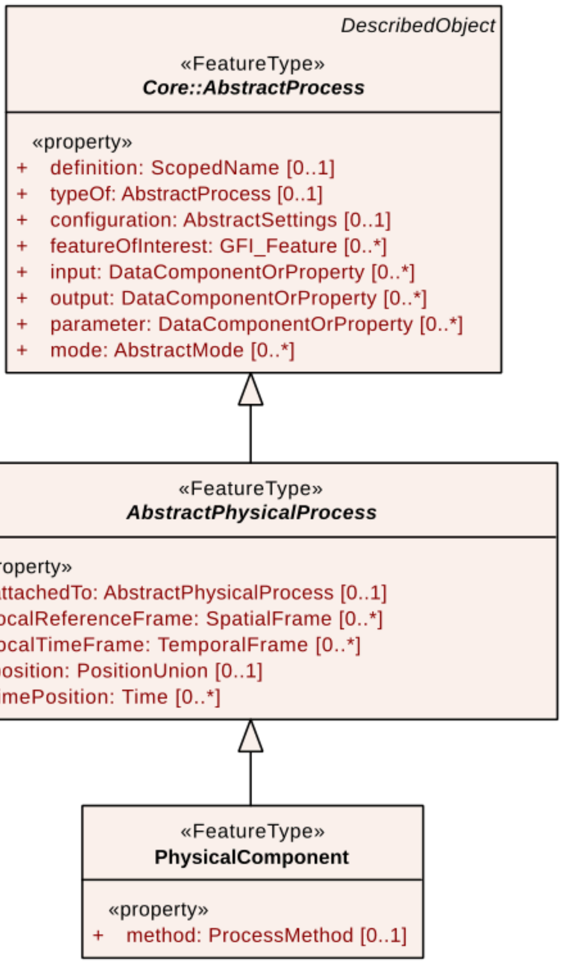

DescribedObject
«FeatureType»
Core::AbstractProcess
«property»
+ definition: ScopedName [0..1]
+ typeOf: AbstractProcess [0..1]
+ configuration: AbstractSettings [0..1]
+ featureOfInterest: GFI_Feature [0..*]
+ input: DataComponentOrProperty [0..*]
+ output: DataComponentOrProperty [0..*]
+ parameter: DataComponentOrProperty [0..*]
+ mode: AbstractMode [0..*]
«FeatureType»
AbstractPhysicalProcess
«property»
+ attachedTo: AbstractPhysicalProcess [0..1]
+ localReferenceFrame: SpatialFrame [0..*]
+ localTimeFrame: TemporalFrame [0..*]
+ position: PositionUnion [0..1]
+ timePosition: Time [0..*]
«FeatureType»
PhysicalComponent
«property»
+ method: ProcessMethod [0..1]

Figure 15 — Model for Physical Process Component

8.5.1.1. attachedTo Property

A physical process (“child process”) may be attached to another physical process (“parent process”) such that the movement of the parent process affects the position of the child process. The attachedTo property provides a reference from the attached process to the process to which it is attached.

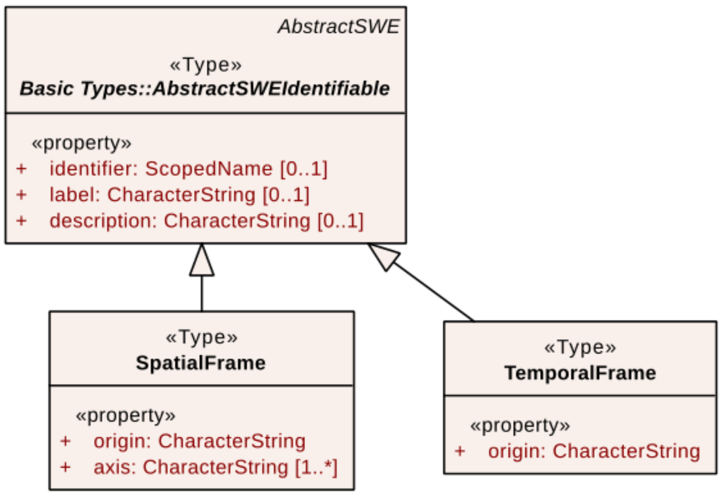
Example: Examples
A video camera is attached to a gimbal that allows rotation of the camera to view a 360° area surrounding the camera. In such a case, the camera is said to be attached to the gimbal. Both are

physical processes (the camera, a sensor; the gimbal, an actuator). The video camera description should thus use the attachedTo property to reference the gimbal description.

8.5.1.2. Position and Spatial Reference Frames

In this standard, the position or dynamic state of a physical object is defined as a relationship of the reference frame of the object to some external reference frame. SensorML allows for the definition of direct orthogonal (i.e., Cartesian) reference frames that are assumed to be attached to the physical component where they are described.

A reference frame is defined by its origin and its axes, which are described relative to the physical object itself using natural language and are not relative to any relationship of the object to some external frame. The relationship of this object’s reference frame to an external reference frame is defined by the position or dynamic state of the object. The models for reference frames and spatial position are provided in Figure 16. The origin of an airplane’s spatial reference frame can be defined as the being at the center of the Inertial Navigation Unit main gyro. The axes can be defined by the following statements: “X is along the symmetric axis of the airplane’s fuselage from the gyro to the nose of the airplane (along the platform roll axis of the airplane), Z is perpendicular to X and toward the belly of the airplane (along the yaw axis of the aircraft), and Y is Z cross X (in the direction of the right wing and along the pitch axis of the airplane)”. The location of this aircraft can then be given as the spatial relationship of the origin of this reference frame to some external reference frame (e.g., Earth-Centered-Earth-Fixed XYZ or latitude-longitude-altitude). Likewise, the orientation of the aircraft can be specified as the angular relationship of the axes of its reference frame to the axes of some external reference frame (e.g., ECEF or North-East-Down).


AbstractSWE
«Type»
Basic Types::AbstractSWEIdentifiable
«property»
+ identifier: ScopedName [0..1]
+ label: CharacterString [0..1]
+ description: CharacterString [0..1]
«Type»
SpatialFrame
«property»
+ origin: CharacterString
+ axis: CharacterString [1..*]
«Type»
TemporalFrame
«property»
+ origin: CharacterString

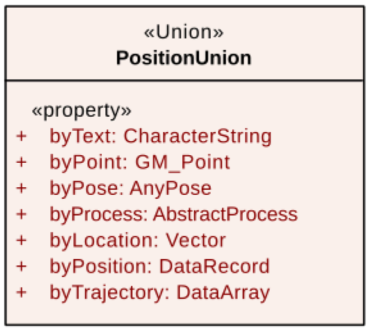


Figure 16 — Models for SpatialFrame and PositionUnion

In this standard, position is defined as the combination of location and orientation. Location is the llinear displacement (translation) of the origin of the object's spatial reference frame relative to the origin of some external reference frame (which must be designated). The orientation of an object is the angular relationship between the axes of the object's reference axes to those of some external reference frame. The dynamic state of an object can include its time-tagged location, orientation, linear velocity, angular velocity, and higher-order derivatives when required (e.g., linear and angular acceleration, jerk, etc.).

An external reference frame can be another object's reference frame (e.g., the reference frame of a ship) or a geographic reference frame (e.g., WGS84 latitude-longitude-altitude).

The PositionUnion class provides various means of specifying the location, position, or dynamic state of an object. These will be described in more detail in the appropriate JSON encoding section, but the following rules apply to the SensorML models.

REQUIREMENT 27	
IDENTIFIER	`/req/model/physicalComponent/byPointOrLocation`
INCLUDED IN	Requirements class 5: `/req/model/physicalComponent`
STATEMENT	Specification of position “byPoint” or “byLocation” shall specify the location of the origin of the object’s reference frame relative to the origin of a well-defined and specified external reference frame.

REQUIREMENT 28	
IDENTIFIER	`/req/model/physicalComponent/byPosition`
INCLUDED IN	Requirements class 5: `/req/model/physicalComponent`
STATEMENT	Specification of position “byPosition” shall specify, using a GeoPose or Relative Pose object, the location and orientation of the object’s reference frame relative to a well-defined and specified external reference frame.

REQUIREMENT 29	
IDENTIFIER	`/req/model/physicalComponent/byTrajectory`
INCLUDED IN	Requirements class 5: `/req/model/physicalComponent`
STATEMENT	Specification of position “byTrajectory” shall specify, at a minimum, the time-tagged location of the object’s reference frame relative to a well-defined and specified external reference frame, but may also include its orientation and any number of derivatives of the location and orientation.

REQUIREMENT 30	
IDENTIFIER	`/req/model/physicalComponent/byProcess`
INCLUDED IN	Requirements class 5: `/req/model/physicalComponent`
STATEMENT	Specification of position “byProcess” shall specify SensorML-modeled process whose output provides, at a minimum, the time-tagged location of the object’s reference frame relative to a well-defined and specified external reference frame, but may also include its orientation and any number of derivatives of the location and orientation.

8.5.1.3. Temporal Reference Frames

Just as spatial position must be related to a spatial reference frame, time must also be related to a temporal reference frame. Temporal reference frames can include a particular calendar, a particular time of day reference frame, or a frame attached to an event of interest.

Example: Examples
A temporal frame can be attached to an event of interest, such as the start of the mission. When such a reference frame is defined, time measurements can be expressed in seconds past the mission start time (which is usually itself referenced to a global time frame such as UTC or TAI).

A temporal reference frame can be defined within a physical process and is particularly useful if the component is a process that outputs its own measure of time (such as an on-board clock or high-resolution counter).

8.5.1.4. 3D Pose

This section introduces data types for expressing 3D pose information within SensorML documents. It builds on the GeoPose Standard.

When a 3D Pose object is used as the value of the `position` property within a SensorML `PhysicalComponent` or `PhysicalSystem` instance, it defines the pose of the local reference frame attached to the component (intrinsic reference frame), relative to an external reference frame (extrinsic reference frame).

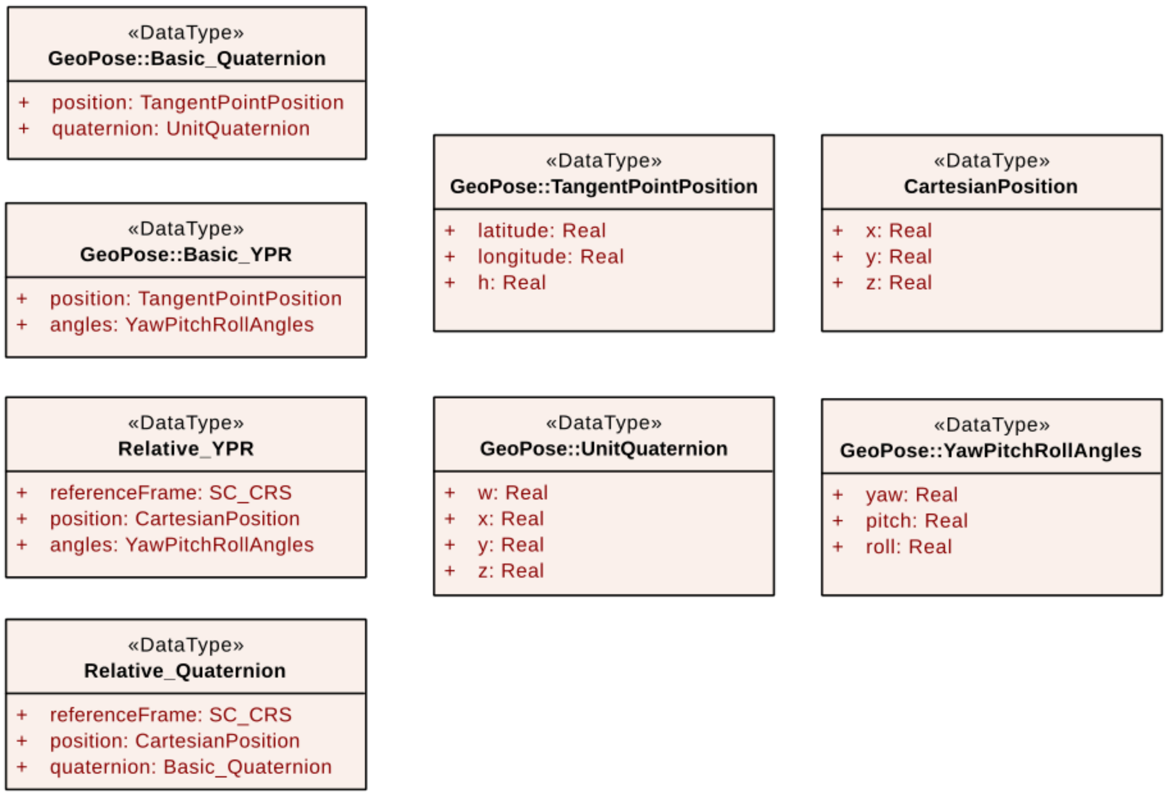


Figure 17 — Pose Data Types

8.5.1.4.1. GeoPose

The GeoPose Basic classes are used to define a pose relative to a tangent reference frame associated to the WGS84 ellipsoid. The location of the local tangent plane (LTP) is provided using EPSG:4979 coordinates and orientation is provided as yaw/pitch/roll angles or quaternion in the local tangent frame.

SensorML uses the Basic-YPR and Basic-Quaternion classes defined in the GeoPose Standard.

These classes are used to define the pose of an object relatively to the earth ellipsoid.

8.5.1.4.2. Relative Pose

The Relative Pose classes `Relative_YPR` and `Relative_Quaternion` are modeled on their GeoPose counterparts, but in this case, both position and orientation are provided relative to a cartesian reference frame.

These classes are used to define the pose of an object relatively to another object (e.g., a sensor relative to its platform).

8.5.2. Physical Component Defined

Any processing device can be considered a physical component, if it provides a processing function with well-defined inputs and outputs, if there is no intent to further divide the device description into component sub-processes, and if knowledge of its physical location is useful. Such devices could include, but not be limited to, detectors, actuators, reflectors, electrical components (e.g transformers, capacitors, resistors), or perhaps even computational units (when knowing their location in a computational facility is helpful).

REQUIREMENT 31	
IDENTIFIER	`/req/model/physicalComponent/definition`
INCLUDED IN	Requirements class 5: `/req/model/physicalComponent`
STATEMENT	A process shall be modeled as a “Physical Component” if it provides a processing function with well-defined inputs and outputs, if there is no intent to further divide the device description into sub-process components, and if knowledge of its physical location is of importance.

As shown in the models of Figure 15, the PhysicalComponent class is a concrete instantiation of an AbstractPhysicalProcess that adds the method property, which takes a ProcessMethod as its value. ProcessMethod was defined earlier in clause 7.3.2.

8.6. Requirements Class: Physical System

REQUIREMENTS CLASS 6: PHYSICAL SYSTEM	
IDENTIFIER	`/req/model/physicalSystem`
TARGET TYPE	Derived Encoding or Software Implementations
CONFORMANCE CLASS	Conformance class A.6: `/conf/model/physicalSystem`
PREREQUISITE	Requirements class 5: `/req/model/physicalComponent`
NORMATIVE STATEMENTS	Requirement 32: `/req/model/physicalSystem/package-fully-implemented` Requirement 33: `/req/model/physicalSystem/definition` Requirement 34: `/req/model/physicalSystem/dependency-core`

A physical system is used to model a hardware device as an aggregate process made of one or more components and whose location in the real world is known and of importance.

REQUIREMENT 32	
IDENTIFIER	`/req/model/physicalSystem/package-fully-implemented`
INCLUDED IN	Requirements class 6: `/req/model/physicalSystem`
STATEMENT	The encoding or software shall correctly implement all UML classes defined in the "PhysicalSystem" package described in this section.

Sensor and actuator systems (e.g., machines and robots) are typically physical systems that perform a particular feat through the coordinated actions of both physical and non-physical sub-processes. Even though a sensor system's overall application is to sense something in the environment, the system itself can consist of sensing components (e.g., detectors and sensing subsystems), action (e.g., actuators and robotic subsystems), and computational components.

Example: Examples
A weather station is an example of physical system that is composed of several sensors (thermometer, barometer, wind sensor, etc.) and other computational process such as an algorithm to compute wind chill. All these components can be described in SensorML and grouped in a PhysicalComponent description representing the station as a whole.

A hand-held digital camera can also be modeled as a physical system with an overall task of sensing radiance in a scene and generating an image. However, the camera is an aggregate of various sub-processes, each of which can be physical or non-physical, and can be sensing, acting, or computational. For example, a light detector outputs a measure of brightness, which serves as the input of a computational process which outputs a signal that provides input into an actuator that controls the opening or closing of the iris. The final iris size is sensed by another detector which inputs that value into a process that encodes that value into an EXIF format that accompanies the image, which is generated by an entirely different subsystem of the camera.

REQUIREMENT 33	
IDENTIFIER	`/req/model/physicalSystem/definition`
INCLUDED IN	Requirements class 6: `/req/model/physicalSystem`
STATEMENT	A process shall be modeled as a "Physical System" if it provides a processing function with well-defined inputs and outputs, if the device description is further divided into subprocess components, and if knowledge of its physical location is of importance.

The model for PhysicalSystem, as shown in Figure 18, is derived from AbstractPhysicalProcess, and adds the components and connections properties that have been described in the non-physical counterpart, Clause 8.4.

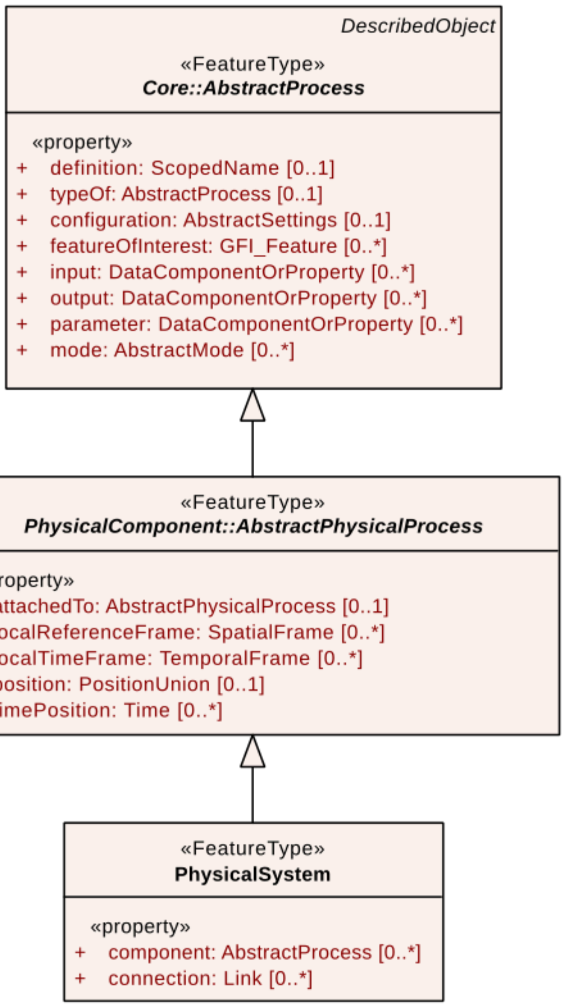

DescribedObject
«FeatureType»
Core::AbstractProcess
«property»
+ definition: ScopedName [0..1]
+ typeOf: AbstractProcess [0..1]
+ configuration: AbstractSettings [0..1]
+ featureOfInterest: GFI_Feature [0..*]
+ input: DataComponentOrProperty [0..*]
+ output: DataComponentOrProperty [0..*]
+ parameter: DataComponentOrProperty [0..*]
+ mode: AbstractMode [0..*]
«FeatureType»
PhysicalComponent::AbstractPhysicalProcess
«property»
+ attachedTo: AbstractPhysicalProcess [0..1]
+ localReferenceFrame: SpatialFrame [0..*]
+ localTimeFrame: TemporalFrame [0..*]
+ position: PositionUnion [0..1]
+ timePosition: Time [0..*]
«FeatureType»
PhysicalSystem
«property»
+ component: AbstractProcess [0..*]
+ connection: Link [0..*]

Figure 18 — Model for Physical Processing System

REQUIREMENT 34	
IDENTIFIER	`/req/model/physicalSystem/dependency-core`
INCLUDED IN	Requirements class 6: `/req/model/physicalSystem`
STATEMENT	An encoding or software passing the “Physical System” model conformance test class shall first pass the “Physical Component” conformance test class.

8.7. Requirements Class: Processes with Advanced Data Types

REQUIREMENTS CLASS 7: PROCESSES WITH ADVANCED DATA TYPES	
IDENTIFIER	`/req/model/advancedProcess`
TARGET TYPE	Derived Encoding or Software Implementation
CONFORMANCE CLASS	Conformance class A.7: `/conf/model/advancedProcess`
PREREQUISITES	Requirements class 3: `/req/model/simpleProcess` http://www.opengis.net/spec/SWE/3.0/req/uml-block-components http://www.opengis.net/spec/SWE/3.0/req/uml-choice-components
NORMATIVE STATEMENTS	Requirement 35: `/req/model/advancedProcess/dependency-core` Requirement 36: `/req/model/advancedProcess/package-fully-implemented`

The “Core Abstract Process” requirements class only requires the support of the record and scalar data types wherever a data type from the SWE Common standard is used.

This class also requires support for more advanced data types defined in the SWE Common standard: DataArray, Matrix, DataStream and Choice.

REQUIREMENT 35	
IDENTIFIER	`/req/model/advancedProcess/dependency-core`
INCLUDED IN	Requirements class 7: `/req/model/advancedProcess`
STATEMENT	An encoding or software passing the “Advanced Data Types” model conformance class shall first pass the “Abstract Core Process” conformance test class.

REQUIREMENT 36	
IDENTIFIER	`/req/model/advancedProcess/package-fully-implemented`
INCLUDED IN	Requirements class 7: `/req/model/advancedProcess`

REQUIREMENT 36	
STATEMENT	The encoding or software shall correctly implement all UML classes defined in the “Core” package and described in this section.

8.8. Requirements Class: Configurable Processes

REQUIREMENTS CLASS 8: CONFIGURABLE PROCESSES	
IDENTIFIER	`/req/model/configurableProcess`
TARGET TYPE	Derived Encoding or Software Implementations
CONFORMANCE CLASS	Conformance class A.8: `/conf/model/configurableProcess`
PREREQUISITE	/req/coreProcess
NORMATIVE STATEMENTS	Requirement 37: `/req/model/configurableProcess/dependency-core` Requirement 38: `/req/model/configurableProcess/package-fully-implemented` Requirement 39: `/req/model/configurableProcess/twoModesRequired` Requirement 40: `/req/model/configurableProcess/settingsProperty` Requirement 41: `/req/model/configurableProcess/setValueRestriction` Requirement 42: `/req/model/configurableProcess/setArrayValueRestriction` Requirement 43: `/req/model/configurableProcess/setConstraintRestriction`

Many processes, both physical and non-physical, are configurable in that they provide one with the ability to set parameters values, enable options, or select modes before or during execution/operation. Thus a general configurable process can be defined and published specifying allowed values for parameters, modes that can be selected, and options that can be enabled or disabled.

A specific process that inherits from this general process can then refine the process in several ways by: (1) specifying values for parameters, (2) further constraining the allowable values of parameters, (3) selecting an operational mode (which then sets a group of parameter values), or (4) enabling or disabling particular options such as particular outputs or components.

In this document, we will refer to the more general process as the “configurable process”, and the more specific process that inherits from it, as the “configured process”.

REQUIREMENT 37	
IDENTIFIER	`/req/model/configurableProcess/dependency-core`
INCLUDED IN	Requirements class 8: `/req/model/configurableProcess`
STATEMENT	An encoding or software passing the “Configurable Process” conformance test class shall first pass the “Core Abstract Process” conformance test class.

REQUIREMENT 38	
IDENTIFIER	`/req/model/configurableProcess/package-fully-implemented`
INCLUDED IN	Requirements class 8: `/req/model/configurableProcess`
STATEMENT	The encoding or software shall correctly implement all UML classes defined in the “Configuration” package described in this section.

A process shall be considered “configurable” if it provides options, variable parameters, or modes that can be selected or set before or during deployment or execution.

Example: Examples
A configurable process based on the linear equation (y=mx+b) defines two parameters for “slope” and “y-intercept” but does not provide values for these parameters. A configured process can inherit from this configurable process and set the values of those parameters (e.g., y=2x+4).

A process becomes “configurable” by one or more of the following characteristics:

- it defines parameters, but not defining their values;
- it defines a range or selection of possible values for parameters using the swe:AllowedValues property;
- it defines modes which in turn set a collection of parameter values when enabled; and
- it allows inputs, outputs, or components to be enabled or disabled.

A process becomes “configured” by having both of the following two characteristics:

- it inherits from a configurable process using the typeOf property; and
- it specifies one or more settings within the configuration property.

8.8.1. Modes

Example: Examples
A Doppler radar for monitoring storms may have several modes from which one can select depending on the prevailing conditions at the time. For instance, there can be “clear-sky”, “storm”, and “severe-storm” modes in which the scanning properties, radar intensity, and gain settings can all change by simply changing the mode setting

A configurable process can but is not required to contain two or more Modes. The modes property takes a list of Modes as its values, as shown in Figure 19. A mode list shall include two or more mode properties that take Mode as their value. In addition to metadata provided by the base DescribedObject class, Mode includes a configuration property that allows one to define a collection of settings for that mode.

REQUIREMENT 39	
IDENTIFIER	`/req/model/configurableProcess/twoModesRequired`
INCLUDED IN	Requirements class 8: `/req/model/configurableProcess`
STATEMENT	A modes list must include two or more mode properties if present.

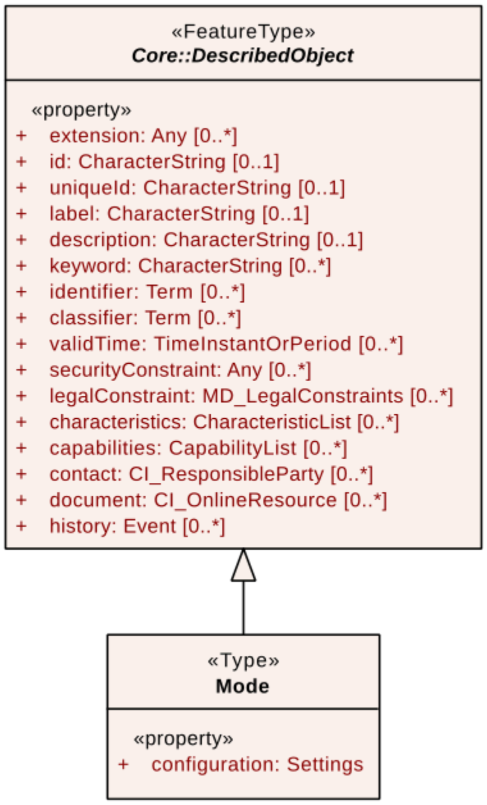

«FeatureType»
Core::DescribedObject
«property»
+ extension: Any [0..*]
+ id: CharacterString [0..1]
+ uniqueId: CharacterString [0..1]
+ label: CharacterString [0..1]
+ description: CharacterString [0..1]
+ keyword: CharacterString [0..*]
+ identifier: Term [0..*]
+ classifier: Term [0..*]
+ validTime: TimeInstantOrPeriod [0..*]
+ securityConstraint: Any [0..*]
+ legalConstraint: MD_LegalConstraints [0..*]
+ characteristics: CharacteristicList [0..*]
+ capabilities: CapabilityList [0..*]
+ contact: CI_ResponsibleParty [0..*]
+ document: CI_OnlineResource [0..*]
+ history: Event [0..*]
«Type»
Mode
«property»
+ configuration: Settings

Figure 19 — Model for Modes

The configuration property takes a Settings object, which will be described in more detail below.

8.8.2. Settings

The configuration property and its Settings value can be utilized in two cases:

- within the Mode definition of a configurable process for defining a collection of settings for that particular mode; and
- as a required property within a configured process for setting one or more configurable properties.

REQUIREMENT 40	
IDENTIFIER	`/req/model/configurableProcess/settingsProperty`
INCLUDED IN	Requirements class 8: `/req/model/configurableProcess`
STATEMENT	A configured process must include a configuration property that takes a Settings class as its value.

The Settings class is shown in Figure 20 with its possible property values shown in Figure 21. For all settings, the property in the configurable process is specified by the DataComponentPath reference.

Within the Settings class, one may: a) set particular values for parameters, b) set an array of values for a parameter — and only a parameter that takes a DataArray as its value, c) further constrain allowed values for parameters, d) set the operational mode, and e) enable or disable an input or output.

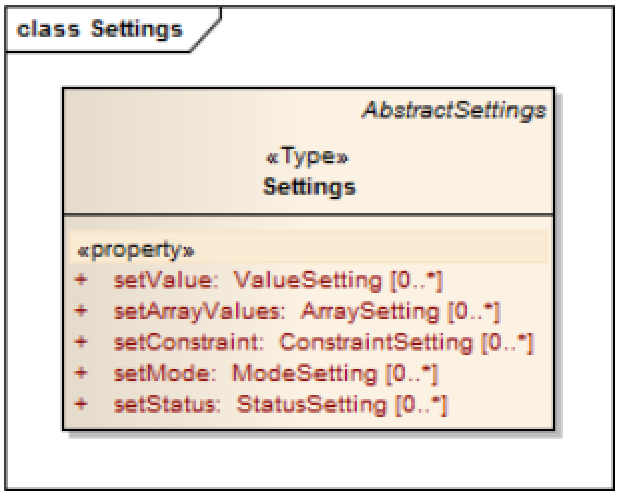

class Settings
AbstractSettings
«Type»
Settings
«property»
+ setValue: ValueSetting [0..*]
+ setArrayValues: ArraySetting [0..*]
+ setConstraint: ConstraintSetting [0..*]
+ setMode: ModeSetting [0..*]
+ setStatus: StatusSetting [0..*]

Figure 20 — Model for Configured Process Settings

REQUIREMENT 41	
IDENTIFIER	`/req/model/configurableProcess/setValueRestriction`
INCLUDED IN	Requirements class 8: `/req/model/configurableProcess`
STATEMENT	The setValue property shall only reference and set values for a parameter defined in a configurable process.

REQUIREMENT 42	
IDENTIFIER	`/req/model/configurableProcess/setArrayValueRestriction`
INCLUDED IN	Requirements class 8: `/req/model/configurableProcess`
STATEMENT	The setConstraint property shall only reference and set constraints for a parameter defined in a configurable process.

REQUIREMENT 43	
IDENTIFIER	`/req/model/configurableProcess/setConstraintRestriction`
INCLUDED IN	Requirements class 8: `/req/model/configurableProcess`
STATEMENT	The setArrayValues and setEncodedValues properties shall only reference and set array values for a parameter defined in a configurable process.

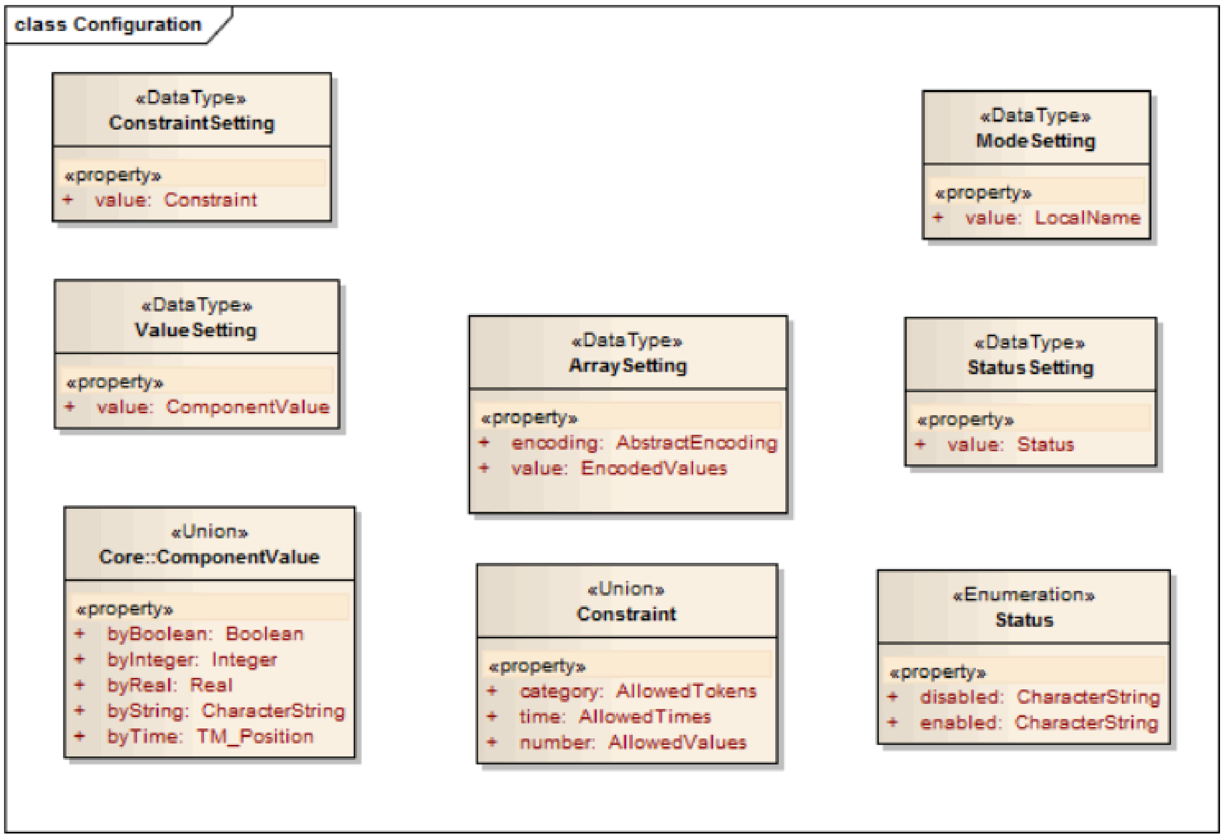


Figure 21 — Model for Settings Elements

8.9. Requirements Class: Deployment

8.9.1. Overview

REQUIREMENTS CLASS 9	
IDENTIFIER	`/req/model/deployment`
TARGET TYPE	Derived Encoding or Software Implementation
CONFORMANCE CLASS	Conformance class A.9: `/conf/model/deployment`
PREREQUISITE	Requirements class 2: `/req/model/coreProcess`
NORMATIVE STATEMENT	Requirement 44: `/req/model/deployment/package-fully-implemented`

REQUIREMENT 44	
IDENTIFIER	`/req/model/deployment/package-fully-implemented`
INCLUDED IN	Requirements class 9: `/req/model/deployment`
STATEMENT	The encoding or software shall correctly implement all UML classes defined in the "Deployment" package described in this section.

8.9.2. Deployment Class

The `Deployment` class is used to describe when, where, why and how physical or non-physical systems are being deployed. The class `Deployment` itself derives from the `DescribedObject` class and thus inherits a wide range of optional metadata supporting discovery, identification, and qualification and an option for domain and community-specific extensions.

In particular, the deployment metadata allows for the provision of:

- Contact information (e.g., the organization operating/maintaining the deployed systems, the pilot of an unmanned vehicle, etc.);
- Domain specific identifiers and classifiers for the deployment (e.g., mission number, mission type, etc.); and
- Characteristics of the deployment (e.g., sensor is mounted under shelter at 2m above ground).

The UML diagram of the `Deployment` class is shown on Figure 22 and Table 2 provides the description of the class properties:

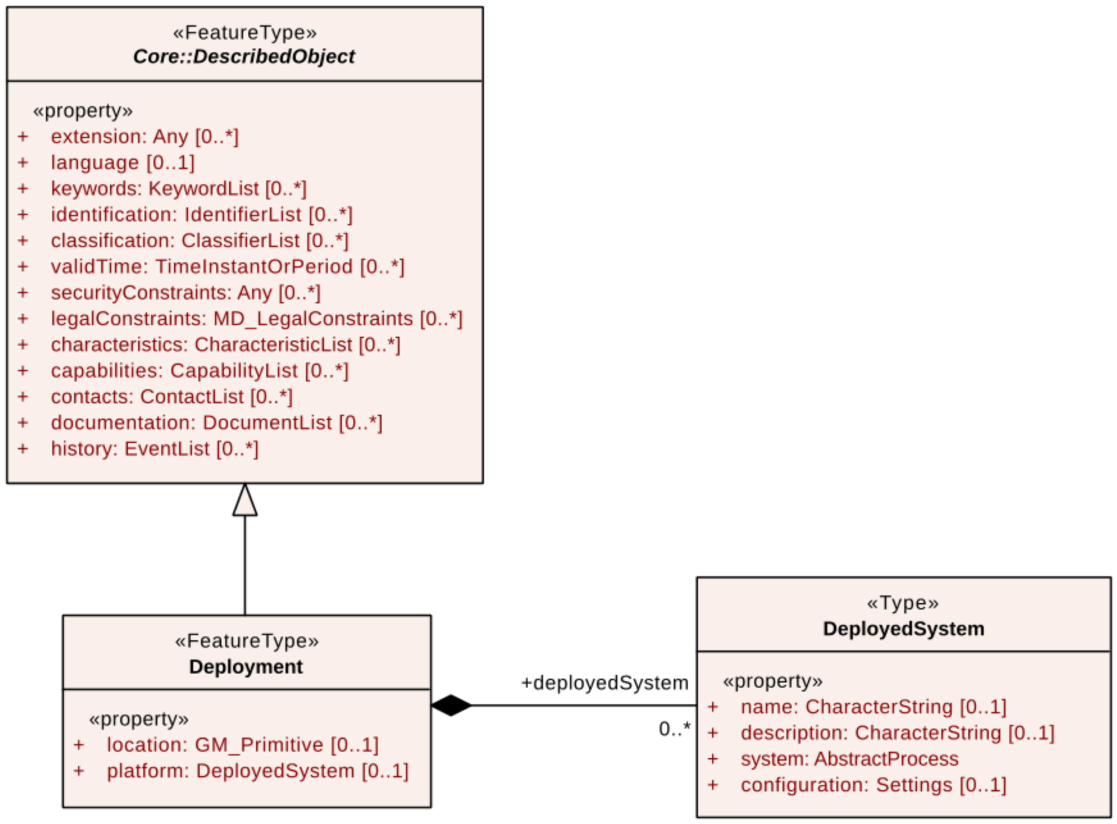

«FeatureType»
Core::DescribedObject
«property»
+ extension: Any [0..*]
+ language [0..1]
+ keywords: KeywordList [0..*]
+ identification: IdentifierList [0..*]
+ classification: ClassifierList [0..*]
+ validTime: TimeInstantOrPeriod [0..*]
+ securityConstraints: Any [0..*]
+ legalConstraints: MD_LegalConstraints [0..*]
+ characteristics: CharacteristicList [0..*]
+ capabilities: CapabilityList [0..*]
+ contacts: ContactList [0..*]
+ documentation: DocumentList [0..*]
+ history: EventList [0..*]
«FeatureType»
Deployment
«property»
+ location: GM_Primitive [0..1]
+ platform: DeployedSystem [0..1]
+deployedSystem
0..*
«Type»
DeployedSystem
«property»
+ name: CharacterString [0..1]
+ description: CharacterString [0..1]
+ system: AbstractProcess
+ configuration: Settings [0..1]

Figure 22 — Deployment Class

Table 2 — Deployment Class Properties

NAME	DEFINITION
`location`	The geographic location or area where the systems are deployed.
`platform`	Reference to the platform on which the systems are deployed.
`deployedSystem`	Description of a deployed system (as a `DeployedSystem` object, see below).

8.9.3. DeployedSystem Class

The `DeployedSystem` class is used to describe each system deployed as part of a deployment. It includes the following properties:

Table 3 — DeployedSystem Class Properties

NAME	DEFINITION
`name`	A code name for the system within the deployment (e.g., UAV1).
`description`	A description of the deployed system that is specific to the deployment.
`system`	Reference to the system, component, or process being deployed.
`configuration`	The configuration of the system used during this deployment.

8.10. Requirements Class: Derived Property

8.10.1. Overview

REQUIREMENTS CLASS 10	
IDENTIFIER	`/req/model/derived-property`
TARGET TYPE	Derived Encoding or Software Implementation
CONFORMANCE CLASS	Conformance class A.10: `/conf/model/derived-property`
PREREQUISITE	http://www.opengis.net/spec/SWE/3.0/req/uml-simple-components
NORMATIVE STATEMENT	Requirement 45: `/req/model/derived-property/package-fully-implemented`

REQUIREMENT 45	
IDENTIFIER	`/req/model/derived-property/package-fully-implemented`
INCLUDED IN	Requirements class 10: `/req/model/derived-property`
STATEMENT	The encoding or software shall correctly implement all UML classes defined in the “Derived Property” package described in this section.

8.10.2. DerivedProperty Class

The `DerivedProperty` class is used to define domain specific properties that are derived from general properties such as the ones provided by the QUDT Quantity Kinds ontology.

The UML diagram of the `DerivedProperty` class is shown on Figure 23 and Table 4 provides the description of the class properties:

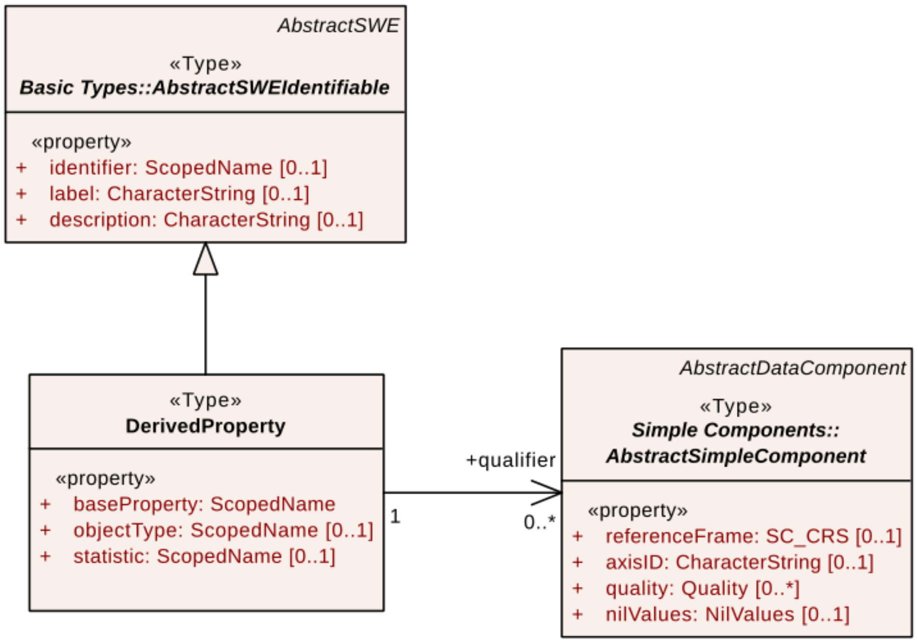

AbstractSWE
«Type»
Basic Types::AbstractSWEIdentifiable
«property»
+ identifier: ScopedName [0..1]
+ label: CharacterString [0..1]
+ description: CharacterString [0..1]
«Type»
DerivedProperty
«property»
+ baseProperty: ScopedName
+ objectType: ScopedName [0..1]
+ statistic: ScopedName [0..1]
+qualifier
1
0..*
AbstractDataComponent
«Type»
Simple Components::
AbstractSimpleComponent
«property»
+ referenceFrame: SC_CRS [0..1]
+ axisID: CharacterString [0..1]
+ quality: Quality [0..*]
+ nilValues: NilValues [0..1]

Figure 23 — DerivedProperty Class

Table 4 — DerivedProperty Class Properties

NAME	DEFINITION
`identifier`	Unique identifier of the property.
`label`	Human readable label for the property.
`description`	Longer human-readable description for the property.
`baseProperty`	Reference to the definition of the base property that this property is derived from (which can be itself a derived property).
`objectType`	Reference to the definition of a type of object/entity that the base property applies to.

NAME	DEFINITION
`statistic`	Reference to the definition of the statistical operator that is applied to the base property (e.g., hourly mean, daily maximum, standard deviation, etc.).
`qualifier`	Additional qualifier for the property (e.g., frequency range, measurement height, medium, etc.).

9

JSON IMPLEMENTATION (NORMATIVE)

9 JSON IMPLEMENTATION (NORMATIVE)

This standard defines a normative JSON implementation of the conceptual models presented in SensorML 2.1 and in the following clauses of this document:

- Clause 8.9, Requirements Class: Deployment
- Clause 8.10, Requirements Class: Derived Property
- Clause 8.5.1.4, 3D Pose

The standardization target type for all requirements classes in this clause is a JSON instance document that seeks compliance with this JSON encoding model.

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

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

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

9.1. Requirements Class: Core Schema

9.1.1. Overview

REQUIREMENTS CLASS 11	
IDENTIFIER	`/req/json-core`
TARGET TYPE	JSON Document
CONFORMANCE CLASS	Conformance class A.11: `/conf/json-core`

REQUIREMENTS CLASS 11	
PREREQUISITE	http://www.opengis.net/spec/SWE/3.0/req/json-block-components
INDIRECT PREREQUISITE	Requirements class 2: `/req/model/coreProcess`
NORMATIVE STATEMENT	Requirement 46: `/req/json-core/media-type`

9.1.2. Media Types

REQUIREMENT 46	
IDENTIFIER	`/req/json-core/media-type`
INCLUDED IN	Requirements class 11: `/req/json-core`
STATEMENT	A SensorML JSON document shall be advertised using the media type specified below.

9.1.2.1. application/sml+json

NOTE: Implementations should use **`application/vnd.ogc.sml+json`** as a preliminary media type until this Standard is stable to avoid confusing future implementations accessing JSON documents from draft versions of this Standard. The media type application/sml+json will be registered for SensorML JSON encoding, if and once this Standard is approved by the OGC Members. This note will be removed before publishing this Standard.

The draft media type submission to IANA is provided below:

```
Type name: application

Subtype name: sml+json

Required parameters: n/a

Optional parameters: n/a

Encoding considerations: binary

Security considerations: n/a

Interoperability considerations: n/a

Published specification: this document

Applications that use this media type: No known applications currently use this
media type.
```

```
   This media type is intended for applications currently using the "application/
vnd.ogc.sml+json"
   media type, which include APIs for managing, publishing or processing sensor
and system metadata.

Additional information:

  Magic number(s): n/a

  File extension(s): .json

  Macintosh file type code: TEXT

Person to contact for further information:

   1. Name: Scott Simmons
   2. Email: ssimmons@ogc.org

Intended usage: COMMON

  SensorML is an OGC Standard (OGC 23-000) that provides conceptual models and
encodings for
  describing sensor systems including sensors, actuators and platforms. This
media type applies to
  the JSON encoding of SensorML.

Restrictions on usage: n/a

Change controller: Open Geospatial Consortium (OGC)
```

Listing 1

9.1.3. General Encoding Principles

9.1.3.1. References

References are implemented in the JSON encodings using the JSON implementation of web linking.

9.1.4. DescribedObject

The JSON schema DescribedObject.json is an implementation of the `DescribedObject` UML class defined in Clause 8.2.2. It is the base schema for the following JSON objects specified in this document:

- SimpleProcess
- AggregateProcess
- PhysicalComponent
- PhysicalSystem

- Mode
- Deployment

It provides a set of metadata properties that are common to all these objects. Rather than repeating this type of metadata in all examples, the following snippets show examples of the various metadata options provided by the `DescribedObject` schema.

9.1.4.1. Unique Identifier

The unique identifier of the object is encoded as a URI that must be globally unique. The following snippets show some example URNs that can be used for this purpose:

Universally Unique Identifiers (UUID)

```
Randomly Generated UUID (version 4):
"uniqueId": "urn:uuid:e3c2ea01-ed37-4bb4-bf45-aff0ad84a331"
```

Listing

GS1 Electronic Product Codes (EPC), used in barcodes and RFID tags

```
Global Individual Asset Identifier (GIAI):
"uniqueId": "urn:epc:id:giai:0614141.12345400"

Component/Part Identifier (CPI):
"uniqueId": "urn:epc:id:cpi:0614141.123ABC.123456789"

Serialised Global Trade Item Number (SGTIN):
"uniqueId": "urn:epc:id:sgtin:123456789012.0.4711"

Global Document Type Identifier (GDTI):
"uniqueId": "urn:epc:id:gdti:0614141.12345.006847"
```

Listing

Other registered URN namespaces

```
Ship identifier in Maritime Resource Name (MRN) namespace
"uniqueId": "urn:mrn:itu:mmsi:538070999"

Navigation aid identifier in Maritime Resource Name (MRN) namespace
"uniqueId": "urn:mrn:iala:aton:us:1234.5"
```

Listing

9.1.4.2. Keywords

Keywords are provided as an array of string, as shown on the following snippet:

```
"keywords": [
  "thermometer",
  "sensor",
  "outdoor"
]
```

Listing

9.1.4.3. Identifiers

The following snippet shows how different kinds of commonly used identifiers are encoded in JSON:

```
"identifiers": [
  {
    "definition": "http://sensorml.com/ont/swe/property/ShortName",
    "label": "Short Name",
    "value": "Davis Vantage Pro2"
  },
  {
    "definition": "http://sensorml.com/ont/swe/property/Manufacturer",
    "label": "Manufacturer Name",
    "value": "Davis Instruments"
  },
  {
    "definition": "http://sensorml.com/ont/swe/property/ModelNumber",
    "label": "Model Number",
    "value": "Vantage Pro2"
  }
]
```

Listing

9.1.4.4. Classifiers

The following snippet shows how different kinds of commonly used classifiers are encoded in JSON (with or without a code space):

```
"classifiers": [
  {
    "definition": "http://sensorml.com/ont/swe/property/SensorType",
    "label": "Sensor Type",
    "value": "Motion Detector"
  },
  {
    "definition": "http://sensorml.com/ont/swe/property/PlatformType",
    "label": "Platform Type",
    "value": "Unmanned Aerial Vehicle"
  },
  {
    "definition": "http://sensorml.com/ont/swe/property/PlatformType",
    "codeSpace": "https://mmisw.org/ont/ioos/platform",
    "label": "Platform Type",
    "value": "subsurface_float"
  }
]
```

Listing

9.1.4.5. Security Constraints

The following snippet shows how to tag a SensorML object with security constraints encoded using the ISM standard:

```
"securityConstraints": [
  {
    "type": "urn:us:gov:ic:ism:v2",
    "classification": "TS",
    "classifiedBy": "USCG",
    "ownerProducer": "USA"
  }
]
```

Listing

9.1.4.6. Valid Time

The temporal validity period is encoded as an array of 2 items for begin and end times. Each date/time is either a ISO8601 date/time string or the value `now`.

```
"validTime": [
  "2023-05-07T12:30:00Z",
  "2023-05-10T00:00:00Z"
]

"validTime": [
  "2023-05-07T12:30:00Z",
  "now"
]
```

Listing

9.1.4.7. Legal Constraints

The following snippet shows how to include legal constraints within a SensorML object:

```
"legalConstraints": [
  {
    "useLimitations": [
      "Disclaimer - While every effort has been made to ensure that the data
from this sensor is accurate and reliable within the limits of the current state
of the art, we cannot assume liability for any damages caused by any errors or
omissions in the data, nor as a result of the failure of the data to function
on a particular system. We make no warranty, expressed or implied, nor does the
fact of distribution constitute such a warranty."
    ],
    "useConstraints": [
      {
        "codeSpace": "http://standards.iso.org/iso/19115/resources/Codelist/cat/
codelists.xml#MD_RestrictionCode",
        "value": "licenceDistributor"
      }
    ]
  }
]
```

Listing

9.1.4.8. Capabilities

The following snippet shows examples of system capabilities that could be provided as part of a UAV datasheet description:

```
"capabilities": [
  {
    "definition": "http://www.w3.org/ns/ssn/systems/SystemCapability",
    "label": "System Capabilities (No Wind)",
    "conditions": [
      {
        "type": "Quantity",
        "name": "wind_speed",
        "definition": "http://mmisw.org/ont/cf/parameter/wind_speed",
        "label": "Wind Speed",
        "uom": { "code": "m/s" },
        "value": 0
      }
    ],
    "capabilities": [
      {
        "type": "Quantity",
        "name": "flight_range",
        "definition": "http://qudt.org/vocab/quantitykind/Distance",
        "label": "Max Travel Distance",
        "uom": { "code": "km" },
        "value": 13
      },
      {
        "type": "Quantity",
        "name": "max_speed",
        "definition": "http://qudt.org/vocab/quantitykind/Speed",
        "label": "Max Speed",
        "uom": { "code": "km/h" },
        "value": 65
      },
      {
        "type": "Quantity",
        "name": "flight_time",
        "definition": "http://www.w3.org/ns/ssn/systems/BatteryLifetime",
        "label": "Battery Lifetime",
        "description": "Maximum flight time in stationary flight",
        "uom": { "code": "min" },
        "value": 24
      }
    ]
  }
]
```

Listing

9.1.4.9. Characteristics

Similarly, the following snippet shows example metadata for detailing battery characteristics in a UAV datasheet description:

```
"characteristics": [
  {
    "label": "Battery Characteristics",
    "characteristics": [
      {
        "type": "Quantity",
        "name": "bat_cap",
        "definition": "http://sensorml.com/ont/swe/property/BatteryCapacity",
        "label": "Battery Capacity",
        "uom": {
          "code": "W.h"
        },
        "value": 43.6
      },
      {
        "type": "Quantity",
        "name": "bat_volt",
        "definition": "http://qudt.org/vocab/quantitykind/Voltage",
        "label": "Operating Voltage",
        "uom": {
          "code": "V"
        },
        "value": 11.4
      },
      {
        "type": "Category",
        "name": "type",
        "definition": "http://dbpedia.org/resource/Battery_types",
        "label": "Battery Type",
        "value": "LiPo 3S"
      }
    ]
  }
]
```

Listing

9.1.4.10. Contacts

The following snippet shows how to encode contact information associated to the surrounding object:

```
"contacts": [
  {
    "role": "http://sensorml.com/ont/swe/property/Manufacturer",
    "organisationName": "Davis Instruments Corp.",
    "contactInfo": {
      "website": "https://www.davisinstruments.com",
      "phone": {
        "voice": "+1 (510) 732-7814"
      },
      "address": {
        "deliveryPoint": "3465 Diablo Avenue",
        "city": "Hayward",
        "postalCode": "94545",
        "administrativeArea": "CA",
        "country": "USA",
        "electronicMailAddress": "support@davisinstruments.com"
      }
    }
  }
]
```

Listing

9.1.4.11. Documents

The following snippet shows how to reference external documents related to the surrounding object:

```
"documents": [
  {
    "role": "http://dbpedia.org/resource/Web_page",
    "name": "Product Web Site",
    "description": "Webpage with specs an other resources",
    "link": {
      "href": "https://www.davisinstruments.com/pages/vantage-pro2",
      "type": "text/html"
    }
  },
  {
    "role": "http://dbpedia.org/resource/Datasheet",
    "name": "Spec Sheet",
    "link": {
      "href": "https://cdn.shopify.com/s/files/1/0515/5992/3873/files/6152c_
6162c_ss.pdf",
      "type": "application/pdf"
    }
  },
  {
    "role": "http://dbpedia.org/resource/Photograph",
    "name": "Photo",
    "link": {
      "href": "https://m.media-amazon.com/images/I/71rycLk7sFL.jpg",
      "type": "image/jpg"
    }
  }
]
```

Listing

9.1.4.12. History

The following snippet shows how to record maintenance events as part of a system description:

```
"history": [
  {
    "label": "Scheduled Maintenance",
    "description": "Monthly maintenance of station hardware.\n-Checked
electronics\n-Checked casing\nChecked power supply.\nEverything OK.",
    "time": [ "2002-03-01T10:00:00Z", "2002-03-01T11:00:00Z" ]
  },
  {
    "label": "Calibration",
    "description": "Recalibration of acquisition electronics using temperature
reference",
    "time": "2002-03-01T18:00:00Z"
  }
]
```

Listing

9.1.5. AbstractProcess

The `AbstractProcess.json` schema is the JSON schema implementation of the `AbstractProcess` UML class defined in Clause 8.2.9.

The `AbstractProcess` schema extends the `DescribedObject` schema and serves as the base class for all processes modelled and encoded in this specification. Thus, all process and system descriptions include the metadata described above plus the elements defined in this section.

9.1.5.1. Type Of

The value of the `typeOf` property is always a weblink as described in Clause 9.1.3.1.

```
"typeOf": {
  "href": "http://data.example.org/api/procedures/123",
  "uid": "urn:x-davis:station:vantagepro2",
  "title": "Davis Vantage Pro 2",
  "type": "application/sml+json"
}
```

Listing

9.1.5.2. Configuration

The following snippet shows an example configuration:

```
"configuration": {
  "setValues": [{
    "ref": "parameters/gain",
    "value": 1.25
  }],
  "setModes": [{
    "ref": "modes/THREAT_LEVEL_MODE",
    "value": "lowThreat"
  }]
}
```

Listing

9.2. Requirements Class: Simple Process Schema

9.2.1. Overview

REQUIREMENTS CLASS 12	
IDENTIFIER	`/req/json-simple-process`
TARGET TYPE	JSON Document
CONFORMANCE CLASS	Conformance class A.12: `/conf/json-simple-process`
PREREQUISITE	Requirements class 11: `/req/json-core`
INDIRECT PREREQUISITE	Requirements class 3: `/req/model/simpleProcess`
NORMATIVE STATEMENT	Requirement 47: `/req/json-simple-process/schema-valid`

9.2.2. SimpleProcess

The `SimpleProcess.json` schema is the JSON schema implementation of the `SimpleProcess` UML class defined in Clause 8.3.

REQUIREMENT 47	
IDENTIFIER	`/req/json-simple-process/schema-valid`
INCLUDED IN	Requirements class 12: `/req/json-simple-process`
STATEMENT	The JSON document instance shall be valid with respect to the JSON schema "SimpleProcess.json".

The `SimpleProcess` schema extends the `AbstractProcess` schema. Thus, it includes all elements described in Clause 9.1.5, AbstractProcess, plus the elements defined in this section.

The following snippet shows an example windchill computation process encoded in JSON:

```
{
  "type": "SimpleProcess",
  "uniqueId": "urn:x-org:process:windchill:001",
  "label": "Wind Chill Process",
  "description": "A simple process for taking temperature and wind speed and
determining wind chill.",
  "inputs": [
    {
      "name": "temp",
      "type": "Quantity",
      "definition": "http://mmisw.org/ont/cf/parameter/air_temperature",
      "label": "Air Temperature",
      "uom": { "code": "Cel", "symbol": "°C" }
    },
    {
      "name": "wind",
      "type": "Quantity",
      "definition": "http://mmisw.org/ont/cf/parameter/wind_speed",
      "label": "Wind Speed",
      "uom": { "code": "km/h" }
    }
  ],
  "outputs": [
    {
      "name": "wind_chill",
      "type": "Quantity",
      "definition": "http://mmisw.org/ont/cf/parameter/wind_chill_of_air_
temperature",
      "label": "Wind Chill Factor",
      "uom": { "code": "Cel", "symbol": "°C" }
    }
  ],
  "method": {
    "description": "The formula used to compute windchill is:\nTwc = 13.12 +
0.6215*Ta - 11.37*v^0.16 + 0.3965*Ta*v^0.16, where\nTwc is the wind chill index
on the Celsius temperature scale;\nTa is the air temperature in degrees Celsius;
\nv is the wind speed at 10 m AGL, in km/h"
  }
}
```

Listing

9.2.3. Process Method

The process method provides a textual or algorithmic description of the method implemented by the process.

9.3. Requirements Class: Aggregate Process Schema

9.3.1. Overview

REQUIREMENTS CLASS 13	
IDENTIFIER	/req/json-aggregate-process
TARGET TYPE	JSON Document
CONFORMANCE CLASS	Conformance class A.13: /conf/json-aggregate-process
PREREQUISITE	Requirements class 12: /req/json-simple-process
INDIRECT PREREQUISITE	Requirements class 4: /req/model/aggregateProcess
NORMATIVE STATEMENT	Requirement 48: /req/json-aggregate-process/schema-valid

9.3.2. AggregateProcess

The AggregateProcess.json schema is the JSON schema implementation of the AggregateProcess UML class defined in Clause 8.4.

REQUIREMENT 48	
IDENTIFIER	/req/json-aggregate-process/schema-valid
INCLUDED IN	Requirements class 13: /req/json-aggregate-process
STATEMENT	The JSON document instance shall be valid with respect to the JSON schema "Aggregate Process.json".

The snippet below shows a simple process chain example with 2 child processes and their connections.

```
{
  "type": "AggregateProcess",
  "uniqueId": "urn:x-ogc:process-chain:001",
  "label": "Simple Process Chain",
  "description": "A simple process chain that applies a linear transformation
and clips the value to a threshold.",
  "inputs": [
    {
      "name": "valueIn",
      "type": "Quantity",
      "definition": "http://sensorml.com/ont/swe/property/DN",
      "label": "Input Value",
      "uom": { "href": "http://www.opengis.net/def/nil/OGC/0/unknown" }
    }
  ],
  "outputs": [
    {
      "name": "valueOut",
      "type": "Quantity",
      "definition": "http://sensorml.com/ont/swe/property/DN",
      "label": "Output Value",
      "uom": { "href": "http://www.opengis.net/def/nil/OGC/0/unknown" }
    }
  ],
  "components": [
    {
      "name": "scale",
      "type": "SimpleProcess",
      "label": "Linear Transform 01",
      "typeOf": {
        "href": "http://example.org/processlib/linearTransform.json",
        "uid": "urn:x-org:process:LinearTransform:v1.0",
        "title": "Linear Transform"
      },
      "configuration": {
        "setValues": [
          { "ref": "parameters/slope", "value": 2.3 },
          { "ref": "parameters/intercept", "value": 1.76 }
        ]
      }
    },
    {
      "name": "clip",
      "type": "SimpleProcess",
      "label": "Threshold Clipper 01",
      "typeOf": {
        "href": "http://example.org/processlib/thresholdClipper.json",
        "uid": "urn:x-org:process:ThresholdClipper:v1.0",
        "title": "Threshold Clip"
      },
      "configuration": {
        "setValues": [
          { "ref": "parameters/threshold", "value": 15.0 }
        ]
      }
    }
  ],
  "connections": [
    { "source": "inputs/valueIn", "destination": "components/scale/inputs/x" },
    { "source": "components/scale/outputs/y", "destination": "components/clip/
inputs/valueIn" },
    { "source": "components/clip/outputs/passValue", "destination": "outputs/
valueOut" }
  ]
}
```

Listing

9.3.3. Components

The `components` property takes a `ComponentList` as its value, that is a list of nested `AbstractProcess` instances.

9.3.4. Connections

The `connections` property takes a `ConnectionList` as its value, that is a list of nested `Link` instances that specify the source and destination of each connection.

9.4. Requirements Class: Physical Component Schema

9.4.1. Overview

REQUIREMENTS CLASS 14	
IDENTIFIER	`/req/json-physical-component`
TARGET TYPE	JSON Document
CONFORMANCE CLASS	Conformance class A.14: `/conf/json-physical-component`
PREREQUISITE	Requirements class 12: `/req/json-simple-process`
INDIRECT PREREQUISITE	Requirements class 5: `/req/model/physicalComponent`
NORMATIVE STATEMENT	Requirement 49: `/req/json-physical-component/schema-valid`

9.4.2. AbstractPhysicalProcess

The schema for this abstract class is provided in AbstractPhysicalProcess.json.

It is imported by the schemas of derived classes and thus does not need to be used directly for validation.

Note that an AbstractPhysicalProcess is not a GeoJSON feature, because GeoJSON implementations do not deal well with complex nested structures, though the position of an AbstractPhysicalProcess may be encoded as GeoJSON point.

9.4.3. PhysicalComponent

The `PhysicalComponent.json` schema is the JSON schema implementation of the `PhysicalComponent` UML class defined in Clause 8.5.

REQUIREMENT 49	
IDENTIFIER	`/req/json-physical-component/schema-valid`
INCLUDED IN	Requirements class 14: `/req/json-physical-component`
STATEMENT	The JSON document instance shall be valid with respect to the JSON schema "Physical Component.json".

The following snippet illustrates how a simple sensor instance can be described as a physical component with a fixed geographic location encoded as a GeoJSON point as defined in IETF RFC 7946.

```
{
  "type": "PhysicalComponent",
  "definition": "http://www.w3.org/ns/sosa/Sensor",
  "uniqueId": "urn:x-org:systems:001",
  "label": "Outdoor Thermometer 001",
  "description": "Digital thermometer located on first floor window 1",
  "typeOf": {
    "href": "https://data.example.org/api/procedures/TP60S?f=sml",
    "title": "ThermoPro TP60S",
    "type" : "application/sml+json"
  },
  "position": {
    "type": "Point",
    "coordinates": [41.8781, -87.6298]
  }
}
```

Listing

9.4.4. 3D Pose

The position of a physical component can also be specified by a 3D pose object as specified by the JSON schema Pose.json.

```
{
  "type": "PhysicalComponent",
  "definition": "http://www.w3.org/ns/sosa/Sensor",
  "uniqueId": "urn:x-org:sensors:001",
  "label": "Sensor with GeoPose",
  "position": {
    "type": "GeoPose",
    "ltpReferenceFrame": "http://www.opengis.net/def/cs/OGC/0/NED",
    "position": {
      "lat": 47.7,
      "lon": -122.3,
      "h": 11.5
    },
    "angles": {
      "yaw": 5.946590591427664,
      "pitch": -0.4683537318018044,
      "roll": 0.0
    }
  }
}
```

Listing

9.5. Requirements Class: Physical System Schema

9.5.1. Overview

REQUIREMENTS CLASS 15	
IDENTIFIER	`/req/json-physical-system`
TARGET TYPE	JSON Document
CONFORMANCE CLASS	Conformance class A.15: `/conf/json-physical-system`
PREREQUISITES	Requirements class 13: `/req/json-aggregate-process` Requirements class 14: `/req/json-physical-component`
INDIRECT PREREQUISITE	Requirements class 6: `/req/model/physicalSystem`
NORMATIVE STATEMENT	Requirement 50: `/req/json-physical-system/schema-valid`

9.5.2. PhysicalSystem

The `PhysicalSystem.json` schema is the JSON schema implementation of the `PhysicalSystem` UML class defined in Clause 8.6.

REQUIREMENT 50	
IDENTIFIER	`/req/json-physical-system/schema-valid`

REQUIREMENT 50	
INCLUDED IN	Requirements class 15: `/req/json-physical-system`
STATEMENT	The JSON document instance shall be valid with respect to the JSON schema “PhysicalSystem.json”.

The following snippet illustrates how the specifications (datasheet) of a complete weather station can be described as a physical system. In this example, each component of the system represents one of the sensors connected to the base station:

```
{
  "type": "PhysicalSystem",
  "definition": "http://www.w3.org/ns/sosa/Sensor",
  "uniqueId": "urn:x-davis:station:vantagepro2",
  "label": "Davis Vantage Pro2 Weather Station",
  "description": "An industrial-grade weather station engineered to handle the
harshest environments...",
  "identifiers": [ ... ],
  "classifiers": [ ... ],
  "characteristics": [ ... ],
  "capabilities": [ ... ],
  "contacts": [ ... ],
  "documents": [ ... ],
  "components": [
    {
      "type": "PhysicalComponent",
      "name": "temp_sensor",
      "definition": "http://www.w3.org/ns/sosa/Sensor",
      "label": "Temperature Sensor",
       ...
    },
    {
      "type": "PhysicalComponent",
      "name": "press_sensor",
      "definition": "http://www.w3.org/ns/sosa/Sensor",
      "label": "Pressure Sensor",
       ...
    },
    {
      "type": "PhysicalComponent",
      "name": "hum_sensor",
      "definition": "http://www.w3.org/ns/sosa/Sensor",
      "label": "Humidity Sensor",
       ...
    },
    {
      "type": "PhysicalComponent",
      "name": "wind_sensor",
      "definition": "http://www.w3.org/ns/sosa/Sensor",
      "label": "Wind Sensor",
       ...
    }
  }
```

Listing

NOTE: This inline example was abridged for clarity. You can see the full example here.

Below is another example describing a specific instance of the weather station, with specifications provided above. The instance refers to the datasheet using the `typeOf` property, and also provides contact information for the operator, as well as a fixed location:

```
{
  "type": "PhysicalSystem",
  "definition": "http://www.w3.org/ns/sosa/Sensor",
  "uniqueId": "urn:x-meteofrance:stations:davis:WS00010",
  "label": "Meteo France Weather Station WS00010",
  "typeOf": {
    "href": "http://example.org/api/procedures/2ev1rrr8dkeuu",
    "uid": "urn:x-davis:station:vantagepro2",
    "type": "application/sml+json"
  },
  "contacts": [
    {
      "role": "http://sensorml.com/ont/swe/property/Operator",
      "organisationName": "Meteo France",
      "contactInfo": {
        "website": "https://www.meteo.fr",
        "phone": {
          "voice": "+33 5 61 07 80 80"
        },
        "address": {
          "deliveryPoint": "42 avenue Gaspard-Coriolis",
          "city": "TOULOUSE",
          "postalCode": "31057 Cedex 1",
          "country": "France"
        }
      }
    }
  ],
  "position": {
    "type": "Point",
    "coordinates": [
      1.35997,
      43.637788
    ]
  }
}
```

Listing

9.6. Requirements Class: Deployment Schema

9.6.1. Overview

REQUIREMENTS CLASS 16	
IDENTIFIER	`/req/json-deployment`

REQUIREMENTS CLASS 16	
TARGET TYPE	JSON Document
CONFORMANCE CLASS	Conformance class A.16: `/conf/json-deployment`
PREREQUISITE	Requirements class 11: `/req/json-core`
INDIRECT PREREQUISITE	/req/model-deployment
NORMATIVE STATEMENT	Requirement 51: `/req/json-deployment/schema-valid`

9.6.2. Deployment

The `Deployment.json` schema is the JSON schema implementation of the `Deployment` UML class defined in Clause 8.9.

REQUIREMENT 51	
IDENTIFIER	`/req/json-deployment/schema-valid`
INCLUDED IN	Requirements class 16: `/req/json-deployment`
STATEMENT	The JSON document instance shall be valid with respect to the JSON schema "Deployment.json".

The following snippet provides an example of a mission with three Saildrones deployed in an area encoded as a GeoJSON polygon as defined in IETF RFC 7946:

```
{
  "type": "Deployment",
  "definition": "http://www.w3.org/ns/ssn/Deployment",
  "uniqueId": "urn:x-saildrone:mission:2025",
  "label": "Saildrone - 2017 Arctic Mission",
  "description": "In July 2017, three saildrones were launched from Dutch
Harbor, Alaska, in partnership with NOAA Research...",
  "contacts": [ ... ],
  "validTime": [
    "2017-07-17T00:00:00Z",
    "2017-09-29T00:00:00Z"
  ],
  "location": {
    "type": "Polygon",
    "coordinates": [[
      [-173.70, 53.76],
      [-173.70, 75.03],
      [-155.07, 75.03],
      [-155.07, 53.76],
      [-173.70, 53.76]
    ]]
  },
  "deployedSystems": [
    {
      "name": "SD1001",
      "system": {
        "href": "https://data.example.org/api/systems/27559?f=sml",
        "uid": "urn:x-saildrone:platforms:SD-1001",
        "title": "Saildrone SD-1001"
      }
    },
    {
      "name": "SD1002",
      "system": {
        "href": "https://data.example.org/api/systems/27560?f=sml",
        "uid": "urn:x-saildrone:platforms:SD-1002",
        "title": "Saildrone SD-1002"
      }
    },
    {
      "name": "SD1003",
      "system": {
        "href": "https://data.example.org/api/systems/27561?f=sml",
        "uid": "urn:x-saildrone:platforms:SD-1003",
        "title": "Saildrone SD-1003"
      }
    }
  ]
}
```

Listing

9.6.3. DeployedSystem

The `DeployedSystem` schema is the JSON schema implementation of the `DeployedSystem` UML class defined in Clause 8.9, Requirements Class: Deployment.

The schema allows associating a configuration to a given deployed system or platform.

9.7. Requirements Class: Derived Property Schema

9.7.1. Overview

REQUIREMENTS CLASS 17	
IDENTIFIER	`/req/json-derived-property`
TARGET TYPE	JSON Document

REQUIREMENTS CLASS 17	
CONFORMANCE CLASS	Conformance class A.17: `/conf/json-derived-property`
PREREQUISITE	Requirements class 11: `/req/json-core`
INDIRECT PREREQUISITE	/req/uml-derived-property
NORMATIVE STATEMENT	Requirement 52: `/req/json-derived-property/schema-valid`

9.7.2. DerivedProperty

The `DerivedProperty.json` schema is the JSON schema implementation of the `DerivedProperty` UML class defined in Clause 8.10.

REQUIREMENT 52	
IDENTIFIER	`/req/json-derived-property/schema-valid`
INCLUDED IN	Requirements class 17: `/req/json-derived-property`
STATEMENT	The JSON document instance shall be valid with respect to the JSON schema "DerivedProperty.json".

The following snippets provide examples of domain specific derived properties:

```
{
  "description": "Mechanical power produced by the engine",
  "baseProperty": "http://qudt.org/vocab/quantitykind/Power",
  "objectType": "http://dbpedia.org/resource/Engine"
}
```

```
{
  "description": "Hourly average of the CPU temperature",
  "baseProperty": "http://qudt.org/vocab/quantitykind/Temperature",
  "objectType": "http://dbpedia.org/resource/Central_processing_unit",
  "statistic": "http://sensorml.com/ont/x-stats/HourlyMean"
}
```

Listing

A

ANNEX A (NORMATIVE) CONFORMANCE CLASS ABSTRACT TEST SUITE

A ANNEX A (NORMATIVE) CONFORMANCE CLASS ABSTRACT TEST SUITE

A.1. 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).

CONFORMANCE CLASS A.1: CONFORMANCE TEST CLASS: CORE CONCEPTS	
IDENTIFIER	`/conf/core`
REQUIREMENTS CLASS	Requirements class 1: `/req/core`
TARGET TYPE	Derived Models and Software Implementations
CONFORMANCE TESTS	Abstract test A.1: `/conf/core/concepts-used` Abstract test A.2: `/conf/core/processes` Abstract test A.3: `/conf/core/uniqueID` Abstract test A.4: `/conf/core/metadata` Abstract test A.5: `/conf/core/execution`

ABSTRACT TEST A.1: CORE CONCEPTS ARE THE BASE OF ALL DERIVED MODELS	
IDENTIFIER	`/conf/core/concepts-used`
REQUIREMENT	Requirement 1: `/req/core/concepts-used`
TEST PURPOSE	Verify that the target implementation correctly implements the core concepts.
TEST METHOD	Inspect the model or software implementation to verify the above.

ABSTRACT TEST A.2: A PROCESS MODEL HAS INPUTS, OUTPUTS, PARAMETERS, AND METHOD	
IDENTIFIER	`/conf/core/processes`
REQUIREMENT	Requirement 2: `/req/core/processes`
TEST PURPOSE	Verify that the target implementation correctly implements a process model.
TEST METHOD	Inspect the model or software implementation to verify the above.

ABSTRACT TEST A.3: A PROCESS MODEL HAS A UNIQUE ID	
IDENTIFIER	`/conf/core/uniqueID`
REQUIREMENT	Requirement 3: `/req/core/uniqueID`
TEST PURPOSE	Verify that the target implementation has a unique ID
TEST METHOD	Inspect the model or software implementation to verify the above.

ABSTRACT TEST A.4: A PROCESS MODEL HAS METADATA	
IDENTIFIER	`/conf/core/metadata`
REQUIREMENT	Requirement 4: `/req/core/metadata`
TEST PURPOSE	Verify that the target implementation has metadata
TEST METHOD	Inspect the model or software implementation to verify the above.

ABSTRACT TEST A.5: METADATA NOT USED IN PROCESS EXECUTION	
IDENTIFIER	`/conf/core/execution`
REQUIREMENT	Requirement 5: `/req/core/execution`
TEST PURPOSE	Verify that the target implementation does not require metadata for successful execution.

ABSTRACT TEST A.5: METADATA NOT USED IN PROCESS EXECUTION	
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.2. UML Models

A.2.1. Core Abstract Process

Tests described in this section shall be used to test conformance of software and encoding models implementing the conceptual models defined in Requirements Class: Core Abstract Process

CONFORMANCE CLASS A.2: CONFORMANCE TEST CLASS: CORE ABSTRACT PROCESS	
IDENTIFIER	`/conf/model/coreProcess`
REQUIREMENTS CLASS	Requirements class 2: `/req/model/coreProcess`
PREREQUISITES	Conformance class A.1: `/conf/core` http://www.opengis.net/spec/SWE/3.0/req/uml-block-components ISO 19115:2003/Cor.1:2006 (All Metadata)
TARGET TYPE	Derived Models Encodings and Software Implementations
CONFORMANCE TESTS	Abstract test A.6: `/conf/model/coreProcess/dependency-core` Abstract test A.7: `/conf/model/coreProcess/package-fully-implemented` Abstract test A.8: `/conf/model/coreProcess/uniqueID` Abstract test A.9: `/conf/model/coreProcess/extension Independence` Abstract test A.10: `/conf/model/coreProcess/extension Restrictions` Abstract test A.11: `/conf/model/coreProcess/SWE-Common-dependency1` Abstract test A.12: `/conf/model/coreProcess/aggregateData` Abstract test A.13: `/conf/model/coreProcess/typeOf` Abstract test A.14: `/conf/model/coreProcess/simple Inheritance` Abstract test A.15: `/conf/model/coreProcess/configuration` Abstract test A.16: `/conf/model/coreProcess/SWE-Common-dependency2`

ABSTRACT TEST A.6: DEPENDENCY ON CORE	
IDENTIFIER	`/conf/model/coreProcess/dependency-core`
REQUIREMENT	Requirement 6: `/req/model/coreProcess/dependency-core`
TEST PURPOSE	Verify that the target implementation passes the “Core Concepts” conformance test class.
TEST METHOD	Apply all tests described in Annex A.1.

ABSTRACT TEST A.7: FULLY IMPLEMENT COREPROCESS	
IDENTIFIER	`/conf/model/coreProcess/package-fully-implemented`
REQUIREMENT	Requirement 7: `/req/model/coreProcess/package-fully-implemented`
TEST PURPOSE	Verify that the target implements all UML classes.
TEST METHOD	Inspect the model or software implementation to verify the above.

ABSTRACT TEST A.8: USING UNIQUEID IDENTIFIER FOR UNIQUE ID IN COREPROCESS	
IDENTIFIER	`/conf/model/coreProcess/uniqueID`
REQUIREMENT	Requirement 8: `/req/model/coreProcess/uniqueID`
TEST PURPOSE	Verify that the target implementation uses a single uniqueId property to provide a unique ID.
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

ABSTRACT TEST A.9: EXTENSIONS MUST BE IN A SEPARATE NAMESPACE	
IDENTIFIER	`/conf/model/coreProcess/extensionIndependence`
REQUIREMENT	Requirement 9: `/req/model/coreProcess/extension Independence`
TEST PURPOSE	Verify that the target implementation uses a unique namespace.

ABSTRACT TEST A.9: EXTENSIONS MUST BE IN A SEPARATE NAMESPACE	
TEST METHOD	Inspect the model or software implementation to verify the above.

ABSTRACT TEST A.10: EXTENSIONS SHALL NOT BE REQUIRED FOR PROCESS EXECUTION	
IDENTIFIER	`/conf/model/coreProcess/extensionRestrictions`
REQUIREMENT	Requirement 10: `/req/model/coreProcess/extensionRestrictions`
TEST PURPOSE	Verify that the target implementation does not require an extension for process execution.
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

ABSTRACT TEST A.11: OBSERVABLEPROPERTY AND SWE COMMON DATA USED FOR PROCESS INPUT, OUTPUT, AND PARAMETERS	
IDENTIFIER	`/conf/model/coreProcess/SWE-Common-dependency1`
REQUIREMENT	Requirement 11: `/req/model/coreProcess/SWE-Common-dependency1`
TEST PURPOSE	Verify that the target implementation uses ObservableProperty and/or SWE Common Data for process input, output, and parameters.
TEST METHOD	Inspect the model or software implementation to verify the above.

ABSTRACT TEST A.12: USE OF SWE COMMON DATA AGGREGATE MODELS FOR PROCESS INPUT, OUTPUT, AND PARAMETERS	
IDENTIFIER	`/conf/model/coreProcess/aggregateData`
REQUIREMENT	Requirement 12: `/req/model/coreProcess/aggregateData`
TEST PURPOSE	Verify that the target implementation models tightly related data as an SWE Common Data aggregate type.
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

ABSTRACT TEST A.13: APPLICATION AND REQUIREMENTS OF TYPEOF PROPERTY	
IDENTIFIER	`/conf/model/coreProcess/typeOf`
REQUIREMENT	Requirement 13: `/req/model/coreProcess/typeOf`
TEST PURPOSE	Verify that the target implementation uses the proper process reference.
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

ABSTRACT TEST A.14: SIMPLE INHERITANCE EXTENDS A BASE CLASS REFERENCED BY TYPEOF	
IDENTIFIER	`/conf/model/coreProcess/simpleInheritance`
REQUIREMENT	Requirement 14: `/req/model/coreProcess/simpleInheritance`
TEST PURPOSE	Verify that the target implementation has a complete process description.
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

ABSTRACT TEST A.15: SUPPORTING CONFIGURATION IN PROCESSES	
IDENTIFIER	`/conf/model/coreProcess/configuration`
REQUIREMENT	Requirement 15: `/req/model/coreProcess/configuration`
TEST PURPOSE	Verify that the target implementation uses the *configuration* property to specify non-inherited restrictions.
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

ABSTRACT TEST A.16: DEPENDENCY ON SWE COMMON DATA SIMPLE TYPES	
IDENTIFIER	`/conf/model/coreProcess/SWE-Common-dependency2`
REQUIREMENT	Requirement 16: `/req/model/coreProcess/SWE-Common-dependency2`
TEST PURPOSE	Verify that the target implementation passes conformance test classes.

ABSTRACT TEST A.16: DEPENDENCY ON SWE COMMON DATA SIMPLE TYPES	
TEST METHOD	Validate according to appropriate SWE Common Data conformance tests

A.2.2. Simple Process

Tests described in this section shall be used to test conformance of software and encoding models implementing the conceptual models defined in Requirements Class: Simple Process.

CONFORMANCE CLASS A.3: CONFORMANCE TEST CLASS: SIMPLE PROCESS	
IDENTIFIER	`/conf/model/simpleProcess`
REQUIREMENTS CLASS	Requirements class 3: `/req/model/simpleProcess`
PREREQUISITE	Conformance class A.2: `/conf/model/coreProcess`
TARGET TYPE	Derived Encoding and Software Implementation
CONFORMANCE TESTS	Abstract test A.17: `/conf/model/simpleProcess/dependency-core` Abstract test A.18: `/conf/model/simpleProcess/package-fully-implemented` Abstract test A.19: `/conf/model/simpleProcess/definition` Abstract test A.20: `/conf/model/simpleProcess/method`

ABSTRACT TEST A.17: DEPENDENCY ON CORE	
IDENTIFIER	`/conf/model/simpleProcess/dependency-core`
REQUIREMENT	Requirement 17: `/req/model/simpleProcess/dependency-core`
TEST PURPOSE	Verify that the target implementation passes the test class.
TEST METHOD	Apply all tests described in Annex A.2.1.

ABSTRACT TEST A.18: FULLY IMPLEMENT SIMPLEPROCESS	
IDENTIFIER	`/conf/model/simpleProcess/package-fully-implemented`

ABSTRACT TEST A.18: FULLY IMPLEMENT SIMPLEPROCESS	
REQUIREMENT	Requirement 18: `/req/model/simpleProcess/package-fully-implemented`
TEST PURPOSE	Verify that the target implements all classes in the UML package.
TEST METHOD	Inspect the model or software implementation to verify the above.

ABSTRACT TEST A.19: SIMPLE PROCESS DEFINITION	
IDENTIFIER	`/conf/model/simpleProcess/definition`
REQUIREMENT	Requirement 19: `/req/model/simpleProcess/definition`
TEST PURPOSE	Verify that the target conforms to the definition of a “Simple Process”.
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

ABSTRACT TEST A.20: SIMPLE PROCESS HAS METHOD	
IDENTIFIER	`/conf/model/simpleProcess/method`
REQUIREMENT	Requirement 20: `/req/model/simpleProcess/method`
TEST PURPOSE	Verify that the target supports the definition of the method.
TEST METHOD	Inspect the model or software implementation to verify the above.

A.2.3. Aggregate Process

Tests described in this section shall be used to test conformance of software and encoding models implementing the conceptual models defined in Requirements Class: Aggregate Process.

CONFORMANCE CLASS A.4: CONFORMANCE TEST CLASS: AGGREGATE PROCESS	
IDENTIFIER	`/conf/model/aggregateProcess`
REQUIREMENTS CLASS	Requirements class 4: `/req/model/aggregateProcess`

CONFORMANCE CLASS A.4: CONFORMANCE TEST CLASS: AGGREGATE PROCESS	
PREREQUISITE	Conformance class A.2: `/conf/model/coreProcess`
TARGET TYPE	Derived Encoding and Software Implementation
CONFORMANCE TESTS	Abstract test A.21: `/conf/model/aggregateProcess/dependency-core` Abstract test A.22: `/conf/model/aggregateProcess/package-fully-implemented` Abstract test A.23: `/conf/model/aggregateProcess/definition` Abstract test A.24: `/conf/model/aggregateProcess/components`

ABSTRACT TEST A.21: DEPENDENCY ON CORE	
IDENTIFIER	`/conf/model/aggregateProcess/dependency-core`
REQUIREMENT	Requirement 21: `/req/model/aggregateProcess/dependency-core`
TEST PURPOSE	Verify that the target implementation passes the test class.
TEST METHOD	Apply all tests described in Annex A.2.1.

ABSTRACT TEST A.22: FULLY IMPLEMENT AGGREGATE PROCESS	
IDENTIFIER	`/conf/model/aggregateProcess/package-fully-implemented`
REQUIREMENT	Requirement 22: `/req/model/aggregateProcess/package-fully-implemented`
TEST PURPOSE	Verify that the target implementation correctly and fully defines “AggregateProcess” UML classes.
TEST METHOD	Verify that the implementation fully implements all of the package.

ABSTRACT TEST A.23: DEFINITION OF AGGREGATE PROCESS	
IDENTIFIER	`/conf/model/aggregateProcess/definition`
REQUIREMENT	Requirement 23: `/req/model/aggregateProcess/definition`

ABSTRACT TEST A.23: DEFINITION OF AGGREGATE PROCESS	
TEST PURPOSE	Verify that the target implementation is modelled as an “aggregate process” if it meets the definition.
TEST METHOD	Verify that the implementation of the conceptual model has constraints that enforce the above.

ABSTRACT TEST A.24: AGGREGATE PROCESS REQUIRES ONE OR MORE COMPONENTS	
IDENTIFIER	`/conf/model/aggregateProcess/components`
REQUIREMENT	Requirement 24: `/req/model/aggregateProcess/components`
TEST PURPOSE	Verify that the target implementation supports one or more component processes.
TEST METHOD	Verify that the implementation of the conceptual model has constraints that enforce the above.

A.2.4. Physical Component

Tests described in this section shall be used to test conformance of software and encoding models implementing the conceptual models defined in Requirements Class: Physical Component.

CONFORMANCE CLASS A.5: CONFORMANCE TEST CLASS: PHYSICAL COMPONENT	
IDENTIFIER	`/conf/model/physicalComponent`
REQUIREMENTS CLASS	Requirements class 5: `/req/model/physicalComponent`
PREREQUISITE	Conformance class A.2: `/conf/model/coreProcess`
TARGET TYPE	Derived Encoding and Software Implementation
CONFORMANCE TESTS	Abstract test A.25: `/conf/model/physicalComponent/package-fully-implemented` Abstract test A.26: `/conf/model/physicalComponent/dependency-core` Abstract test A.27: `/conf/model/physicalComponent/byPointOrLocation` Abstract test A.28: `/conf/model/physicalComponent/byPosition`

CONFORMANCE CLASS A.5: CONFORMANCE TEST CLASS: PHYSICAL COMPONENT	
	Abstract test A.29: `/conf/model/physicalComponent/byTrajectory` Abstract test A.30: `/conf/model/physicalComponent/byProcess` Abstract test A.31: `/conf/model/physicalComponent/definition`

ABSTRACT TEST A.25: FULLY IMPLEMENT PHYSICAL COMPONENT	
IDENTIFIER	`/conf/model/physicalComponent/package-fully-implemented`
REQUIREMENT	Requirement 25: `/req/model/physicalComponent/package-fully-implemented`
TEST PURPOSE	Verify that the target implementation correctly implements all “Physical Component” UML classes.
TEST METHOD	Verify that the implementation fully implements all of the package.

ABSTRACT TEST A.26: DEPENDENCY ON CORE PROCESS	
IDENTIFIER	`/conf/model/physicalComponent/dependency-core`
REQUIREMENT	Requirement 26: `/req/model/physicalComponent/dependency-core`
TEST PURPOSE	Verify that the target implementation correctly passes the “Core Abstract Process” comformance test class.
TEST METHOD	Apply all tests described in Annex A.2.1.

ABSTRACT TEST A.27: POSITION BY POINT	
IDENTIFIER	`/conf/model/physicalComponent/byPointOrLocation`
REQUIREMENT	Requirement 27: `/req/model/physicalComponent/byPointOrLocation`
TEST PURPOSE	Verify that the target implementation correctly specifies reference frame origin.
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

ABSTRACT TEST A.28: POSITION BY LOCATION AND ORIENTATION	
IDENTIFIER	`/conf/model/physicalComponent/byPosition`
REQUIREMENT	Requirement 28: `/req/model/physicalComponent/byPosition`
TEST PURPOSE	Verify that the target implementation correctly specifies “byPosition” reference frame origin.
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

ABSTRACT TEST A.29: POSITION BY TRAJECTORY	
IDENTIFIER	`/conf/model/physicalComponent/byTrajectory`
REQUIREMENT	Requirement 29: `/req/model/physicalComponent/byTrajectory`
TEST PURPOSE	Verify that the target implementation correctly specifies “byTrajectory” reference frame origin.
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

ABSTRACT TEST A.30: POSITION BY PROCESS	
IDENTIFIER	`/conf/model/physicalComponent/byProcess`
REQUIREMENT	Requirement 30: `/req/model/physicalComponent/byProcess`
TEST PURPOSE	Verify that the target implementation correctly specifies “byProcess” reference frame origin.
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

ABSTRACT TEST A.31: PHYSICAL COMPONENT DEFINITION	
IDENTIFIER	`/conf/model/physicalComponent/definition`
REQUIREMENT	Requirement 31: `/req/model/physicalComponent/definition`
TEST PURPOSE	Verify that the target implementation aligns to the specification of “Physical Component”.

ABSTRACT TEST A.31: PHYSICAL COMPONENT DEFINITION	
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.2.5. Physical System

Tests described in this section shall be used to test conformance of software and encoding models implementing the conceptual models defined in Requirements Class: Physical System.

CONFORMANCE CLASS A.6: CONFORMANCE TEST CLASS: PHYSICAL SYSTEM	
IDENTIFIER	`/conf/model/physicalSystem`
REQUIREMENTS CLASS	Requirements class 6: `/req/model/physicalSystem`
PREREQUISITE	Conformance class A.5: `/conf/model/physicalComponent`
TARGET TYPE	Derived Encoding and Software Implementation
CONFORMANCE TESTS	Abstract test A.32: `/conf/model/physicalSystem/package-fully-implemented` Abstract test A.33: `/conf/model/physicalSystem/definition` Abstract test A.34: `/conf/model/physicalSystem/dependency-core`

ABSTRACT TEST A.32: FULLY IMPLEMENT PHYSICAL SYSTEM	
IDENTIFIER	`/conf/model/physicalSystem/package-fully-implemented`
REQUIREMENT	Requirement 32: `/req/model/physicalSystem/package-fully-implemented`
TEST PURPOSE	Verify that the target implementation correctly implements “PhysicalSystem” UML classes.
TEST METHOD	Inspect the model or software implementation to verify the above.

ABSTRACT TEST A.33: PHYSICAL SYSTEM DEFINITION	
IDENTIFIER	`/conf/model/physicalSystem/definition`

ABSTRACT TEST A.33: PHYSICAL SYSTEM DEFINITION	
REQUIREMENT	Requirement 33: `/req/model/physicalSystem/definition`
TEST PURPOSE	Verify that the target implementation correctly models itself as a "Physical System".
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

ABSTRACT TEST A.34: PHYSICAL SYSTEM DEPENDENCY	
IDENTIFIER	`/conf/model/physicalSystem/dependency-core`
REQUIREMENT	Requirement 34: `/req/model/physicalSystem/dependency-core`
TEST PURPOSE	Verify that the target implementation correctly passes the "Physical Component" conformance test class.
TEST METHOD	Apply all tests described in Annex A.2.4.

A.2.6. Process with Advanced Data Types

Tests described in this section shall be used to test conformance of software and encoding models implementing the conceptual models defined in Requirements Class: Process with Advanced Data Types.

CONFORMANCE CLASS A.7: CONFORMANCE TEST CLASS: PROCESS WITH ADVANCED DATA TYPES	
IDENTIFIER	`/conf/model/advancedProcess`
REQUIREMENTS CLASS	Requirements class 7: `/req/model/advancedProcess`
PREREQUISITES	Conformance class A.2: `/conf/model/coreProcess` http://www.opengis.net/spec/SWE/3.0/req/uml-block-components http://www.opengis.net/spec/SWE/3.0/req/uml-choice-components
TARGET TYPE	Derived Encoding and Software Implementation
CONFORMANCE TESTS	Abstract test A.35: `/conf/model/advancedProcess/dependency-core` Abstract test A.36: `/conf/model/advancedProcess/package-fully-implemented`

ABSTRACT TEST A.35: ADVANCED PROCESS DEPENDENCE	
IDENTIFIER	`/conf/model/advancedProcess/dependency-core`
REQUIREMENT	Requirement 35: `/req/model/advancedProcess/dependency-core`
TEST PURPOSE	Verify that the target implementation passes the “Abstract Core Process” conformance test class.
TEST METHOD	Apply all tests described in Annex A.2.1.

ABSTRACT TEST A.36: FULLY IMPLEMENT ADVANCEDPROCESS	
IDENTIFIER	`/conf/model/advancedProcess/package-fully-implemented`
REQUIREMENT	Requirement 36: `/req/model/advancedProcess/package-fully-implemented`
TEST PURPOSE	Verify that the target implementation correctly implements “Core” package UML classes.
TEST METHOD	Inspect the model or software implementation to verify the above.

A.2.7. Configurable Processes

Tests described in this section shall be used to test conformance of software and encoding models implementing the conceptual models defined in Requirements Class: Configurable Processes.

CONFORMANCE CLASS A.8: CONFORMANCE TEST CLASS: CONFIGURABLE PROCESSES	
IDENTIFIER	`/conf/model/configurableProcess`
REQUIREMENTS CLASS	Requirements class 8: `/req/model/configurableProcess`
PREREQUISITE	/conf/req/coreProcess
TARGET TYPE	Derived Encoding and Software Implementation
CONFORMANCE TESTS	Abstract test A.37: `/conf/model/configurableProcess/dependency-core` Abstract test A.38: `/conf/model/configurableProcess/package-fully-implemented`

CONFORMANCE CLASS A.8: CONFORMANCE TEST CLASS: CONFIGURABLE PROCESSES	
	Abstract test A.39: `/conf/model/configurableProcess/twoModesRequired` Abstract test A.40: `/conf/model/configurableProcess/settingsProperty` Abstract test A.41: `/conf/model/configurableProcess/setValueRestriction` Abstract test A.42: `/conf/model/configurableProcess/setArrayValueRestriction` Abstract test A.43: `/conf/model/configurableProcess/setConstraintRestriction`

ABSTRACT TEST A.37: DEPENDENCY ON CORE PROCESS	
IDENTIFIER	`/conf/model/configurableProcess/dependency-core`
REQUIREMENT	Requirement 37: `/req/model/configurableProcess/dependency-core`
TEST PURPOSE	Verify that the target implementation passes the “Core Abstract Process” conformance test class.
TEST METHOD	Apply all tests described in Annex A.2.1.

ABSTRACT TEST A.38: FULLY IMPLEMENT CONFIGURABLE PROCESS	
IDENTIFIER	`/conf/model/configurableProcess/package-fully-implemented`
REQUIREMENT	Requirement 38: `/req/model/configurableProcess/package-fully-implemented`
TEST PURPOSE	Verify that the target implementation correctly implements all “Configuration” package UML classes.
TEST METHOD	Inspect the model or software implementation to verify the above.

ABSTRACT TEST A.39: MODE LIST REQUIRES 2 OR MORE MODES	
IDENTIFIER	`/conf/model/configurableProcess/twoModesRequired`
REQUIREMENT	Requirement 39: `/req/model/configurableProcess/twoModesRequired`
TEST PURPOSE	Verify that the target implementation implements two or more *mode* properties.

ABSTRACT TEST A.39: MODE LIST REQUIRES 2 OR MORE MODES	
TEST METHOD	Inspect the model or software implementation to verify the above.

ABSTRACT TEST A.40: A CONFIGURED PROCESS REQUIRES A SETTINGS ELEMENT	
IDENTIFIER	`/conf/model/configurableProcess/settingsProperty`
REQUIREMENT	Requirement 40: `/req/model/configurableProcess/settingsProperty`
TEST PURPOSE	Verify that the target implementation includes a *configuration* property that takes a *Settings* class as its value.
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

ABSTRACT TEST A.41: ONLY PARAMETER VALUES CAN BE SET BY SETVALUE	
IDENTIFIER	`/conf/model/configurableProcess/setValueRestriction`
REQUIREMENT	Requirement 41: `/req/model/configurableProcess/setValueRestriction`
TEST PURPOSE	Verify that the target implementation only references and sets values for a *parameter* defined in a configurable process.
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

ABSTRACT TEST A.42: ONLY PARAMETER ARRAY VALUES CAN BE SET BY SETARRAYVALUESANDSETENCODEDVALUES	
IDENTIFIER	`/conf/model/configurableProcess/setArrayValueRestriction`
REQUIREMENT	Requirement 42: `/req/model/configurableProcess/setArrayValueRestriction`
TEST PURPOSE	Verify that the target implementation only references and sets array values for a *parameter* defined in a configurable process.
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

ABSTRACT TEST A.43: ONLY PARAMETER ARRAY VALUES CAN BE SET BY SETARRAYVALUESANDSETENCODEDVALUES	
IDENTIFIER	`/conf/model/configurableProcess/setConstraintRestriction`
REQUIREMENT	Requirement 43: `/req/model/configurableProcess/setConstraint Restriction`
TEST PURPOSE	Verify that the target implementation only references and sets constraints for a *parameter* defined in a configurable process for the *setConstraint* property.
TEST METHOD	Verify that the implementation of the conceptual model has a constraint that enforces the above.

A.2.8. Deployment

Tests described in this section shall be used to test conformance of software and encoding models implementing the conceptual models defined in Requirements Class: Deployment.

CONFORMANCE CLASS A.9: CONFORMANCE TEST CLASS: DEPLOYMENT	
IDENTIFIER	`/conf/model/deployment`
REQUIREMENTS CLASS	Requirements class 9: `/req/model/deployment`
PREREQUISITES	Conformance class A.1: `/conf/core` http://www.opengis.net/spec/SWE/3.0/req/uml-block-components ISO 19115:2003/Cor.1:2006 (All Metadata)
TARGET TYPE	Derived Models Encodings and Software Implementations
CONFORMANCE TEST	Abstract test A.44: `/conf/model/deployment/package-fully-implemented`

ABSTRACT TEST A.44	
IDENTIFIER	`/conf/model/deployment/package-fully-implemented`
REQUIREMENT	Requirement 44: `/req/model/deployment/package-fully-implemented`
TEST PURPOSE	Verify that the target implements all UML classes.
TEST METHOD	Inspect the model or software implementation to verify the above.

A.2.9. Derived Property

Tests described in this section shall be used to test conformance of software and encoding models implementing the conceptual models defined in Requirements Class: Derived Property.

CONFORMANCE CLASS A.10: CONFORMANCE TEST CLASS: DERIVED PROPERTY	
IDENTIFIER	`/conf/model/derived-property`
REQUIREMENTS CLASS	Requirements class 10: `/req/model/derived-property`
PREREQUISITES	Conformance class A.1: `/conf/core` http://www.opengis.net/spec/SWE/3.0/req/uml-block-components ISO 19115:2003/Cor.1:2006 (All Metadata)
TARGET TYPE	Derived Models Encodings and Software Implementations
CONFORMANCE TEST	Abstract test A.45: `/conf/model/derived-property/package-fully-implemented`

ABSTRACT TEST A.45	
IDENTIFIER	`/conf/model/derived-property/package-fully-implemented`
REQUIREMENT	Requirement 45: `/req/model/derived-property/package-fully-implemented`
TEST PURPOSE	Verify that the target implements all UML classes.
TEST METHOD	Inspect the model or software implementation to verify the above.

A.3. JSON Implementation

Tests in the following conformance test classes shall be used to check conformance of JSON documents created according to the schemas this Standard. They shall also be used to check conformance of software implementations that output these JSON documents.

A.3.1. Core Schema

CONFORMANCE CLASS A.11: CONFORMANCE TEST CLASS: CORE SCHEMA	
IDENTIFIER	`/conf/json-core`
REQUIREMENTS CLASS	Requirements class 11: `/req/json-core`
TARGET TYPE	JSON Document
CONFORMANCE TEST	Abstract test A.46: `/conf/json-core/media-type`

ABSTRACT TEST A.46	
IDENTIFIER	`/conf/json-core/media-type`
REQUIREMENT	Requirement 46: `/req/json-core/media-type`
TEST METHOD	Check that the media type used when retrieving the document is set to the `application/sml+json`.

A.3.2. Simple Process Schema

CONFORMANCE CLASS A.12: CONFORMANCE TEST CLASS: SIMPLE PROCESS SCHEMA	
IDENTIFIER	`/conf/json-simple-process`
REQUIREMENTS CLASS	Requirements class 12: `/req/json-simple-process`
TARGET TYPE	JSON Document
CONFORMANCE TEST	Abstract test A.47: `/conf/json-simple-process/schema-valid`

ABSTRACT TEST A.47	
IDENTIFIER	`/conf/json-simple-process/schema-valid`

ABSTRACT TEST A.47	
REQUIREMENT	Requirement 47: `/req/json-simple-process/schema-valid`
TEST METHOD	Validate the JSON document using the JSON schema "SimpleProcess.json".

A.3.3. Aggregate Process Schema

CONFORMANCE CLASS A.13: CONFORMANCE TEST CLASS: AGGREGATE PROCESS SCHEMA	
IDENTIFIER	`/conf/json-aggregate-process`
REQUIREMENTS CLASS	Requirements class 13: `/req/json-aggregate-process`
TARGET TYPE	JSON Document
CONFORMANCE TEST	Abstract test A.48: `/conf/json-aggregate-process/schema-valid`

ABSTRACT TEST A.48	
IDENTIFIER	`/conf/json-aggregate-process/schema-valid`
REQUIREMENT	Requirement 48: `/req/json-aggregate-process/schema-valid`
TEST METHOD	Validate the JSON document using the JSON schema "AggregateProcess.json".

A.3.4. Physical Component Schema

CONFORMANCE CLASS A.14: CONFORMANCE TEST CLASS: PHYSICAL COMPONENT SCHEMA	
IDENTIFIER	`/conf/json-physical-component`
REQUIREMENTS CLASS	Requirements class 14: `/req/json-physical-component`
TARGET TYPE	JSON Document

CONFORMANCE CLASS A.14: CONFORMANCE TEST CLASS: PHYSICAL COMPONENT SCHEMA	
CONFORMANCE TEST	Abstract test A.49: `/conf/json-physical-component/schema-valid`

ABSTRACT TEST A.49	
IDENTIFIER	`/conf/json-physical-component/schema-valid`
REQUIREMENT	Requirement 49: `/req/json-physical-component/schema-valid`
TEST METHOD	Validate the JSON document using the JSON schema "PhysicalComponent.json".

A.3.5. Physical System Schema

CONFORMANCE CLASS A.15: CONFORMANCE TEST CLASS: PHYSICAL SYSTEM SCHEMA	
IDENTIFIER	`/conf/json-physical-system`
REQUIREMENTS CLASS	Requirements class 15: `/req/json-physical-system`
TARGET TYPE	JSON Document
CONFORMANCE TEST	Abstract test A.50: `/conf/json-physical-system/schema-valid`

ABSTRACT TEST A.50	
IDENTIFIER	`/conf/json-physical-system/schema-valid`
REQUIREMENT	Requirement 50: `/req/json-physical-system/schema-valid`
TEST METHOD	Validate the JSON document using the JSON schema "PhysicalSystem.json".

A.3.6. Deployment Schema

CONFORMANCE CLASS A.16: CONFORMANCE TEST CLASS: DEPLOYMENT SCHEMA	
IDENTIFIER	`/conf/json-deployment`
REQUIREMENTS CLASS	Requirements class 16: `/req/json-deployment`
TARGET TYPE	JSON Document
CONFORMANCE TEST	Abstract test A.51: `/conf/json-deployment/schema-valid`

ABSTRACT TEST A.51	
IDENTIFIER	`/conf/json-deployment/schema-valid`
REQUIREMENT	Requirement 51: `/req/json-deployment/schema-valid`
TEST METHOD	Validate the JSON document using the JSON schema “Deployment.json”.

A.3.7. Derived Property Schema

CONFORMANCE CLASS A.17: CONFORMANCE TEST CLASS: DERIVED PROPERTY SCHEMA	
IDENTIFIER	`/conf/json-derived-property`
REQUIREMENTS CLASS	Requirements class 17: `/req/json-derived-property`
TARGET TYPE	JSON Document
CONFORMANCE TEST	Abstract test A.52: `/conf/json-derived-property/schema-valid`

ABSTRACT TEST A.52	
IDENTIFIER	`/conf/json-derived-property/schema-valid`
REQUIREMENT	Requirement 52: `/req/json-derived-property/schema-valid`
TEST METHOD	Validate the JSON document using the JSON schema “DerivedProperty.json”.

B

ANNEX B (INFORMATIVE) REVISION HISTORY

B

ANNEX B (INFORMATIVE) REVISION HISTORY

Table B.1 — Revision History

DATE	RELEASE	AUTHOR	PARAGRAPH MODIFIED	DESCRIPTION
2012-03-16	2.0 draft	Mike Botts	All	Initial reviewed version
2012-07-27	2.0	Mike Botts	All	Completed specification
2012-09-12	2.0	John Greybeal	All	Edits and corrections throughout
2012-09-12	2.0	Alex Robin	All	Edits and corrections throughout
2013-07-04	2.0	Mike Botts	All	Final draft version
2018-08-08	2.1	Eric Hirschorn	7.9.2 (Req 43), 8.5.3, 8.1.3.15, A.8.6	Added setEncodedValues
2019-04-21	2.1	Eric Hirschorn	Introduction, Normative References	Edits requested by OAB review
2019-07-11	2.1	Eric Hirschorn	All	Requested edits from public review, including 2.0 → 2.1
2024-04-30	3.0	Alex Robin	All	Refactored to v3.0, added JSON encoding, removed XML encoding sections
2024-09-04	3.0	Alex Robin	All	Updated ATS
2025-03-18	3.0	Christian Autermann	All	Incorporated feedback from public comments

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Katharina Schleidt, Ilkka Rinne: OGC 20-082r4, *Topic 20 — Observations, measurements and samples*. Open Geospatial Consortium (2023). http://www.opengis.net/doc/as/om/3.0.

[2] Benjamin Pross, Panagiotis (Peter) A. Vretanos: OGC 18-062r2, *OGC API — Processes — Part 1: Core*. Open Geospatial Consortium (2021). http://www.opengis.net/doc/IS/ogcapi-processes-1/1.0.0.

[3] Mike Botts, Alexandre Robin, Eric Hirschorn: OGC 12-000r2, *OGC SensorML: Model and XML Encoding Standard*. Open Geospatial Consortium (2020). http://www.opengis.net/doc/IS/SensorML/2.1.0.

[4] Alexandre Robin: OGC 08-094r1, *OGC® SWE Common Data Model Encoding Standard*. Open Geospatial Consortium (2011). https://portal.ogc.org/files/?artifact_id=41157.

[5] Arne Bröring, Christoph Stasch, Johannes Echterhoff: OGC 12-006, *OGC® Sensor Observation Service Interface Standard*. Open Geospatial Consortium (2012). http://www.opengis.net/doc/IS/SOS/2.0.0.

[6] Ingo Simonis, Johannes Echterhoff: OGC 09-000, *OGC® Sensor Planning Service Implementation Standard*. Open Geospatial Consortium (2011). https://portal.ogc.org/files/?artifact_id=38478.