UML-to-GML Application Schema Pilot (UGAS-2020) Engineering Report

Table of Contents

1. Background . . . 4
2. Summary . . . 5
   2.1. Document contributor contact points . . . 6
   2.2. Foreword . . . 6
3. References . . . 7
4. Terms and definitions . . . 8
   4.1. Abbreviated terms . . . 8
   4.2. Definitions . . . 10
5. Overview . . . 11
   5.1. Future work . . . 11
      5.1.1. Extending the JSON Schema Conversion Rules . . . 11
         5.1.1.1. Encoding the Scale of a Numeric Type . . . 11
         5.1.1.2. Union Inheritance . . . 11
         5.1.1.3. Conversion of OCL Constraints to check JSON data . . . 13
      5.1.2. Conversion of JSON data to RDF using JSON-LD . . . 14
      5.1.3. Continue work on a JSON encoding for features . . . 17
      5.1.4. Investigate the impact of OCL language constructs unsupported by SHACL for the NAS . . . 17
      5.1.5. Revise the OWL conversion rule for basic types . . . 17
6. UML to JSON Schema Encoding Rule . . . 19
   6.1. Overview . . . 19
   6.2. Schema Conversion Rules . . . 19
      6.2.1. Documentation . . . 20
      6.2.2. Schema Packages . . . 26
         6.2.2.1. Definitions Schema . . . 26
         6.2.2.2. JSON Schema Version . . . 30
         6.2.2.3. Schema Identifier . . . 31
      6.2.3. Types . . . 31
         6.2.3.1. Mappings . . . 31
         6.2.3.2. Class Name . . . 32
            6.2.3.2.1. Location Independent Schema Identifiers . . . 33
            6.2.3.2.2. Type Identification . . . 34
         6.2.3.3. Abstractness . . . 36
         6.2.3.4. Inheritance . . . 36
            6.2.3.4.1. Class Generalization and Property Inheritance . . . 36
            6.2.3.4.2. Virtual Generalization . . . 39
            6.2.3.4.3. Class Specialization and Property Ranges . . . 40
         6.2.3.5. Feature and Object Type . . . 45
            6.2.3.5.1. Identifier . . . 45

6.2.3.5.2. Nested Properties . . . 47
6.2.3.6. Data Type . . . 47
6.2.3.7. Mixin Type . . . 47
6.2.3.8. Union . . . 48
6.2.3.8.1. Property Choice . . . 48
6.2.3.8.2. Type Discriminator . . . 49
6.2.3.9. Enumeration . . . 51
6.2.3.10. Code List . . . 52
6.2.3.11. Basic Type . . . 54
6.2.3.12. Default Geometry . . . 59
6.2.4. Properties . . . 61
6.2.4.1. Value Type . . . 62
6.2.4.2. Multiplicity . . . 65
6.2.4.3. Voidable . . . 67
6.2.4.4. Fixed / readOnly . . . 70
6.2.4.5. Initial Value . . . 71
6.2.5. Association Class . . . 71
6.2.6. Constraints . . . 71
6.2.7. Additional rules . . . 76
6.3. Instance Conversion Rules . . . 77
6.3.1. Coordinate Reference System in JSON data . . . 77
6.4. Conceptual Model Transformation Rules . . . 77
6.4.1. Flattening Inheritance . . . 78
6.4.2. Flattening Multiplicity . . . 78
6.4.3. Flattening Complex Types . . . 79
6.4.4. Mapping Association Classes . . . 79
6.4.5. Transforming Stereotype <<propertyMetadata>> . . . 79
6.4.6. Generating NilReason Properties for Nillable Properties . . . 81
6.4.7. Transforming OCL Constraints Defining Value Type Restrictions . . . 82
6.5. Encoding Rules . . . 83
6.5.1. GeoJSON Schema Encoding Rule . . . 84
6.5.2. Plain JSON Schema Encoding Rule . . . 87
7. Features Core Profile of Key Community Conceptual Schemas . . . 89
7.1. Overview . . . 89
7.2. Scope of the Features Core Profile . . . 90
7.3. The Features Core Profile and its encoding in JSON . . . 91
7.3.1. Overview . . . 91
7.3.2. Basic Types . . . 93
7.3.2.1. Simple Types . . . 93
7.3.2.2. Non-simple Types . . . 93
7.3.2.2.1. Measure . . . 93

7.3.2.2.2. Any . . . 96
7.3.2.2.3. Record . . . 97
7.3.2.2.4. Record Type . . . 98
7.3.3. Spatial Types . . . 98
7.3.3.1. JSON Schema implementation . . . 99
7.3.4. Temporal Types . . . 116
7.3.5. Features . . . 119
7.3.5.1. General remarks . . . 119
7.3.5.2. AnyFeature . . . 119
7.3.5.3. Thematic attributes . . . 121
7.3.5.4. Spatial attributes . . . 121
7.3.5.5. Temporal attributes . . . 121
7.3.5.6. Metadata or data quality attributes . . . 121
7.3.5.7. Association roles . . . 121
7.3.5.8. Feature collections . . . 124
7.3.5.9. Example . . . 125
7.4. Extensions . . . 126
7.4.1. Spatio-temporal geometries . . . 126
7.4.2. Non-linear Geometries . . . 130
7.4.3. Temporal geometries in another temporal coordinate reference system . . . 132
8. Using SHACL for Validation of Linked Data . . . 134
8.1. Overview . . . 134
8.2. Validation of Linked Data . . . 135
8.3. Shapes Constraint Language (SHACL) . . . 141
8.3.1. Shapes . . . 142
8.3.1.1. Focus Nodes . . . 142
8.3.1.2. Targets . . . 142
8.3.1.3. Declaring Messages for a Shape . . . 143
8.3.1.4. Deactivating a Shape . . . 143
8.3.2. Node Shapes . . . 143
8.3.3. Property Shapes . . . 143
8.3.3.1. SHACL Property Paths . . . 143
8.3.3.2. Non-Validating Property Shape Characteristics . . . 144
8.3.4. Graphs . . . 144
8.3.4.1. Shapes Graph . . . 145
8.3.4.2. Data Graph . . . 145
8.3.5. Core Constraint Components . . . 145
8.3.5.1. Value Type Constraint Components . . . 146
8.3.5.2. Cardinality Constraint Components . . . 146
8.3.5.3. Value Range Constraint Components . . . 146
8.3.5.4. String-based Constraint Components . . . 147

8.3.5.5. Property Pair Constraint Components . . . 147
8.3.5.6. Logical Constraint Components . . . 148
8.3.5.7. Shape-based Constraint Components . . . 148
8.3.5.8. Other Constraint Components . . . 149
8.3.6. SPARQL-based Constraints . . . 149
8.4. SHACL Conversion Rules . . . 149
8.4.1. Documentation . . . 151
8.4.2. Package . . . 152
8.4.2.1. Name and Namespace . . . 152
8.4.2.2. Imports . . . 152
8.4.2.3. Entailment . . . 154
8.4.3. Types . . . 155
8.4.3.1. General . . . 155
8.4.3.2. Mapping . . . 155
8.4.3.3. Type Name . . . 156
8.4.3.4. Abstractness . . . 156
8.4.3.5. Generalization/Inheritance . . . 156
8.4.3.6. Feature and Object Types . . . 157
8.4.3.7. Mixin Types . . . 157
8.4.3.8. Data Types . . . 158
8.4.3.9. Basic Types . . . 158
8.4.3.10. Unions . . . 160
8.4.3.10.1. Type Discriminator . . . 161
8.4.3.10.2. Attribute Choices . . . 162
8.4.3.11. Enumerations . . . 165
8.4.3.11.1. Choice of Literals . . . 165
8.4.3.11.2. As Code List . . . 166
8.4.3.12. Code Lists . . . 166
8.4.4. Property . . . 166
8.4.4.1. General . . . 166
8.4.4.2. Documentation . . . 167
8.4.4.3. Range . . . 167
8.4.4.4. Multiplicity . . . 167
8.4.4.5. rdfs:subPropertyOf Relationships . . . 168
8.4.4.6. Attribute . . . 170
8.4.4.7. Association Role . . . 170
8.4.4.8. Property Metadata Stereotype . . . 170
8.4.5. Association Class . . . 171
8.4.6. OCL Constraints . . . 171
8.4.6.1. Translation solely based on SHACL Core . . . 171
8.4.6.2. Translation using SHACL Core and SHACL Advanced Features . . . 172

8.4.6.3. Translation using SPARQL queries, embedded in SHACL constructs . . . 174
8.4.6.4. SHACL Implementations for specific NAS OCL Constraints . . . 176
8.4.6.4.1. Property Co-constraint (related entity, numeric comparison) . . . 176
8.4.6.4.2. Property Co-constraint (type conditional, numeric comparison) . . . 178
8.4.6.4.3. Property Valid Numeric Interval . . . 179
8.4.6.4.4. Related Entity Property Numeric Range Restriction . . . 181
8.4.6.4.5. Special case #1 . . . 183
8.4.6.4.6. Special case #2 . . . 184
8.4.6.4.7. Related Datatype Use Required (at least one) . . . 185
8.5. Summary . . . 186
9. Generating OpenAPI definitions from an application schema in UML . . . 188
9.1. Introduction . . . 188
9.2. Scenario . . . 189
9.3. Analysis and design . . . 191
9.3.1. Analyze the target OpenAPI definition . . . 191
9.3.1.1. Starting with an OpenAPI template . . . 191
9.3.1.2. Add support for conformance class "Core" . . . 192
9.3.1.3. In "Core", add support for the encoding conformance classes . . . 199
9.3.1.4. Conformance class "OpenAPI 3.0 Specification" . . . 212
9.3.1.5. Add support for the conformance class "Coordinate Reference System by Reference" . . . 212
9.3.1.6. Add support for additional query parameters before feature processing . . . 215
9.3.1.7. Feature type specific modifications . . . 217
9.3.1.7.1. Identifying the feature collections . . . 217
9.3.1.7.2. Constrain the collection identifier values . . . 218
9.3.1.7.3. Constrain the response schema for each feature collection. . . . 218
9.3.1.8. Add support for additional query parameters during finalization of the OpenAPI definition . . . 221
9.3.2. Design of a minimal OpenAPI target in ShapeChange . . . 222
9.3.2.1. Additional information in the UML model . . . 222
9.3.2.2. Dependency on the JSON Schema target . . . 223
9.3.2.3. ShapeChange configuration . . . 224
9.3.3. JSON Schema variants . . . 227
9.3.3.1. General remarks . . . 227
9.3.3.2. Issue: anchors . . . 227
9.3.3.3. Issue: no type arrays . . . 227
9.3.3.4. Issue: `nullable` instead of a type `null` . . . 228
9.3.3.5. Selecting the OpenAPI 3.0 JSON Schema variant . . . 228
9.4. Results . . . 228
Annex A: OpenAPI and JSON Schema Files . . . 229
A.1. Results from the OpenAPI Scenario . . . 229

Annex B: ShapeChange Configurations for Derivation of JSON Schemas from the NAS Conceptual Model . . . 254
B.1. NSG Application Schema . . . 257
B.2. ISO 19100-series Schemas . . . 296
B.3. U.S. Intelligence Community Metadata Schema . . . 306
B.4. NAS SWE Common Implementation Schema . . . 311
B.5. SWE Common 2.0 JSON Schema . . . 321
Annex C: Examples of SPARQL-based SHACL Functions . . . 328
Annex D: Revision History . . . 333
Annex E: Bibliography . . . 334

Publication Date: 2021-01-18

Approval Date: 2020-12-11

Submission Date: 2020-11-06

Reference number of this document: OGC 20-012

Reference URL for this document: http://www.opengis.net/doc/PER/ugas2020

Category: OGC Engineering Report

Editor: Johannes Echterhoff

Title: UML-to-GML Application Schema Pilot (UGAS-2020) Engineering Report

OGC Engineering Report

COPYRIGHT

WARNING

This document is not an OGC Standard. This document is an OGC Engineering Report created as a deliverable in an OGC Interoperability Initiative and is not an official position of the OGC membership. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an OGC Standard. Further, any OGC Engineering Report should not be referenced as required or mandatory technology in procurements. However, the discussions in this document could very well lead to the definition of an OGC Standard.

LICENSE AGREEMENT

Permission is hereby granted by the Open Geospatial Consortium, ("Licensor"), free of charge and subject to the terms set forth below, to any person obtaining a copy of this Intellectual Property and any associated documentation, to deal in the Intellectual Property without restriction (except as set forth below), including without limitation the rights to implement, use, copy, modify, merge, publish, distribute, and/or sublicense copies of the Intellectual Property, and to permit persons to whom the Intellectual Property is furnished to do so, provided that all copyright notices on the intellectual property are retained intact and that each person to whom the Intellectual Property is furnished agrees to the terms of this Agreement.

If you modify the Intellectual Property, all copies of the modified Intellectual Property must include, in addition to the above copyright notice, a notice that the Intellectual Property includes modifications that have not been approved or adopted by LICENSOR.

THIS LICENSE IS A COPYRIGHT LICENSE ONLY, AND DOES NOT CONVEY ANY RIGHTS UNDER ANY PATENTS THAT MAY BE IN FORCE ANYWHERE IN THE WORLD. THE INTELLECTUAL PROPERTY IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE DO NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE INTELLECTUAL PROPERTY WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE INTELLECTUAL PROPERTY WILL BE UNINTERRUPTED OR ERROR FREE. ANY USE OF THE INTELLECTUAL PROPERTY SHALL BE MADE ENTIRELY AT THE USER'S OWN RISK. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR ANY CONTRIBUTOR OF INTELLECTUAL PROPERTY RIGHTS TO THE INTELLECTUAL PROPERTY BE LIABLE FOR ANY CLAIM, OR ANY DIRECT, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM ANY ALLEGED INFRINGEMENT OR ANY LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR UNDER ANY OTHER LEGAL THEORY, ARISING OUT OF OR IN CONNECTION WITH THE IMPLEMENTATION, USE, COMMERCIALIZATION OR PERFORMANCE OF THIS INTELLECTUAL PROPERTY.

This license is effective until terminated. You may terminate it at any time by destroying the Intellectual Property together with all copies in any form. The license will also terminate if you fail to comply with any term or condition of this Agreement. Except as provided in the following sentence, no such termination of this license shall require the termination of any third party end-user sublicense to the Intellectual Property which is in force as of the date of notice of such termination. In addition, should the Intellectual Property, or the operation of the Intellectual Property, infringe, or in LICENSOR's sole opinion be likely to infringe, any patent, copyright, trademark or other right of a third party, you agree that LICENSOR, in its sole discretion, may terminate this license without any compensation or liability to you, your licensees or any other party. You agree upon termination of any kind to destroy or cause to be destroyed the Intellectual Property together with all copies in any form, whether held by you or by any third party.

Except as contained in this notice, the name of LICENSOR or of any other holder of a copyright in all or part of the Intellectual Property shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Intellectual Property without prior written authorization of LICENSOR or such copyright holder. LICENSOR is and shall at all times be the sole entity that may authorize you or any third party to use certification marks, trademarks or other special designations to indicate compliance with any LICENSOR standards or specifications.

This Agreement is governed by the laws of the Commonwealth of Massachusetts. The application to this Agreement of the United Nations Convention on Contracts for the International Sale of Goods is hereby expressly excluded. In the event any provision of this Agreement shall be deemed unenforceable, void or invalid, such provision shall be modified so as to make it valid and enforceable, and as so modified the entire Agreement shall remain in full force and effect. No decision, action or inaction by LICENSOR shall be construed to be a waiver of any rights or remedies available to it.

None of the Intellectual Property or underlying information or technology may be downloaded or otherwise exported or reexported in violation of U.S. export laws and regulations. In addition, you are responsible for complying with any local laws in your jurisdiction which may impact your right to import, export or use the

Intellectual Property, and you represent that you have complied with any regulations or registration procedures required by applicable law to make this license enforceable.

Chapter 1. Background

International Organization for Standardization (ISO) 19109:2015 *Geographic information – Rules for application schema* Clause 2.3 *UML application schema*, defines a Unified Modeling Language (UML) metaschema for feature-based data. That General Feature Model (GFM) serves as the key underpinning for a family of ISO standards employed by national and international communities to define the structure and content of geospatial data.

OGC 07-036r1 *OpenGIS Geography Markup Language (GML) Encoding Standard* Annex E *UML-to-GML application schema encoding rules*, defines a set of encoding rules that may be used to transform a GFM-conformant UML concept model to a corresponding XML Schema (XSD) based on the structure and requirements of GML. The resulting XSD file(s) may be used to define the structure of, and validate the content of, XML instance documents used in the exchange of geospatial data among open-source, commercial, consortia, and government systems. This methodology for deriving technology-specific encodings of GFM-conformant UML concept models based on formalized sets of encoding rules has been successfully applied in other contexts than GML (e.g., technologies based on Resource Description Framework (RDF) encodings) and has come to be generically known as “UGAS.”

ShapeChange (https://shapechange.net/) is an open-source implementation of the UGAS methodology that is regularly employed by the international standards community in the application of ISO standards to the resolution of issues involving data model standardization, content validation, and exchange of geospatial data. The U.S. National System for Geospatial Intelligence (NSG) Application Schema (NAS) standardizes an NSG-wide model for geospatial data that is mission-agnostic and technology-neutral. From it, using Model Driven Architecture (MDA) techniques, technology-tied Platform Specific Models (PSM) may be automatically derived and directly employed in system development. ShapeChange is a key technological underpinning that is heavily employed in development and management of NAS-related technology artifacts.

Chapter 2. Summary

During UGAS-2020 emerging technology requirements for NAS employment in the NSG, and with general applicability for the wider geospatial community, were investigated and solutions developed in four areas.

1. To enable a wide variety of analytic tradecrafts in the NSG to consistently and interoperably exchange data, the NAS defines an NSG-wide standard UML-based application schema in accordance with the ISO 19109 General Feature Model. In light of continuing technology evolution in the commercial marketplace it is desirable to be able to employ (NAS-conformant) JSON-based data exchanges alongside existing (NAS-conformant) XML-based data exchanges. A prototype design and implementation of UML Application Schema to JSON Schema rules (see the OWS-9 SSI UGAS Conversion Engineering Report) was reviewed and revised based on the final draft IETF JSON Schema standard "draft 2019-09." The revised implementation was evaluated using NAS Baseline X-3. This work is reported in section UML to JSON Schema Encoding Rule.

2. To maximize cross-community data interoperability the NAS employs conceptual data schemas developed by communities external to the NSG, for example as defined by the ISO 19100-series standards. At the present time there are no defined JSON-based encodings for those conceptual schemas. A JSON-based core profile was developed for key external community conceptual schemas, particularly components of those ISO 19100-series standards used to enable data discovery, access, control, and use in data exchange in general, including in the NSG. This work is reported in section Features Core Profile of Key Community Conceptual Schemas.

   The Features Core Profile and its JSON encoding have been specified with a broader scope than the NAS. It builds on the widely used GeoJSON standard and extends it with minimal extensions to support additional concepts that are important for the wider geospatial community and the OGC API standards, including support for solids, coordinate reference systems, and time intervals. These extensions have been kept minimal to keep implementation efforts as low as possible. If there is interest in the OGC membership, the JSON encoding of the Core Profile could be a starting point for a JSON encoding standard for features in the OGC. A new Standards Working Group for a standard OGC Features and Geometries JSON [https://portal.ogc.org/files/?artifact_id=95319&version=1] has been proposed.

3. Linked data is increasingly important in enabling "connect the dots" correlation and alignment among diverse, distributed data sources and data repositories. Validation of both data content and link-based data relationships is critical to ensuring that the resulting virtual data assemblage has logical integrity and thus constitutes meaningful information. SHACL, a language for describing and validating RDF graphs, appears to offer significant as yet unrealized potential for enabling robust data validation in a linked-data environment. The results of evaluating that potential – with emphasis on deriving SHACL from a UML-based application schema - are reported in section Using SHACL for Validation of Linked Data.

4. The OpenAPI initiative is gaining traction in the commercial marketplace as a next-generation approach to defining machine-readable specifications for RESTful APIs in web-based environments. The OGC is currently shifting towards interface specifications based on the OpenAPI 3.1 specification. That specification defines both the interface (interactions between the client and service) and the structure of data payloads (content) offered by that service. It is

desirable to be able to efficiently model the service interface using UML and then automatically derive the physical expression of that interface (for example, as a JSON file) using Model Driven Engineering (MDE) techniques alongside the derivation of JSON Schema defining data content. A preliminary analysis and design based on the OGC API Features standard, parts 1 and 2, for sections other than for content schemas, is reported in section Generating OpenAPI definitions from an application schema in UML.

All ShapeChange enhancements developed within the UGAS-2020 Pilot have been publicly released as a component of ShapeChange v2.10.0. https://shapechange.net has been updated to document the enhancements.

2.1. Document contributor contact points

All questions regarding this document should be directed to the editor or the contributors:

Contacts

Name	Organization	Role
Johannes Echterhoff	interactive instruments GmbH	Editor
Clemens Portele	interactive instruments GmbH	Contributor

2.2. Foreword

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

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

Chapter 3. References

The following normative documents are referenced in this document.

- Internet Engineering Task Force (IETF). RFC 8259: **The JavaScript Object Notation (JSON) Data Interchange Format** [online]. Edited by T. Bray. 2017 [viewed 2020-04-02]. Available at https://tools.ietf.org/html/rfc8259
- Internet Engineering Task Force (IETF). RFC 7946: **The GeoJSON Format** [online]. Edited by H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, T. Schaub. 2016 [viewed 2020-04-02]. Available at https://tools.ietf.org/html/rfc7946
- ISO 8601-2:2019 **Date and time — Representations for information interchange — Part 1: Extensions**
- ISO 19109:2015 **Geographic information – Rules for application schema**
- Open Geospatial Consortium (OGC). OGC 17-069r3: **OGC API - Features - Part 1: Core, Version 1.0.0** [online]. Edited by C. Portele, P. Vretanos. 2019 [viewed 2020-04-02]. Available at http://docs.opengeospatial.org/is/17-069r3/17-069r3.html
- Open Geospatial Consortium (OGC). OGC 07-036r1: **OpenGIS Geography Markup Language (GML) Encoding Standard, Version 3.2.2** [online]. Edited by C. Portele. 2016 [viewed 2020-04-02]. Available at https://portal.opengeospatial.org/files/?artifact_id=74183
- OpenAPI Initiative (OAI). **OpenAPI Specification 3.0** [online]. 2020 [viewed 2020-04-02]. The latest patch version at the time of publication of this standard was 3.0.3, available at http://spec.openapis.org/oas/v3.0.3

Chapter 4. Terms and definitions

- **Linked Data**:

> Linked Data is the data format that supports the Semantic Web. The basic rules for Linked Data are defined as:
>
> - Use Uniform Resource Identifiers (URIs) to identify things;
> - Use HTTP URIs so that these things can be referred to and looked up ("dereferenced") by people and user agents;
> - Provide useful information about the thing when its URI is dereferenced, using standard formats such as RDF/XML; and
> - Include links to other, related URIs in the exposed data to improve discovery of other related information on the Web.
>
> — W3C Semantic Web Wiki [https://www.w3.org/2001/sw/wiki/Semantic_Web_terminology#linked_data]

4.1. Abbreviated terms

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CFP	Call for Proposals
CRS	Coordinate Reference System
CWA	Closed-World Assumption
DASH	Data Shapes Vocabulary
DCAT	Data Catalog Vocabulary
DDL	Data Definition Language
EAP	Enterprise Architect Project
ECMA	European association for standardizing information and communication systems
EPSG	European Petroleum Survey Group
ER	Engineering Report
GFM	General Feature Model
GML	Geography Markup Language
GML-SF	GML Simple Features
GRDDL	Gleaning Resource Descriptions from Dialects of Languages
GUI	Graphical User Interface

HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IP	Innovation Program
IRI	Internationalized Resource Identifier
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
JSON-LD	JSON for Linked Data
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MDG	Model Driven Generation
MSL	Mean Sea Level
NAS	NSG Application Schema
NEO	NSG Enterprise Ontology
NSG	U.S. National System for Geospatial Intelligence
OAI	OpenAPI Initiative
OCL	Object Constraint Language
OGC	Open Geospatial Consortium
OWA	Open-World Assumption
OWL	Web Ontology Language
POWDER	Protocol for Web Description Resources
PSM	Platform Specific Model
R2RML	RDB to RDF Mapping Language
RDB	Relational Databases
RDF	Resource Description Framework
RDFS	RDF Schema
RDFa	RDF in Attributes
RIF	Rule Interchange Format
SCXML	ShapeChange XML
SHACL	Shapes Constraint Language
SKOS	Simple Knowledge Organization System
SPARQL	SPARQL query language for RDF
SQL	Structured Query Language
SSI	System Security Interoperability

SWE	Sensor Web Enablement
SWG	Standards Working Group
SWRL	Semantic Web Rule Language
TC	Technical Committee
TOSH	TopBraid Data Shapes Library
UCUM	The Unified Code for Units of Measure
UGAS	UML to GML Application Schema
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Unform Resource Locator
W3C	World Wide Web Consortium
WGS	World Geodetic System
XML	Extensible Markup Language
XPath	XML Path Language
XQuery	XML Query Language
XSD	XML Schema
YAML	YAML Ain't Markup Language

4.2. Definitions

- **feature type** - A feature type as defined by the General Feature Model (see ISO 19109:2015). In an application schema, a feature type is typically modeled using stereotype <<featureType>>, or a stereotype that maps to that stereotype.
- **object type** - An object type is a standard class, instances are plain objects with identify. An object type is not a feature type. In order for ShapeChange to recognize a class as an object type, the class must have no stereotype, or must have stereotype <<type>>, or must have a stereotype that maps to one of these two options.
- **data type** - As defined by ISO 19103:2015, section 6.10, a data type is a class with stereotype [dataType] (or a stereotype that maps to that stereotype), which is a set of properties that lacks identity.

Chapter 5. Overview

This Engineering Report documents results of the UGAS-2020 Pilot. It consists of the following chapters:

- UML to JSON Schema Encoding Rule
- Features Core Profile of Key Community Conceptual Schemas
- Using SHACL for Validation of Linked Data
- Generating OpenAPI definitions from an application schema in UML

5.1. Future work

5.1.1. Extending the JSON Schema Conversion Rules

During UGAS-2020, several ideas for additional conversion rules, or extensions of existing conversion rules, were discussed. The following sections document these ideas. They may be implemented in the future.

5.1.1.1. Encoding the Scale of a Numeric Type

The JSON Schema keyword "multipleOf" can be used to restrict a numeric value to only be valid if division by the keyword's value (which must be a numeric greater than 0) results in an integer. The keyword could be used to define and validate the scale of a numeric property.

For example, using the type definition: `{ "type": "number", "multipleOf": 0.01 }`, the values 5, 5.1, and 5.12 would be valid, while value 5.123 would be invalid.

Future work can define a conversion rule to support encoding the scale of a numeric property using the JSON Schema keyword "multipleOf."

5.1.1.2. Union Inheritance

A union type can be converted to JSON Schema in two different ways, as documented in section Union. Currently, inheritance relationships between union types are not supported.

Even though union inheritance has not been used in application schema development thus far, it does not appear to be forbidden by any ISO 19100 standard. However, the semantics of using a union subtype instead of a supertype as a property value are undefined and would need to be established - probably similar to how this would need to be done for inheritance between enumerations and code lists. Because of the lack of a standards-based definition for inheritance of unions, enumerations, and code lists, the conversion of inheritance relationships between unions has not been implemented in UGAS-2020.

However, UGAS-2020 briefly analyzed how union inheritance could be realized. The generalization relationship between a union subtype and its supertype could be realized using the JSON Schema keyword "anyOf", as shown in Listing 1 (using conversion rule *rule-json-cls-union-propertyCount* to encode the options of each union). This example can be useful for future work.

Listing 1. Example of encoding a generalization relationship between two unions

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "definitions": {
    "UnionA": {
      "type": "object",
      "properties": {
        "option1": {
          "type": "string"
        },
        "option2": {
          "type": "number"
        }
      },
      "additionalProperties": false,
      "minProperties": 1,
      "maxProperties": 1
    },
    "UnionB": {
      "anyOf": [
        {
          "$ref": "#/definitions/UnionA"
        },
        {
          "type": "object",
          "properties": {
            "option2": {
              "type": "string"
            },
            "option3": {
              "type": "number"
            }
          },
          "additionalProperties": false,
          "minProperties": 1,
          "maxProperties": 1
        }
      ]
    }
  },
  "$ref": "#/definitions/UnionB"
}
```

These JSON objects are valid against the schema from Listing 1:

```
{
  "option1": "x"
}
{
  "option2": "x"
}
{
  "option2": 3.1
}
```

These JSON object are invalid against the schema from Listing 1:

```
{
  "option2": true
}
{
  "option3": "x"
}
```

5.1.1.3. Conversion of OCL Constraints to check JSON data

One of the main goals in UGAS-2020 was to develop UML to JSON Schema conversion rules. A JSON Schema produced using these rules defines a JSON format, for encoding application data in JSON. The JSON Schema can be used to ensure that a given JSON object complies with that format. This is a powerful building block for interoperable web applications, because web applications can now ensure - using JSON Schema validators - that JSON data they exchange actually is compliant with the agreed format.

However, there may be some aspects of the data format that cannot be checked by the JSON Schema, and thus need to be checked by the application itself. One example is that an object which is only encoded by reference in the JSON data actually is one of the allowed types. Another example is data requirements defined by OCL constraints.

In application schema modelling, OCL constraints are used to define requirements that cannot be expressed in UML alone. A full analysis of if and how OCL can be converted to JSON Schema, or if some specification or tool exists, with which OCL expressions can be checked on JSON data, was out-of-scope for UGAS-2020.

Only a particular kind of OCL constraints defined by the NAS, identified as critical for achieving a useful JSON Schema encoding of the NAS, has been investigated in UGAS-2020: constraints that disallow related entity types. For further details, see section Constraints.

Future work may therefore analyse the conversion of OCL constraints to artifacts that can validate the constraint against JSON encoded application schema data.

NOTE	XML Schema based validation also does not cover the aforementioned checks. In general, it is up to the application how the remaining checks are realized. They could be implemented with application specific logic. However, for some checks, additional validation tools may be available. For example, in order to check requirements defined by OCL constraints, Schematron rules can be derived from these constraints, and evaluated against XML encoded data. At the time when this report was written, a similarly powerful tool to perform additional checks on JSON data did not seem to be available. However, there were some developments in that regard (e.g., jsontron [https://amer-ali.github.io/jsontron]), which could be useful starting points for future work on this topic.

5.1.2. Conversion of JSON data to RDF using JSON-LD

JSON data certainly is of interest to typical web applications. It can also be of interest to the semantic community. With JSON-LD, it is possible to convert JSON data to RDF data. The conversion thereby defines the semantics of the JSON data. The resulting RDF data can be used by semantic applications. The definition of semantics of JSON data through the use of JSON-LD has been investigated in OGC Testbed-14. For further details, also on limitations of that approach, see the OGC Testbed-14: Application Schemas and JSON Technologies Engineering Report.

UGAS-2020 investigated the implications of a few JSON Schema conversion rules regarding the ability for mapping JSON data (that complies to these rules) to RDF using JSON-LD. **It is recommended that a thorough analysis of mapping JSON data to RDF using JSON-LD be conducted in a future activity, specifically taking into account the JSON Schema conversion rules developed in UGAS-2020. That activity should investigate how JSON-LD context documents can automatically be derived from an application schema, given a set of ontology and JSON Schema encoding rules that apply to the application schema.** The analysis should be accompanied by a prototypical implementation.

The following paragraphs summarize the results of the investigations on JSON-LD that were conducted in UGAS-2020.

The JSON Schema conversion of code lists describes three different ways to encode code lists and thus also code values.

- By default, code lists are converted to a simple JSON Schema "type" definition, typically a string. Using a tagged value, it is also possible to use other JSON Schema types, for example a number (for a list of numeric codes).
- Using *rule-json-cls-codelist-uri-format*, codes will be represented as JSON strings, with format "uri."
- Finally, using *rule-json-cls-codelist-link*, it is also possible to convert a code list using a JSON Schema reference to a link object.

In the first two cases, a code is represented by a simple JSON value, while in the case of a link object, a code is represented by a JSON object.

The cases where a code is represented by anything other than a simple JSON string are problematic when using JSON-LD to map JSON data to RDF. The reason is that, following the ISO 19150-2

conversion rule for code lists (as documented in the [OGC Testbed-12 ShapeChange Engineering Report [http://docs.opengeospatial.org/per/16-020.html#_code_lists]]), codes are represented in RDF/OWL as individuals - which are directly used as the RDF/OWL property that represents a code list valued UML property.

If the JSON representation of a code is a JSON number, then that number cannot be mapped to RDF using JSON-LD. That is a limitation of JSON-LD, which is documented in more detail in the OGC Testbed-14: Application Schemas and JSON Technologies Engineering Report [http://docs.opengeospatial.org/per/18-091r2.html#JSON_LD_NEO_issues_numeric_code_values], section 6.2.2.2.

If the JSON representation of a code is a link object, with one of the members of the (JSON) link object containing the code value, then the link object cannot be mapped to a simple RDF/OWL individual using JSON-LD. Instead, the mapping result would be an RDF resource that represents the link object, with a property that represents the member that contains the code value. Consider the example shown in Listing 2.

Listing 2. JSON LD for mapping a property with a link object encoding a code value

```
{
 "@context": {
  "@version": 1.1,
  "ex": "http://example.com/myontology",
  "excode": "http://example.com/myontology/mycodelist",
  "codeValuedProperty": {
   "@id": "ex:myCodeProperty",
   "@context": {
    "codeList": "@type",
    "codeListValue": {
     "@id": "ex:code",
     "@type": "@vocab"
    },
    "a": "excode:codeA",
    "b": "excode:codeB"
   }
  }
 },
 "codeValuedProperty": {
  "codeList": "http://example.com/myontology/mycodelist",
  "codeListValue": "a"
 }
}
```

A link object always has a specific structure, which in this example is related to the encoding of code values in ISO 19139. As we can see, the code value is encoded within the "codeListValue" member. Running this example in the JSON-LD playground [https://json-ld.org/playground/index.html], we get the N-Quads representation shown in Listing 3.

NOTE	N-Quads is one of several formats for encoding RDF data.

Listing 3. RDF data resulting from JSON-LD based mapping of a link object

```
_:b0 <ex:myCodeProperty> _:b1 .
_:b1 <ex:code> <excode:codeA> .
_:b1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://example.com/myontology/mycodelist> .
```

We can see that the code value is encoded as property value of a nested resource (blank node b1), instead of being the actual value of ex:myCodeProperty, which would be expected for the RDF/OWL encoding that is based on the ISO 19150-2 conversion rule for code lists.

This mismatch can be prevented if the code value was a simple JSON string, see Listing 4, and the resulting N-Quads in Listing 5.

Listing 4. JSON LD for mapping a property with a simple string encoding the code value

```
{
 "@context": {
  "@version": 1.1,
  "ex": "http://example.com/myontology",
  "excode": "http://example.com/myontology/mycodelist",
  "codeValuedProperty": {
   "@id": "ex:myCodeProperty",
   "@type": "@vocab",
   "@context": {
    "a": "excode:codeA",
    "b": "excode:codeB"
   }
  }
 },
 "codeValuedProperty": "a"
}
```

Listing 5. RDF data resulting from JSON-LD based mapping of the string encoded code value

```
_:b0 <ex:myCodeProperty> <excode:codeA> .
```

With the code value being a uri encoded as a JSON string which is the same IRI as the individual that identifies the code in the ontological representation, the JSON data would be even easier to map to RDF using JSON-LD - see Listing 6, which results in the same RDF data as shown in Listing 5.

Listing 6. JSON LD for mapping a property with a uri encoding the code value

```
{
 "@context": {
  "@version": 1.1,
  "ex": "http://example.com/myontology",
  "codeValuedProperty": {
   "@id": "ex:myCodeProperty",
   "@type": "@id"
  }
 },
 "codeValuedProperty": "http://example.com/myontology/mycodelist/codeA"
}
```

5.1.3. Continue work on a JSON encoding for features

The Features Core Profile and its JSON encoding build on the widely used GeoJSON standard and extends it with minimal extensions to support additional concepts that are important for the wider geospatial community and the OGC API standards, including support for solids, coordinate reference systems, and time intervals. A new Standards Working Group for a standard OGC Features and Geometries JSON [https://portal.ogc.org/files/?artifact_id=95319&version=1] has been proposed.

The design of the JSON encoding should be:

- validated through multiple implementations writing and reading such JSON;
- validated through additional experiments with extensions to ensure its extensibility; and
- kept consistent with the JSON Schema encoding rule for GeoJSON that has been specified in this ER.

5.1.4. Investigate the impact of OCL language constructs unsupported by SHACL for the NAS

Three approaches for translating OCL constraints using SHACL have been investigated in UGAS-2020. None of them supports a full translation of all OCL language constructs that were investigated. It would be useful to analyze the frequency of use of non-supported OCL language constructs and/or the significance/impact of avoiding their use in the NAS. The results would help make a decision of which of the three approaches would be best suited for implementation, in support of encoding the knowledge contained in NAS OCL constraints in SHACL.

5.1.5. Revise the OWL conversion rule for basic types

rule-owl-cls-encode-basictypes, an OWL conversion rule for basic types developed in OGC Testbed-12 [http://docs.opengeospatial.org/per/16-020.html#rdf_cr_class_basictype], converts basic types - i.e., classes that inherit (directly or indirectly) from a simple type (e.g., CharacterString) - to OWL classes. That means that a basic type value always needs to be encoded as an instance. However, whenever the supertype of a basic type is mapped to an RDFS/OWL datatype, one would expect that a value of this basic type is encoded as a literal (with the according datatype).

The conversion rule should be revised, and the implementation in the ShapeChange ontology target be updated accordingly.

A basic type with a supertype that maps to an RDFS/OWL datatype should be defined as a custom datatype [https://www.w3.org/TR/owl-syntax/#Datatype_Definitions], which restricts the XSD datatype of the RDFS/OWL datatype using an OWL DatatypeRestriction. That restriction defines the restricting facets (defined for the basic type, typically using tagged values), such as xsd:minInclusive.

A custom datatype can be used in the range declaration of an OWL data property. However, a custom datatype cannot be used within an OWL DatatypeRestriction because "datatypes defined by datatype definition axioms support no facets" (source: https://www.w3.org/TR/owl-syntax/#Datatype_Definitions). That means that a basic type A, which is defined in UML as a subtype of another basic type B, cannot be expressed using B as datatype in the OWL DatatypeRestriction. Instead, basic type A needs to be defined as a datatype with an OWL DatatypeRestriction that combines all restricting facets from its direct and indirect supertypes (here: B) and the restrictions declared for basic type A itself.

Example:

- A type *PositiveReal*, which inherits from *Real* and restrict the value space to non-negative double values (so including zero), would be represented as: `DatatypeDefinition(<PositiveReal> DatatypeRestriction(xsd:double xsd:minInclusive "0"^^xsd:double))`
- A type *Real0to400*, which inherits from *PositiveReal* and restricts the value space to [0,400], would be represented as: `DatatypeDefinition(<Real0to400> DatatypeRestriction(xsd:double xsd:minInclusive "0"^^xsd:double xsd:maxInclusive "400"^^xsd:double))`

NOTE	ShapeChange map entries for RDFS/OWL datatypes will need to provide the XSD datatype (e.g., via a map entry parameter) that is used by the RDFS/OWL datatype, plus information about any restricting facets, so that the custom datatype definition can be created. Alternatively, ShapeChange could try to parse that information from the vocabulary definition of the RDFS/OWL datatype. This requires further investigation and actual testing.

NOTE	Subtypes of type Measure, defined in an application schema with the intent to define some restrictions (e.g., restricting the value range from 0 to 360), would only qualify for this kind of encoding in OWL if Measure was mapped to an RDFS/OWL datatype.

Chapter 6. UML to JSON Schema Encoding Rule

6.1. Overview

ISO / TC 211 defines Standards in the field of digital geographic information. A couple of these Standards, especially ISO 19109, are used by the geospatial community to define so called application schemas. An application schema is a conceptual schema for data required by one or more applications. It is typically defined using the Unified Modeling Language (UML).

OGC 07-036r1 defines rules for encoding an application schema in XML. The result is an XML Schema, which defines the structure for encoding application data in XML. Applications would use this XML as a format for interoperable information exchange.

XML has been and still is a widely used format for encoding data on the web. However, web applications also use other formats. JSON is another prominent format for encoding and exchanging data on the web. The ISO / TC 211 standards do not define rules for encoding an application schema in JSON. The main reason for this gap likely is that in the past, XML was a more prominent format than JSON for (geospatial) web service interactions. Another reason could be that a schema language for JSON has not fully been standardized yet. JSON Schema is such a language. The JSON Schema specification - a draft IETF standard - has improved considerably over the past couple of years. It is a serious candidate for the definition of rules for encoding application schema data in JSON.

NOTE

The current version of JSON Schema, in early 2020, is draft 2019-09. It consists of three documents, JSON Schema core [https://tools.ietf.org/html/draft-handrews-json-schema-02] ([1]), a schema for validation [https://tools.ietf.org/html/draft-handrews-json-schema-validation-02] ([2]), and a hyper schema [https://tools.ietf.org/html/draft-handrews-json-schema-hyperschema-02] ([3]). For the analysis in UGAS-2020, the first two documents were of primary interest. Note also that the website http://www.json-schema.org provides a list of the current and older drafts of the specification, as well as the latest unreleased version [http://json-schema.org/specification-links.html]. The site also provides an overview of existing JSON Schema implementations [http://json-schema.org/implementations.html].

In UGAS-2020, JSON Schema draft 2019-09 was analyzed and a set of rules as well as recommendations for converting an application schema in UML to JSON Schema has been developed. These rules and recommendations are documented in the following sections.

6.2. Schema Conversion Rules

The following subsections describe a number of conversion rules, which define how the content of an application schema, represented using UML as the conceptual schema language, is converted to JSON Schema.

NOTE	An encoding rule consists of a set of conversion rules – as required by a community. The Encoding Rules section describes two such rules - one for a GeoJSON compliant encoding, and one for a plain JSON encoding.

6.2.1. Documentation

With *rule-json-all-documentation*, descriptive information of application schema elements (packages, classes, properties, and associations) can be encoded via JSON Schema *annotations*.

NOTE	*Annotations* represent one category of JSON Schema keywords (for further details, see JSON Schema core, section 7 [https://tools.ietf.org/html/draft-handrews-json-schema-02#section-7]). *Annotations* attach information that applications may use as they see fit. The other categories are *assertions*, which validate that a JSON instance satsifies constraints, and *applicators*, which apply subschemas to parts of the instance and combine their results.

WARNING	In UGAS-2020, only the design for converting the documentation of application schema elements has been developed. *rule-json-all-documentation* has not been implemented in UGAS-2020, for two reasons. 1. Omitting the documentation will result in significantly smaller JSON Schema documents. The reduction of file size is preferable for processes that need to download the schema in order to apply validation. This is even more important if cross-references between JSON Schemas exist. 2. When validating JSON data against a JSON Schema, a JSON Schema validator typically focuses on the JSON Schema assertions and applicators, and will ignore most JSON Schema annotations - especially meta-data annotations [https://tools.ietf.org/html/draft-handrews-json-schema-validation-02#section-9], such as "title" and "description."

Descriptive information of a model element in ShapeChange, i.e., properties (attributes and association roles), classes, and packages, includes the pieces of information, called *descriptors*, that are documented in Table 1.

NOTE	A model element can have all, a subset, or none of these descriptors.

Table 1. Well-known descriptors

Descriptor Name (and ID)	Explanation
Name (name)	The name of the model element (as named in the source UML, i.e., using upper and lower camel case).
Alias (alias)	An alternative, human-readable name for the model element.

Descriptor Name (and ID)	Explanation
Definition (definition)	The normative specification of the model element.
Description (description)	Additional information about the model element.
Documentation (documentation)	The overall documentation of the model element. May be structured, containing other descriptors (such as definition and description).
Example(s) (example)	Example(s) illustrating the model element.
Global identifier (globalIdentifier)	The globally unique identifier of the model element; that is, unique across models.
Legal basis (legalBasis)	The legal basis for the model element.
Data capture statement(s) (dataCaptureStatement)	Statement(s) describing how to capture instances of this model element from the real world.
Primary code (primaryCode)	The primary code for this model element. **NOTE** The main code for a model element should be assigned to this descriptor. The primary code may be the only one. Optional additional tagged values may be added for other codes.

NOTE The descriptor ID is used in ShapeChange configuration elements that define JSON Schema annotations.

Typically, a community has a preferred way to model and encode this information. For example, one community may want to encode the description of a model element via the "description" annotation (which is one of a set of basic meta-data annotations defined in JSON Schema validation, section 9 [https://tools.ietf.org/html/draft-handrews-json-schema-validation-02#section-9]), while another may prefer to encode the values of multiple descriptors of a model element within a single "description" annotation.

ShapeChange can support this type of diversity through *JSON Schema annotation* elements, which can be defined in the ShapeChange configuration. An annotation element specifies how the content of a specific JSON Schema annotation (that shall be generated while converting a model element) shall be constructed. The annotation element takes into account that a UML model element may not have an actual value for a descriptor, and that some descriptors can have multiple values, e.g., the descriptor *example*.

In addition to the well-known descriptors (see previous table), additional descriptive information

can be incorporated through UML tagged values from the application schema. For example, the "name" tagged value on classes in the NAS could be used to create a JSON Schema "title" annotation.

Different types of annotation elements are available for configuring a ShapeChange JSON Schema target.

- JsonSchemaNumberAnnotation - For annotations with a(n array of) JSON number(s) as value.
- JsonSchemaBooleanAnnotation - For annotations with a(n array of) JSON boolean(s) as value.
- JsonSchemaStringAnnotation - For annotations with a(n array of) JSON string(s) as value.
- JsonSchemaTemplateAnnotation - For annotations with a(n array of) JSON string(s) as value, defined via a template that can include multiple descriptors and tagged values.

NOTE The JSON Schema annotation "examples," defined by JSON Schema validation, section 9.5 [https://tools.ietf.org/html/draft-handrews-json-schema-validation-02#section-9.5], is an example for an annotation that has a JSON array as value, with the type of array items being unrestricted. In other words, the array can contain mixed value types. The "examples" annotation can thus have an array of strings (e.g., ["abc","xyz"]), numbers (e.g., [4,2]), booleans (e.g., [true, true]), and a mix thereof (e.g., ["abc", 2, true]) as value.

NOTE ShapeChange JSON Schema annotation elements are not designed to support the creation of annotations with complex JSON arrays or objects as value. Only simple values, or an array thereof, can be created. So far, no use cases have been identified that require a more complex annotation value. In the future, if such use cases were identified, ShapeChange could be extended to support them.

The following two tables document the structure of ShapeChange JSON Schema annotation elements. Listing 7 provides examples.

Table 2. ShapeChange JSON Schema annotation element - for annotations with string, number, or boolean values

Configuration Information Item	Datatype & Structure	Required / Optional	Default Value	Description
annotation	string	Required	*not applicable*	Name of the JSON Schema annotation keyword that shall be added to the JSON Schema element which represents the UML model element.

Configuration Information Item	Datatype & Structure	Required / Optional	Default Value	Description
descriptorOrTaggedValue	string	Required	*not applicable*	Either a *descriptor-ID*, identifying one of the well-known descriptors, or a string identifying a tagged value. In order to identify a tagged value, add prefix "TV:" to the name of the tagged value. If a tagged value is known to contain a list of values, combined in a string using a specific separator, and these values shall be used as individual values, rather than using the whole string as value, use the prefix "TV(*separator*):," followed by the tag name. ShapeChange will then split the tagged value around matches of the given separator (which is treated as a literal). Note that the type of the ShapeChange JSON Schema annotation element defines how ShapeChange will encode the values of the descriptor / tagged value. • In case of a string annotation, each value will be quoted. • In case of a number annotation, each value will be encoded as-is, i.e., without quotes. • In case of a boolean annotation, each value will be parsed: if the value is "True" (ignoring case") or 1, then the value will be encoded as the JSON value *true*; otherwise it will be encoded as the JSON value *false*.
noValueBehavior	enum: *ignore* or *populateOnce*	Optional	*ignore*	Determines the behavior in case that no value is available for the descriptor or tagged value. • *ignore*: No annotation is created. • *populateOnce*: A single annotation is created, with the *noValueValue* being used as value.
noValueValue	string	Optional	*the empty string*	If the descriptor or tagged value has no value, then this information item provides the value to use instead (e.g., 0, or true).

Configuration Information Item	Datatype & Structure	Required / Optional	Default Value	Description
arrayValue	boolean	Optional	*false*	If true, then the annotation value will always be encoded as an array, even if only a single value is present. Otherwise, the default behavior is to only encode multiple values within a JSON array.

Table 3. ShapeChange JSON Schema annotation element - for annotations with string values, based on templates

Information Item	Datatype & Structure	Required / Optional	Default Value	Description
annotation	as defined in the previous table			
valueTemplate	string	Required	*not applicable*	Textual template where an occurrence of the field "[[*descriptor-ID*]]" is replaced with the value(s) of that descriptor. The IDs of supported descriptors are listed in the table above. An occurrence of the field "[[TV:name]]" is replaced with the value(s) of the UML tagged value with the given name from the input schema. The content of a tagged value can also be split into multiple parts. In that case, use field "[[TV(*separator*):name]]." The tagged value will be split around matches of the given separator (which is treated as a literal).
noValueBehavior	enum: *ignore* or *populateOnce*	Optional	*ignore*	Determines the behavior in case that no value is available for any of the fields (tagged values and descriptors) contained in the template. • *ignore*: No annotation is created. • *populateOnce*: A single annotation is created, with the *noValueValue* being used for all fields.
noValueValue	string	Optional	*the empty string*	If a descriptor used in a template has no value, then this information item provides the value to use instead (e.g., "N/A" or "FIXME").

Information Item	Datatype & Structure	Required / Optional	Default Value	Description
arrayValue	as defined in the previous table			
multiValueBehavior	enum: either *connectInSingleAnnotationValue* or *createMultipleAnnotationValues*	Optional	*connectInSingleAnnotationValue*	Specifies how a case where one or more of the descriptors and tagged values contained in the template have multiple values, shall be encoded. • *connectInSingleAnnotation*: Multiple values of a descriptor or tagged value contained in the template are combined in a single string value, using the *multiValueConnectorToken* to connect them. • *createMultipleAnnotationValues*: Multiple values for one or more descriptor or tagged value result in an array of annotation values, with one value for each combination of multi-valued descriptors / tagged values (resulting in a permutation of the values of each descriptor / tagged value contained in the template).
multiValueConnectorToken	string	Optional	*a single space character*	If a descriptor or tagged value used in the *valueTemplate* has multiple values, and the *multiValueBehavior* is set to *connectInSingleAnnotationValue*, then the values are concatenated to a single string value using this token as connector between two values.

NOTE Conversion rules exist to populate "default" and "readOnly". For further details, see sections Fixed / readOnly and Initial Value.

Listing 7. Example of ShapeChange JSON Schema annotation elements

```
<annotations>
 <JsonSchemaBooleanAnnotation annotation="deprecated"
descriptorOrTaggedValue="TV:deprecated"/>
 <JsonSchemaNumberAnnotation annotation="code"
descriptorOrTaggedValue="TV:codeNumber"/>
 <JsonSchemaStringAnnotation annotation="title" descriptorOrTaggedValue="alias"
noValueBehavior="populateOnce" noValueValue="NA"/>
 <JsonSchemaStringAnnotation annotation="label"
descriptorOrTaggedValue="TV(|):aliasList"/>
 <JsonSchemaStringAnnotation annotation="examples" descriptorOrTaggedValue="example"
arrayValue="true"/>
 <JsonSchemaTemplateAnnotation annotation="description" valueTemplate="Definition:
[[TV:definition]]  Description: [[TV:description]]" noValueValue="[None
Specified]"/>
 <JsonSchemaTemplateAnnotation annotation="isDefinedBy"
valueTemplate="http://nsgreg.nga.mil/as/view?i=[[TV:itemIdentifier]]"/>
</annotations>
```

6.2.2. Schema Packages

Schema packages have the stereotype <<applicationSchema>>, <<schema>>, or an alias (e.g., using a specific language, like <<anwendungsschema>>). An <<applicationSchema>> package represents an application schema according to ISO 19109. The stereotype <<schema>> has been introduced for packages that should be treated like application schemas, but do not contain feature types.

6.2.2.1. Definitions Schema

A UML application schema and its classes are converted into one or more so-called *definitions schemas*. A definitions schema is a JSON Schema that has the "$defs" keyword.

NOTE	In the JSON Schema specification draft, version 07, the keyword of the definitions section was "definitions". In JSON Schema specification draft, version 2019-09, the keyword was changed to "$defs."

The "$defs" keyword has a JSON object as value, where each member represents the JSON Schema definition of a class from the application schema.

Listing 8. JSON Schema example of a definitions schema

```
{
  "$schema": "http://json-schema.org/draft/2019-09/schema",
  "$defs": {
    "Class1": {
      "type": "object",
      "properties": {
        "prop1": {"type": "string"}
      },
      "required": ["prop1"]
    },
    "Class2": {
      "type": "object",
      "properties": {
        "prop2": {"type": "number"}
      },
      "required": ["prop2"]
    }
  }
}
```

NOTE The current definitions schema examples in this document mostly use "definitions" instead of "$defs." The reason is that most JSON Schema validators support JSON Schema draft-07, but at the time when the examples were developed, these validators did not fully support JSON Schema 2019-09.

By default, a UML application schema is encoded as a single definitions schema. By setting tagged value *jsonDocument* on subpackages of the schema package, the classes within these packages (as well as subpackages for which the tagged value is not set) will be encoded in additional definitions schemas.

NOTE Tagged value *jsonDocument* can also be set on the schema package itself, to define the file name of the definitions schema that will be produced for the schema package. If tagged value *jsonDocument* is not defined for the schema package, or does not have a value, then ShapeChange will use the package name as fallback, replacing all spaces and forward slashes with underscores, and appending '.json'. For example, if a schema package was named 'Ba / nanas' then the file name would be 'Ba___nanas.json'.

NOTE

If the conversion process does not add any actual definitions to a definitions schema, then that schema will not be written by ShapeChange. Reasons for no definition being added to a definitions schema are:

- The package represented by the definitions schema is empty;
- No type is directly defined in the package, and subpackages with types have their own definitions schema; and/or
- None of the types whose definition would be added to the definitions schema is actually encoded. That can be the case if a type is of a category that is not encoded by default (e.g., a union), and no conversion rule for converting the type is included in the encoding rule. Another case would be that the rule to not encode a type (*rule-json-all-notEncoded*) applies to the model element.

Figure 1 provides an example of a UML application schema, where the tagged value is set both on the schema package itself and on one of its leaf packages. Figure 2 illustrates the structure of the resulting definitions schemas.

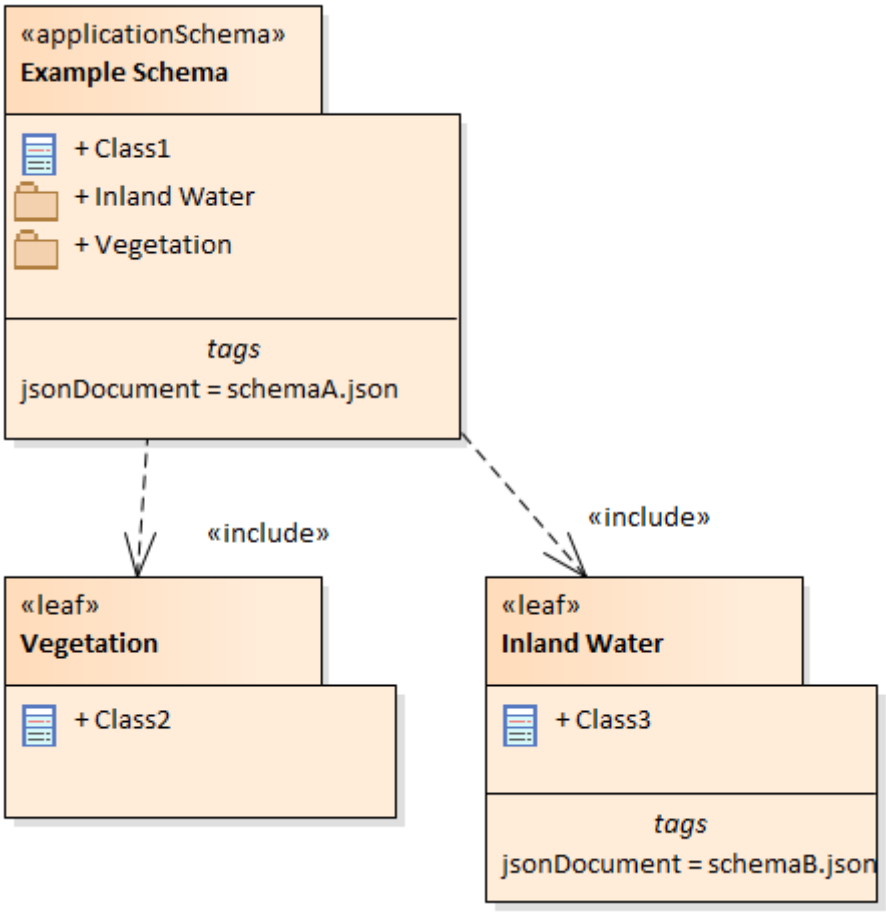

Figure 1. Example of a UML application schema with tagged value jsonDocument set

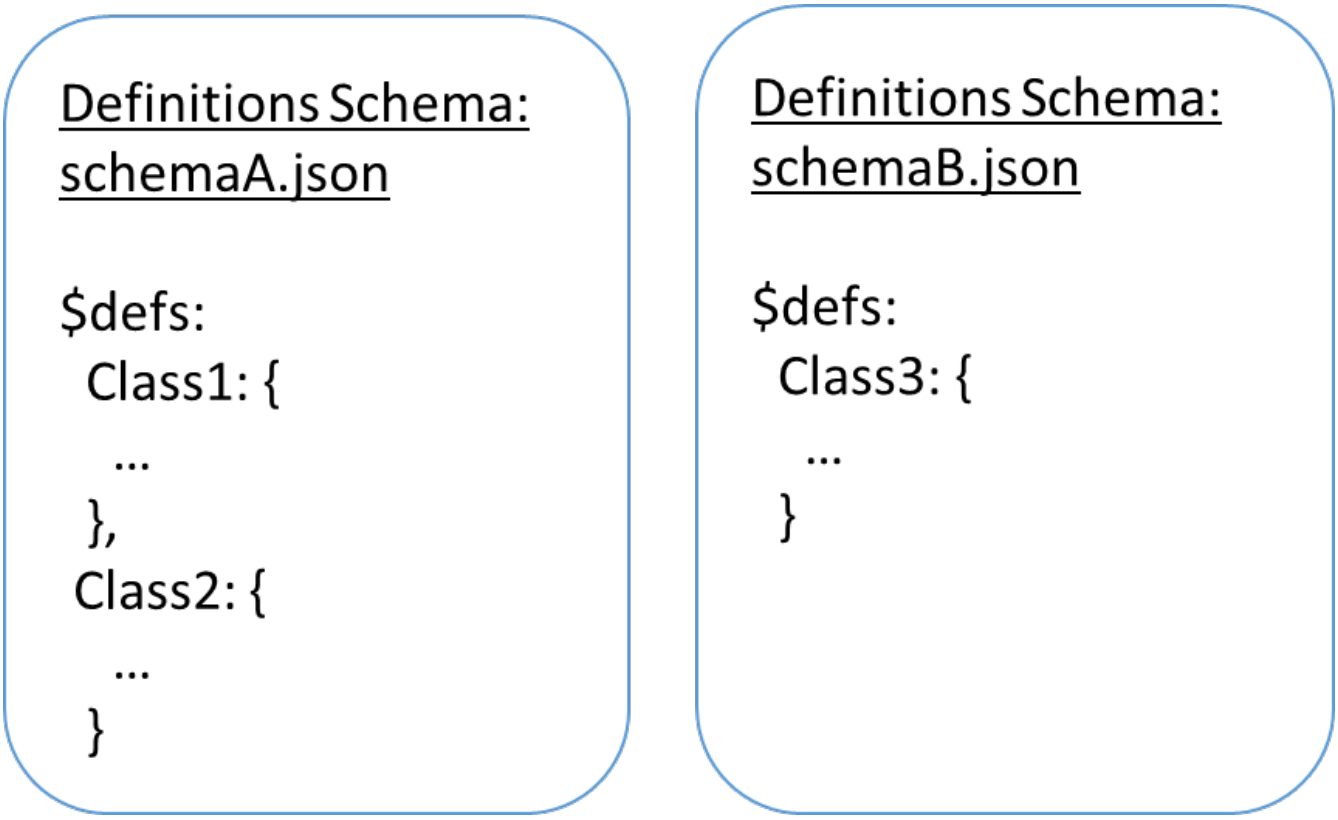

Figure 2. Deriving JSON Schemas from an application schema

References from types of the application schema to other types (within the same or within an external schema) are converted as references to the according definitions schemas, using the JSON Schema keyword "$ref" - see Figure 3.

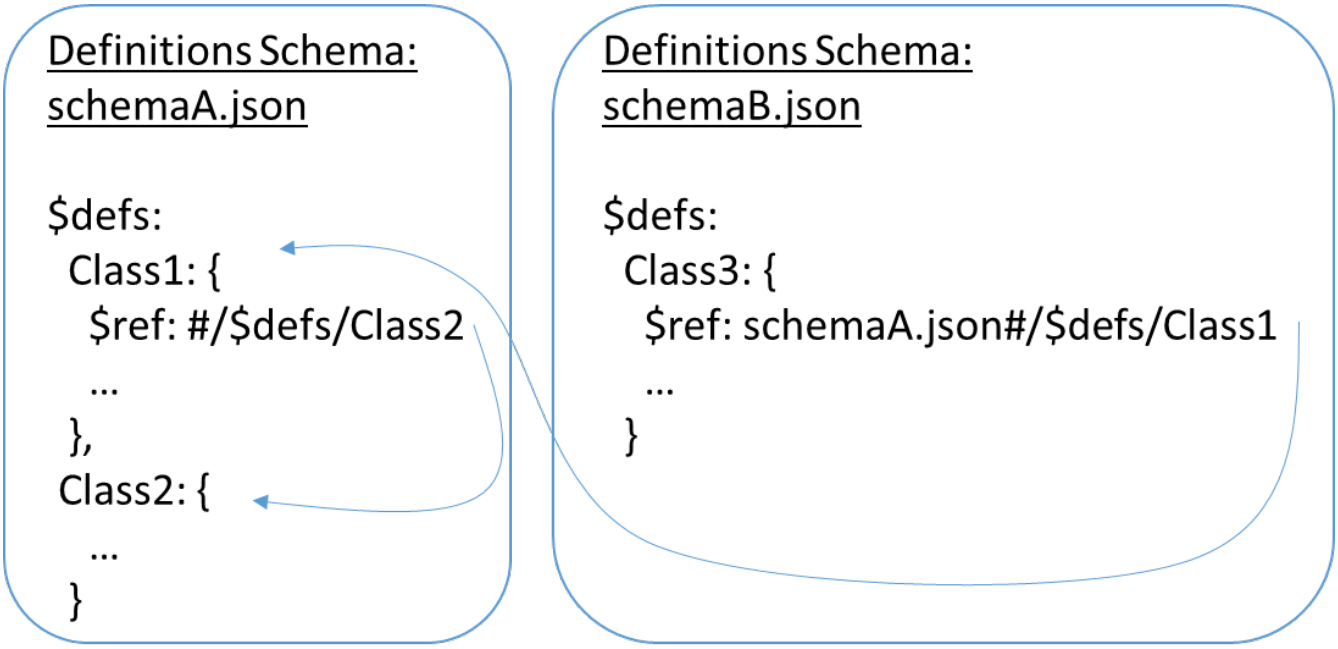

Figure 3. References between JSON Schemas using $ref

A link to a particular definition within a definitions schema requires the use of a JSON Pointer or an anchor in the fragment identifier of the link URL.

JSON Pointer [https://tools.ietf.org/html/rfc6901#section-6], chapter 6, explicitly states that the media type in which a JSON value is provided needs to support this kind of fragment identifier, and that this is

not the case for the media type application/json. If a JSON Schema was published with this media type, then it is possible that the application ignores a fragment identifier (because the media type does not support fragment identifiers).

Definitions schemas therefore should not be published under media type application/json. Instead, a JSON Schema should be published with media type application/schema+json - which is defined by the JSON Schema specification. The media type application/schema+json supports JSON Pointers and plain names as fragment identifiers. For further details, see JSON Schema core, chapter 5 [https://tools.ietf.org/html/draft-handrews-json-schema-02#section-5].

NOTE	The JSON Schema that shall be used to validate a JSON document cannot be identified within that document itself. In other words, JSON Schema does not define a concept like an xsi:schemaLocation, which is typically used in an XML document to reference the applicable XML Schema(s). Instead, JSON Schema uses link headers and media type parameters to tie a JSON Schema to a JSON document (for further details, see JSON Schema core [https://tools.ietf.org/html/draft-handrews-json-schema-02], sections 11.1 and 11.2). The relationship between a JSON document and the JSON Schema for validation can also be defined explicitly by an application.

6.2.2.2. JSON Schema Version

According to JSON Schema core [https://tools.ietf.org/html/draft-handrews-json-schema-02#section-8.1.1], section 8.1.1, the root schema of a JSON Schema document should contain a "$schema" keyword. The value of this keyword identifies the JSON Schema meta-schema against which the schema is valid. Typically, that is a meta-schema defined by a specific version of the JSON Schema specification.

The "$schema" keyword is therefore added to the definitions schema. Its value is defined via the ShapeChange JSON Schema target configuration parameter *jsonSchemaVersion*. The values supported for the parameter are:

- "2019-09" (the default value of the parameter) - corresponding to the schema URI "https://json-schema.org/draft/2019-09/schema"
- "draft-07" - corresponding to the schema URI "http://json-schema.org/draft-07/schema#"
- "OpenApi30" - with no schema URI; introduced to support the OpenAPI 3.0 Schema object, which will become obsolete once OpenAPI 3.1 has been adopted (for further details, see JSON Schema variants)

NOTE	The ShapeChange JSON Schema target supports both JSON Schema 2019-09, and the older draft 07, because implementation support for the latter was more prevalent at the time when the JSON Schema work was conducted in UGAS-2020.

NOTE	The "$schema" of the definitions schema examples in this document is mostly set to "http://json-schema.org/draft-07/schema#." The reason is that most JSON Schema validators support JSON Schema draft-07, but at the time when this chapter was written they did not fully support JSON Schema 2019-09.

6.2.2.3. Schema Identifier

According to JSON Schema core [https://tools.ietf.org/html/draft-handrews-json-schema-02#section-8.2.2.1], section 8.2.2.1, the root schema of a JSON Schema document should contain an "$id" keyword with an absolute URI. The "$id" identifies the schema resource with its canonical URI.

NOTE	The URI is an identifier and not necessarily a resolvable URL. If the "$id" is a URL, there is no expectation that the JSON Schema can be downloaded at that URL. However, it is recommended that the URL is stable, persistent, and globally unique.

The definitions schemas derived from the application schema package thus each receive a unique "$id." The value of this id uses the following URI template.

```
{jsonBaseUri}/{jsonDirectory}/{jsonDocument}
```

Where:

- *{jsonBaseUri}* is either specified via tagged value *jsonBaseUri* on the application schema package, or defined via the ShapeChange JSON Schema target configuration parameter *jsonBaseUri* (which has default value "http://example.org/FIXME"). If both are defined, the tagged value takes precedence over the configuration parameter.
- *{jsonDirectory}* is the value of the tagged value of the same name on the application schema package. If that tagged value is undefined, the value of the *xmlns* tagged value is used. If that tagged value is also not defined, then the string *default* is used.
- *{jsonDocument}* is the file name of the definitions schema, which is either defined in the UML model using tagged value of the same name on a package, or automatically derived from the application schema name.

Example: With *jsonBaseUri* = `https://example.org`, *jsonDirectory* = `json/schemas/schemaX/1.0`, and *jsonDocument* = `testschema.json`, ShapeChange will produce:

```
"$id": "https://example.org/json/schemas/schemaX/1.0/testschema.json"
```

NOTE	The "$id" of the definitions schema is not included in other examples within this chapter, because declaring an absolute, non-existent URL in these examples often prevents JSON Pointers from these examples from working when testing the examples, for instance on https://www.jsonschemavalidator.net/ (which is a useful tool for testing JSON Schema).

6.2.3. Types

6.2.3.1. Mappings

Application schemas typically use types from other schemas, for example the types defined by ISO 19103 and ISO 19107. External types can be used as value types of properties, and as supertypes for types defined in the application schema that is being converted.

Whenever an external type is used, its JSON Schema definition is needed. Either an external type is implemented as one of the simple JSON value types (e.g., string - maybe with a certain format), or it is defined by a particular JSON Schema. In case of a JSON Schema, the URL of that schema needs to be known during the conversion process. If the schema is a definitions schema, then the URL needs to be augmented with a fragment identifier that includes a JSON Pointer or an anchor reference within the schema.

Information about the JSON Schema implementation of external types must explicitly be provided to the ShapeChange JSON Schema target via so called map entries. These map entries are part of the target configuration. A map entry of the JSON Schema target must:

- identify the schema type that is being mapped, by name
- define the JSON Schema implementation of that type:
  - either as one of the few simple JSON value types (string, number, integer, boolean), potentially with additional keywords conveyed via map entry parameter:
    - for any simple JSON value type: keyword *format*
    - for JSON value type *string*: keywords *enum*, *const*, *pattern*, *maxLength*, *minLength*
      - NOTE: Complex regular expressions intended to be used as *pattern* may need to be base64 encoded, in order to avoid problems with syntax rules of the map entry parameter. For base64 encoded regular expressions, use the *patternBase64* characteristic of the ShapeChange map entry parameter *keywords*.
    - for JSON value types *integer* and *number*: keywords *enum*, *const*, *multipleOf*, *maximum*, *minimum*, *exclusiveMaximum*, *exclusiveMinimum*
  - or as a URL that references the JSON Schema definition of the external type
- declare the path to the JSON member that is used to encode the name of the type that the JSON object represents, (e.g., "type", "entityType", or "properties/observationType") - if such a member exists in the target type
  - This information is useful to support specialization (for further details, see Class Specialization and Property Ranges) and encoding value type options.

The Features Core Profile of Key Community Conceptual Schemas chapter documents a core profile of ISO schemas that are used in the NSG, as well as the JSON Schema definitions for the types contained in that profile. In UGAS-2020, type mappings have been created for this profile, in order to create a NAS JSON Schema. The mappings can also be used for the conversion of other application schemas.

6.2.3.2. Class Name

The following use cases have been identified where converting the name of a type is useful.

- Defining location independent identifiers within the definitions schema, to create simple references to schema definitions.
- Supporting type identification, thereby enabling at least some level of type inheritance checks and semantic mapping.

6.2.3.2.1. Location Independent Schema Identifiers

With *rule-json-cls-name-as-anchor*, the name of a class is encoded as an "$anchor," which is added at the start of the schema definition of the class (within the definitions schema). Schema definitions that have an "$anchor" can be referenced using the plain text value of the anchor as fragment identifier, instead of using a more complex JSON Pointer.

NOTE The "$anchor" keyword was added in JSON Schema draft 2019-09. It replaces the somewhat ambiguous use of the "$id" keyword in JSON Schema draft 07 to define plain name fragment identifiers for subschemas. For further details, see section 8.2.3 of both JSON Schema draft 2019-09 [https://tools.ietf.org/html/draft-handrews-json-schema-02#section-8.2.3] and JSON Schema draft 07 [https://tools.ietf.org/html/draft-handrews-json-schema-01#section-8.2.3].

Listing 9. JSON Schema (version 2019-09) example of location independent schema identifiers

```
{
  "$schema": "http://json-schema.org/draft/2019-09/schema",
  "$defs": {
    "TypeA": {
      "$anchor": "TypeA",
      "...": "..."
    },
    "TypeB": {
      "$anchor": "TypeB",
      "...": "..."
    }
  }
}
```

If the ShapeChange target parameter *jsonSchemaVersion* (see JSON Schema Version) is set to "draft-07," then *rule-json-cls-name-as-anchor* results in the creation of the "$id" keyword, instead of the "$anchor" keyword.

NOTE JSON Schema draft 07 requires the value of "$id" to start with "#", thus when producing a JSON Schema compliant to JSON Schema draft 07, the combination of "#" and the class name is used as value of the "$id" key.

Listing 10. JSON Schema (draft 07) example of location independent schema identifiers

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "definitions": {
    "TypeA": {
      "$id": "#TypeA",
      "...": "..."
    },
    "TypeB": {
      "$id": "#TypeB",
      "...": "..."
    }
  }
}
```

6.2.3.2.2. Type Identification

rule-json-cls-name-as-entityType adds another JSON member to the JSON object which represents the class that is being converted.

The name of the JSON member can be configured using the ShapeChange JSON Schema target parameter *entityTypeName*. The default value of the parameter is "entityType". The JSON member is required and string-valued. It should be used to encode the name of the type that is represented by the JSON object.

NOTE	By default, the property value is not restricted using "const", because doing so would prevent JSON Schema constraints that support inheritance-related checks. However, if the application schema did not use inheritance, then such restrictions could be defined.

Listing 11. JSON Schema example with property "entityType" used for identifying the type of a JSON object

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "definitions": {
    "Type": {
      "properties": {
        "entityType": {
          "type": "string"
        },
        "property": {
          "type": "string"
        }
      },
      "required": [
        "entityType", "property"
      ]
    }
  },
  "$ref": "#/definitions/Type"
}
```

The following JSON instance is valid against the schema:

```
{
  "entityType": "Type",
  "property": "x"
}
```

Encoding the type name in JSON objects is useful.

- Encoding the type of a JSON object together with its other properties supports a more complete validation of property values, where the property type is a supertype. For further details, see Value Type.
- As described in chapter 6 of the OGC Testbed-14: Application Schemas and JSON Technologies Engineering Report [http://docs.opengeospatial.org/per/18-091r2.html#JSON_LD], having a key within a JSON object with a string value that identifies the type of the object allows that object to be mapped to RDF. More specifically, the string value can be mapped to an IRI that identifies the type of an RDFS resource.

There are also some cases in which *rule-json-cls-name-as-entityType* is ignored or conditional.

- To prevent the addition of unnecessary JSON members (here: because the JSON member would already be inherited), the rule is ignored for a type T if T is a subtype and *rule-json-cls-name-as-entityType* already applies to one of its supertypes.
- By default, the rule does not apply to unions, enumerations, and code lists.

  However, if *rule-json-cls-name-as-entityType-union* is enabled together with *rule-json-cls-name-*

as-entityType, then the latter also applies to unions. The considerations that led to the addition of the former conversion rule are: Unions can be converted to JSON objects (see Union, more specifically: Property Choice). The ontology target of ShapeChange [https://shapechange.net/targets/ontology/uml-rdfowl-based-isois-19150-2/] encodes a union as a class, with cardinality restrictions to ensure that only one option (defined by the union) is used. For further details, also see the OGC Testbed-12 ShapeChange Engineering Report [http://docs.opengeospatial.org/per/16-020.html#rdf_cr_class_union]. This is an argument for applying *rule-json-cls-name-as-entityType* to unions, because it would support a JSON-LD based mapping to the union class in RDF/OWL.

6.2.3.3. Abstractness

JSON Schema does not directly support abstractness. An abstract class is therefore encoded like a non-abstract class.

NOTE	Encoding a JSON object that represents an abstract type, with the "entityType" having the abstract type name as value, would be useful with regards to linked data applications, and conversion of JSON data to RDF using JSON-LD. Abstractness is also not supported in RDF/OWL, so RDF resources can define the RDFS/OWL class or datatype, which represent an abstract type from the conceptual model, as their type. That makes sense for cases in which the exact type of a resource or "thing" is not known yet, but a more general type is.

6.2.3.4. Inheritance

JSON Schema does not support the concept of inheritance itself. A workaround for this issue would be to transform the conceptual model and flatten all inheritance hierarchies. For further details, see Flattening Inheritance.

The following sections document the conversion of an inheritance relationship, covering the topics of Class Generalization and Property Inheritance and Class Specialization and Property Ranges. A special case of generalization, for classes with specific stereotypes, is discussed in section Virtual Generalization.

6.2.3.4.1. Class Generalization and Property Inheritance

The generalization relationship of a subtype to its supertype is converted by combining the structural constraints of the subtype and its supertype using the JSON Schema keyword "allOf."

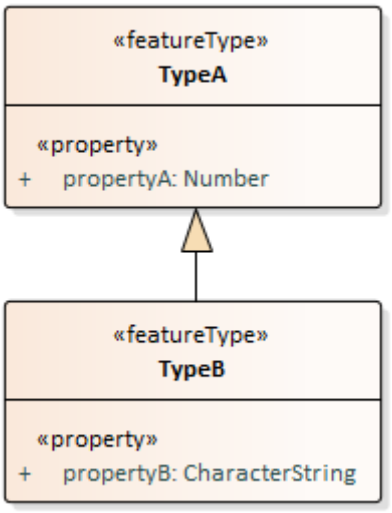

Figure 4. Example of type inheritance

Listing 12. JSON Schema example for realizing generalization using "allOf"

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "definitions": {
    "TypeA": {
      "properties": {
        "propertyA": {
          "type": "number"
        }
      },
      "required": [
        "propertyA"
      ]
    },
    "TypeB": {
      "allOf": [
        {
          "$ref": "#/definitions/TypeA"
        },
        {
          "type": "object",
          "properties": {
            "propertyB": {
              "type": "string"
            }
          },
          "required": [
            "propertyB"
          ]
        }
      ]
    }
  },
  "$ref": "#/definitions/TypeB"
}
```

This JSON object is valid against the schema from Listing 12:

```
{
  "propertyA": 2,
  "propertyB": "x"
}
```

This JSON object is invalid (because "propertyA" is missing) against the schema from Listing 12:

```
{
  "propertyB": "x"
}
```

NOTE	This also works for an encoding where the properties of a class are nested within a key-value pair (like "properties" for a GeoJSON encoding).
NOTE	The case where a property from a supertype is redefined by a property from the subtype is supported. Redefinition in UML requires that the value type of the subtype property is "kind of" the type of the redefined property of the supertype. Therefore, the property value, when encoded in JSON, would satisfy the JSON Schema constraints defined by both the subtype property and the redefined supertype property.

This approach to converting a generalization relationship has the following restrictions.

- The JSON Schema keyword "additionalProperties" must not be set to false in the definitions of both the super- and the subtype.
- The approach only works for generalization relationships of feature, object, and data types. For unions, enumerations, and code lists generalization relationships are not supported.
- It only converts the generalization relationship from subtype to supertype. It does not support the other direction of an inheritance relationship, i.e., specialization. Given a JSON object that encodes a subtype, and the JSON Schema of the supertype, then only the constraints of the supertype are checked, but not all the constraints that apply to the subtype. That is an issue when encoding a UML property whose value type is or could be a supertype (via a subtype that is added by an external, so far unknown schema). Conceptually, the actual value of that property can be a supertype object, but it could just as well be an object whose type is a subtype of that supertype. This issue can only be solved to a certain degree with JSON Schema, as explained in the Class Specialization and Property Ranges section.

Multiple inheritance is supported by adding all supertypes as elements of "allOf."

6.2.3.4.2. Virtual Generalization

It is often useful to encode all classes with a certain stereotype with a common base type. The generalization relationship to such a base type is often implied with the stereotype, for a given encoding. In GML, for example, the common base type for classes with stereotype <<featureType>> is gml:AbstractFeature. Rather than explicitly modeling such a base type (e.g., *AnyFeature* defined by ISO 19109), as well as explicitly modeling generalization relationships to the base type, the encoding rule typically takes care of adding that relationship to relevant schema types.

This kind of virtual generalization is supported via *rule-json-cls-virtualGeneralization*. The rule adds generalization relationships to specific kinds of classes - if a) according ShapeChange JSON Schema target parameters have been set, and b) the class does not already have that generalization relationship via one of its supertypes:

- feature type - configuration parameter *baseJsonSchemaDefinitionForFeatureTypes*

- object type - configuration parameter *baseJsonSchemaDefinitionForObjectTypes*
- data type - configuration parameter *baseJsonSchemaDefinitionForDataTypes*

The parameter value shall be a URI to reference the JSON Schema that defines the common base type. For example, in order for all feature types to use the GeoJSON Feature definition as common base, set `baseJsonSchemaDefinitionForFeatureTypes = https://geojson.org/schema/Feature.json`.

NOTE	Being able to choose any URI as parameter value can be useful for offline and private-network situations.

NOTE	The parameters do not have a default value. If a parameter is not set or does not have a value, then *rule-json-cls-virtualGeneralization* will not have an effect for the kind of class (feature, object, or data type) for which the parameter applies.

The virtual generalization relationship is implemented by converting the class to a JSON Schema that consists of an "allOf" with two subschemas: the first being a "$ref" with the URI defined by the target parameter, the second being the schema produced by applying the other conversion rules to the class.

The only exception is *rule-json-cls-name-as-anchor*, because the "$anchor" created by that rule is not encoded in the second subschema, but in the schema that contains the "allOf".

6.2.3.4.3. Class Specialization and Property Ranges

By default, validation of a property value encoded in JSON, with the value having a complex type (i.e., being a JSON object or an array of JSON objects), only encompasses checking the JSON Schema constraints defined for that type. If the property value actually is a subtype of that type, then the constraints defined for that subtype would not be checked. Ideally, the constraints of known and unknown subtypes would automatically be checked, but JSON Schema does not support this.

NOTE	If JSON data was transformed to RDF using JSON-LD, then class specialization and property ranges could fully be checked by validating the RDF data using SHACL. For further details, see the SHACL Conversion Rules section of this document.

To a limited extent specialization relationships could be represented in JSON Schema, but only for subtypes in the same schema/model and using complex, verbose constructs from JSON Schema. This would make the schemas hard to read (by humans) and to parse (by application schema parsers in clients). The value of such a capability is therefore questionable and currently not supported.

NOTE	To capture the discussion, a potential conversion rule is documented below, but has not been implemented in the pilot. The conversion rule would support known subtypes. "Known" are the subtypes defined in the UML model that contains the application schema from which the JSON Schema is derived. There would be a limitation that subtypes from different schemas with identical

class names, but conflicting definitions (e.g., `SchemaA::ExtensionX.propA` has type `string`, while `SchemaB::ExtensionX.propA` has type `number`) would not be supported.

A pre-condition of the conversion rule would be that the subtype name must be included in the encoding of the JSON object. The conversion rule therefore would require *rule-cls-name-as-entityType* (see section Type Identification) to be part of the encoding rule with the default name "entityType" - or some alternative mechanism to identify the JSON member that encodes the type.

The definitions schema includes a definition for each type that is being converted. Under the additional conversion rule for specialization, a second definition would be generated for each supertype of the application schema (that is a feature, object, or data type). The name of that definition would be constructed as: `{type-name}_valueType`. If *rule-cls-name-as-anchor* (see Location Independent Schema Identifiers) is also part of the encoding rule, then that name would also be encoded as "$anchor" of the new definition.

The JSON Schema of the new definition would be constructed as follows.

- Determine the list of direct and indirect subtypes. If abstract types are also converted (see Abstractness), then abstract subtypes would be included, too.
- Create a sequence of if-then-else expressions, where the if-condition checks the "entityType" against the name of a subtype, the "then" case refers to the JSON Schema definition of that type, and the "else" case represents the if-then-else for the next subtype. The last "else" case refers to the JSON Schema definition of the supertype. Listing 13 illustrates how this would look like, for the UML model from Figure 5.

The resulting JSON Schema looks for an "entityType" match. If one is found, the JSON object is validated against the JSON Schema definition of that type. If no match is found, then the JSON Schema of the supertype is used as a fallback for validation. The JSON object would then at least have to fulfill the constraints defined by the schema of the supertype. However, any additional content of the JSON object would not be validated.

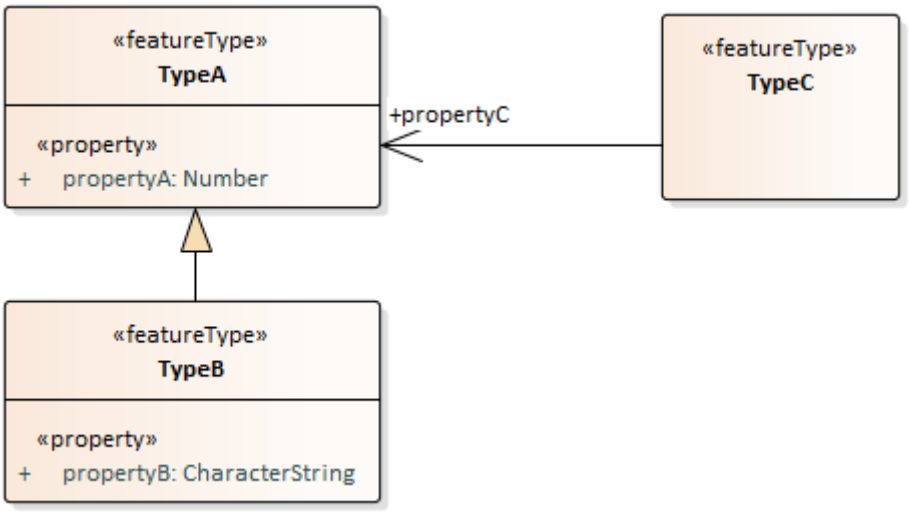

Figure 5. Example of a value type being a supertype

Listing 13. JSON Schema example of a value type definition for a supertype

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "definitions": {
    "TypeA": {
      "properties": {
        "entityType": {
          "type": "string"
        },
        "propertyA": {
          "type": "number"
        }
      },
      "required": [
        "entityType",
        "propertyA"
      ]
    },
    "TypeA_valueType": {
      "if": {
        "properties": {
          "entityType": {
            "const": "TypeB"
          }
        }
      },
      "then": {
        "$ref": "#/definitions/TypeB"
      },
      "else": {
        "$ref": "#/definitions/TypeA"
      }
    },
    "TypeB": {
```

```
      "allOf": [
        {
          "$ref": "#/definitions/TypeA"
        },
        {
          "type": "object",
          "properties": {
            "entityType": {
              "type": "string"
            },
            "propertyB": {
              "type": "string"
            }
          },
          "required": [
            "entityType",
            "propertyB"
          ]
        }
      ]
    },
    "TypeC": {
      "properties": {
        "entityType": {
          "type": "string"
        },
        "propertyC": {
          "$ref": "#/definitions/TypeA_valueType"
        }
      },
      "required": [
        "entityType",
        "propertyC"
      ]
    }
  },
  "$ref": "#/definitions/TypeC"
}
```

The following two JSON objects are valid against the schema from Listing 13:

```
{
  "entityType": "TypeC",
  "propertyC": {
    "entityType": "TypeA",
    "propertyA": 3
  }
}
```

```
{
  "entityType": "TypeC",
  "propertyC": {
    "entityType": "TypeB",
    "propertyA": 3,
    "propertyB": "x"
  }
}
```

The next JSON object is also valid against the schema from Listing 13, because the value of "propertyC" matches the schema of supertype TypeA, even though the entity type is unknown:

```
{
  "entityType": "TypeC",
  "propertyC": {
    "entityType": "UnknownExtensionTypeA",
    "propertyA": 3
  }
}
```

The following two JSON objects are invalid against the schema from Listing 13 (in both cases because the type of "propertyA" is string instead of number):

```
{
  "entityType": "TypeC",
  "propertyC": {
    "entityType": "TypeB",
    "propertyA": "y",
    "propertyB": "x"
  }
}
```

```
{
  "entityType": "TypeC",
  "propertyC": {
    "entityType": "UnknownExtensionTypeB",
    "propertyA": "z"
  }
}
```

Note that if the last else-case simply was 'false', then that would prevent any unknown subtype from being encoded - at least with the value type definition created and used within the definitions schema that is being produced. For a non-abstract supertype, the if-then-else construct would then have to contain an

additional `if (entityType=Supertype) then $ref: schemaSupertype` - before the case of "else false."

6.2.3.5. Feature and Object Type

In the conceptual model, feature and object types represent objects that have identity. That differentiates these types from, for example, data types. Other than that, feature and object types are encoded as JSON objects, just like a data type.

A feature or object type - in the following summarily called types with identity - is converted to a JSON Schema definition which is added to the definitions schema, using the type name as definition key. Note that ISO 19109 requires class names to be unique within the scope of an application schema.

The conversion of the class properties is defined in the Properties section. General type conversion rules, such as those documented in the Class Name section, may apply. Additional conversion rules and behavior for types with identity are described in the following sections.

6.2.3.5.1. Identifier

The conceptual model of a type with identity often does not contain a property whose value is used by applications to identify objects of that type. Instead, the according information is added or defined in platform specific encodings. For example, a GML application schema offers the gml:id attribute as well as the gml:identifier element to encode identifying information.

In a web publishing context, the URI at which a JSON object is published can be used as its identifier. Therefore, the default behavior of the ShapeChange JSON Schema target is to not add an identifier property. However, in many applications it is useful to have a member within a JSON object that provides the identifier of that object. Existing specifications often include such a capability, e.g., the "id" member in GeoJSON or "@id" in JSON-LD.

With *rule-json-cls-identifierForTypeWithIdentity*, an identifier JSON member will be added to the JSON object that represents the type with identity. The key, and value type of that member can be configured using ShapeChange JSON Schema target parameters:

- *objectIdentifierName*: "id" (the default) or any other suitable string that does not conflict with other member names);
- *objectIdentifierType*: "string" (the default), "number", or "string, number";
- *objectIdentifierRequired*: "false" (the default) or "true" is used to define if the property is required or optional.

Listing 14. JSON Schema example of a feature type with a required identifier property "id"

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "definitions": {
    "TypeA": {
      "properties": {
        "entityType": {
          "type": "string"
        },
        "id": {
          "type": "string"
        },
        "propertyA": {
          "type": "number"
        }
      },
      "required": [
        "entityType",
        "id",
        "propertyA"
      ]
    }
  },
  "$ref": "#/definitions/TypeA"
}
```

The following JSON object is valid against the schema from Listing 14:

```
{
  "entityType": "TypeA",
  "id": "42445fdasd7asd6f7",
  "propertyA": 3
}
```

rule-json-cls-identifierForTypeWithIdentity is ignored if one of the following conversion rules is part of the encoding rule as well:

- *rule-json-cls-identifierStereotype* - This conversion rule assumes that all types with identity have an attribute with stereotype <<identifier>> (directly, or inherited from a supertype). That attribute is used to encode the identifier.

  NOTE If the maximum multiplicity of an <<identifier>> attribute is greater than 1, ShapeChange will log an error.

- *rule-json-cls-ignoreIdentifier* - With this rule, the identifier of a type with identity will be encoded using an identifier member that is provided by a common base type (e.g., the "id" member of a GeoJSON Feature, to which a generalization relationship exists for a given feature

type - see Virtual Generalization). That means that no additional identifier property is created. *rule-json-cls-identifierForTypeWithIdentity* is therefore overridden by *rule-json-cls-ignoreIdentifier*. Also, all identifier properties that are identified by *rule-json-cls-identifierStereotype* - if also included in the encoding rule - will simply be ignored when encoding the type with identity.

NOTE The JSON Schema for a GeoJSON Feature [https://geojson.org/schema/Feature.json] does not include "id" (which is defined in the GeoJSON standard, section 3.2 [https://tools.ietf.org/html/rfc7946#section-3.2]) - not even as optional property of a "Feature". A PullRequest has been created which would fix this, see https://github.com/geojson/schema/pull/9, but it has not been merged (as of March 11, 2020).

To prevent the addition of unnecessary JSON members (here: because the JSON member would already be inherited), *rule-json-cls-identifierForTypeWithIdentity* is ignored for a type T if T is a subtype and *rule-json-cls-identifierForTypeWithIdentity* already applies to one of its supertypes.

6.2.3.5.2. Nested Properties

By default, the properties of a type are converted to first-level properties of the resulting JSON object. In GeoJSON, feature properties are encoded within the GeoJSON "properties" member. Notable exceptions from that rule are the GeoJSON members "id," "geometry," and "bbox." In order to produce a JSON Schema that converts the properties of a type with identity to be encoded within a nested "properties" member - minus any properties that are mapped to the other aforementioned GeoJSON keys - the conversion rule *rule-json-cls-nestedProperties* needs to be included in the encoding rule.

NOTE Listing 26 illustrates the result of applying *rule-json-cls-nestedProperties*, given the feature type in Figure 15.

6.2.3.6. Data Type

A <<dataType>> is converted to the JSON Schema definition of a JSON object. The properties of the data type are converted to the properties of that object, as described in the Properties section.

6.2.3.7. Mixin Type

ShapeChange supports the notion of mixin type (for further details, see http://shapechange.net/targets/xsd/extensions/mixin/). They are primarily used by the XML Schema target. However, if that target is contained in the ShapeChange configuration, it has implications on how UML types are loaded. In this case, it may lead to UML types being loaded as mixin types. A UML type is loaded as a mixin type if:

- *rule-xsd-cls-mixin-classes* is contained in the XSD encoding rule and:
  - the tagged value *gmlMixin* is set to true, or
  - The type has the stereotype <<type>>, is abstract, and the tagged value *gmlMixin* is not set to false.

For the JSON Schema conversion rules, a mixin type is treated like a data type.

6.2.3.8. Union

Application schemas have two ways of using types with stereotype <<union>>.

- According to ISO 19103:2015, a <<union>> type consists *"of one and only one of several alternative datatypes (listed as member attributes). This is similar to a discriminated union in many programming languages"*. According to this definition, only the types of the UML attributes defined for a <<union>> are of interest.
- In practice, unions defined in application schemas can also have another use: they define a choice between a number of options, where each option is defined by a UML attribute. In other words, the attribute itself has meaning (not just its value type). Multiple options can have the same value type. The UML-to-GML application schema encoding rules support this way of using unions (see OGC 07-036r1, section E.2.4.10).

The following sections document the conversion rules that support these two approaches of using unions.

6.2.3.8.1. Property Choice

rule-json-cls-union-propertyCount encodes a choice between the properties - i.e., the options - defined by a <<union>>.

The <<union>> is converted to the JSON Schema definition of a JSON object. Each union option is represented as an optional member of the JSON object. The choice between the options defined by the union is encoded using "maxProperties" = "minProperties" = 1. That is, the number of members that are allowed for the JSON object is restricted to exactly one.

An `"additionalProperties": false` is used to prevent any undefined properties. The result of applying these rules to the union from Figure 6 is shown in Listing 15.

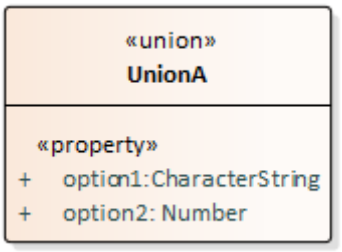

Figure 6. <<union>> example

Listing 15. Example of a JSON Schema for a <<union>> class, representing the property choice using "minProperties" and "maxProperites"

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "definitions": {
    "UnionA": {
      "type": "object",
      "properties": {
        "option1": {
          "type": "string"
        },
        "option2": {
          "type": "number"
        }
      },
      "additionalProperties": false,
      "minProperties": 1,
      "maxProperties": 1
    }
  },
  "$ref": "#/definitions/UnionA"
}
```

NOTE	An alternative approach would be using the "oneOf" keyword, with one subschema per union property, which only defines that property, and requires it (but does not perform any other checks). This option is more verbose, harder to read and understand and, therefore, not implemented.

This JSON object is valid against the schema from Listing 15:

```
{
  "option1": "x"
}
```

This JSON object is invalid (because "option2" has a string value, rather than a numeric value) against the schema from Listing 15:

```
{
  "option2": "x"
}
```

6.2.3.8.2. Type Discriminator

rule-json-cls-union-typeDiscriminator encodes a type discriminator defined by a <<union>>.

The <<union>> is converted to a JSON Schema definition that represents a choice between the value

types of the union properties.

- If the value types are only simple, without a specific format definition or other restrictions defined by JSON Schema keywords, then the JSON Schema will only contain a "type" member, with an array of the simple types.
- Otherwise, a "oneOf" member is added to the JSON Schema definition, with:
  - one "$ref" per non-simple type,
  - one "type" for all simple types without specific keywords, and
  - one "type" per simple type with specific keywords.

The result of applying the rule to the union from Figure 7 is shown in Listing 15.

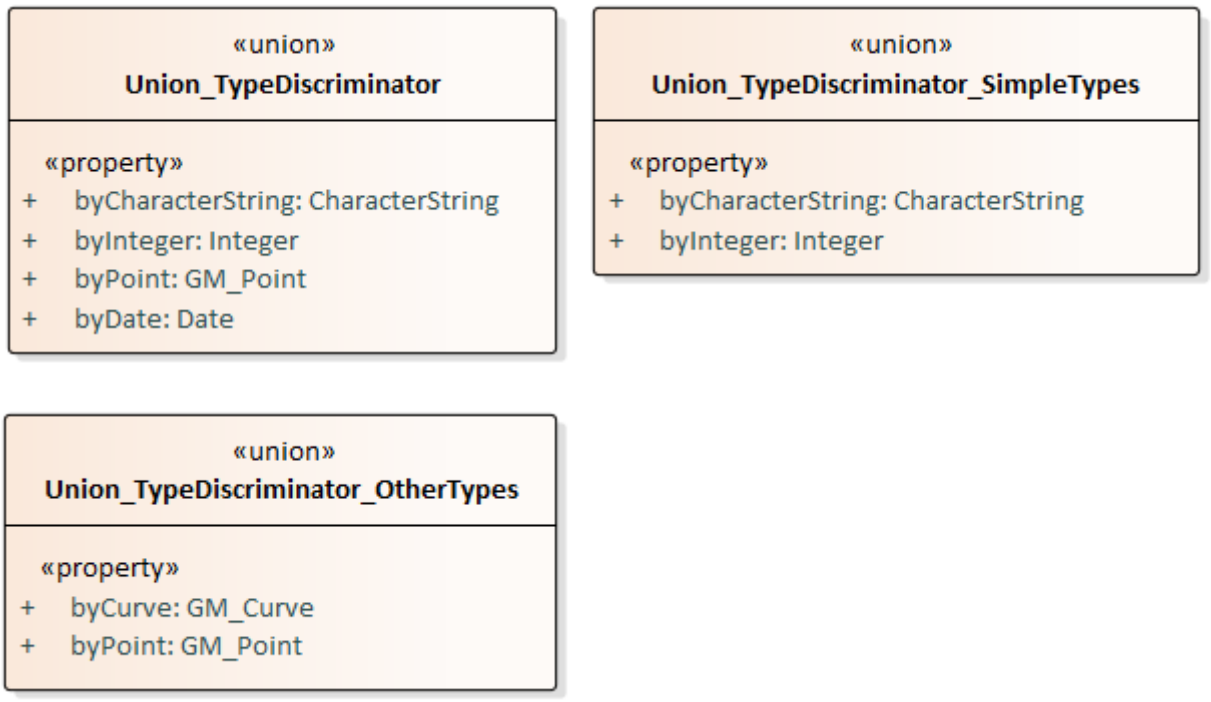

Figure 7. Example of type discriminator unions

Listing 16. Example of a JSON Schema for unions, encoding them as type discriminators

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "definitions": {
    "Union_TypeDiscriminator": {
      "oneOf": [
        {
          "type": [
            "string",
            "integer"
          ]
        },
        {
          "$ref": "https://geojson.org/schema/Point.json"
        },
        {
          "type": "string",
          "format": "date"
        }
      ]
    },
    "Union_TypeDiscriminator_OtherTypes": {
      "oneOf": [
        {
          "$ref": "https://geojson.org/schema/LineString.json"
        },
        {
          "$ref": "https://geojson.org/schema/Point.json"
        }
      ]
    },
    "Union_TypeDiscriminator_SimpleTypes": {
      "type": [
        "string",
        "integer"
      ]
    }
  }
}
```

6.2.3.9. Enumeration

An <<enumeration>> is converted to a JSON Schema definition with a type defined by evaluating tagged value *literalEncodingType*. In addition, it uses the "enum" keyword to restrict the value to one of the enums from the enumeration.

The tagged value *literalEncodingType* identifies the conceptual type that applies to the enumeration values. If the tagged value is not set on the enumeration, or has an empty value, then the literal encoding type is set to be CharacterString. The literal encoding type is mapped to a JSON Schema

type. The result should be a simple JSON Schema type (string, number, integer, or boolean). The enumeration values will be encoded accordingly.

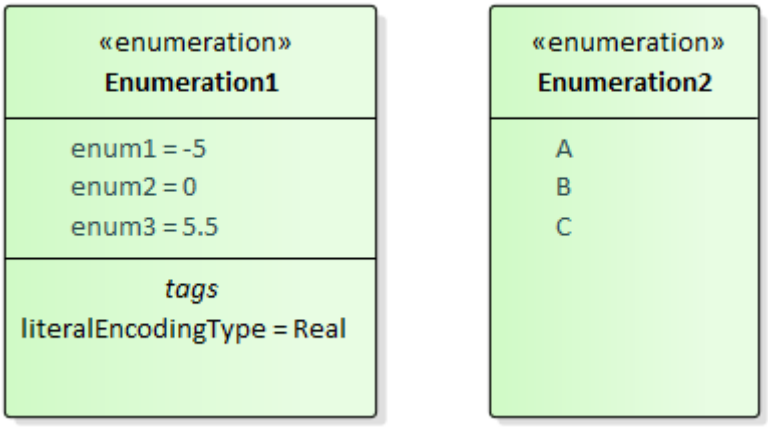

Figure 8. <<enumeration>> example

Listing 17. Example of enumerations encoded in JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "definitions": {
    "Enumeration1": {
      "type": "number",
      "enum": [-5, 0, 5.5]
    },
    "Enumeration2": {
      "type": "string",
      "enum": ["A","B","C"]
    }
  },
  "anyOf": [
    {"$ref": "#/definitions/Enumeration1"},
    {"$ref": "#/definitions/Enumeration2"}
  ]
}
```

6.2.3.10. Code List

By default, a <<codelist>> is converted to a JSON Schema definition with a type defined by evaluating tagged value *literalEncodingType*.

The tagged value *literalEncodingType* identifies the conceptual type that applies to the code values. If the tagged value is not set on the code list, or has an empty value, then the literal encoding type is set to be CharacterString. The literal encoding type is mapped to a JSON Schema type. The result should be a simple JSON Schema type (string, number, integer, or boolean).

With *rule-json-cls-codelist-uri-format*, all code lists will be represented by a JSON Schema that restricts the type to "string", and states that the "format" is "uri" (as defined by JSON Schema validation, section 7.3.5 [https://tools.ietf.org/html/draft-handrews-json-schema-validation-02#section-7.3.5]).

With *rule-json-cls-codelist-link*, all code lists will be represented by a JSON Schema that restricts the type to a "Link" object [http://schemas.opengis.net/ogcapi/features/part1/1.0/openapi/schemas/link.yaml] as specified by IETF RFC 8288 and implemented in the OGC API standards. The Link object provides "href" and "title" members like the simple Xlinks in GML.

NOTE	The URL to the JSON Schema of the "Link" object (as shown in Listing 18) can be configured using the ShapeChange JSON Schema target parameter *linkObjectUri*.

Listing 18. JSON Schema of the "Link" object

```
{
 "$schema": "http://json-schema.org/draft-07/schema#",
 "type": "object",
 "required": ["href","rel"],
 "properties": {
  "href": {
   "type": "string",
   "examples": ["http://data.example.com/buildings/123"]
  },
  "rel": {
   "type": "string",
   "examples": ["alternate"]
  },
  "type": {
   "type": "string",
   "examples": ["application/geo+json"]
  },
  "hreflang": {
   "type": "string",
   "examples": ["en"]
  },
  "title": {
   "type": "string",
   "examples": ["Trierer Strasse 70, 53115 Bonn"]
  },
  "length": {
    "type": "integer"
  }
 }
}
```

In Figure 9, the encoding rule for code list "CodelistLinkObject" contains *rule-json-cls-codelist-link*, whereas the encoding rule for code list "CodelistUriFormat" contains *rule-json-cls-codelist-uri-format*, and the encoding rule for the remaining two code lists contains none of these conversion rules. The resulting JSON Schema is shown in Listing 19.

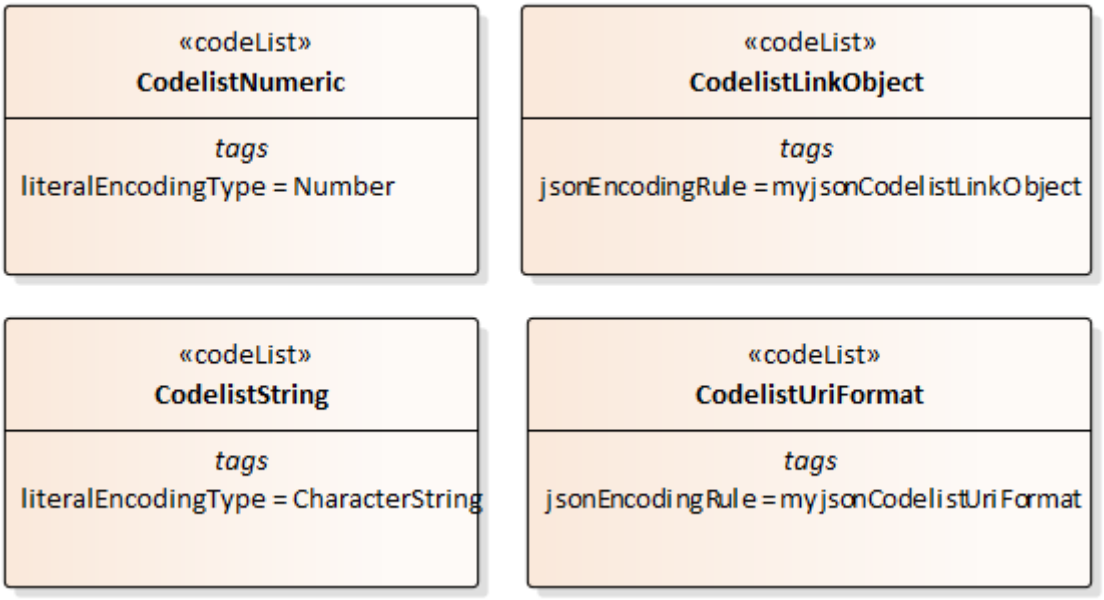

Figure 9. <<codeList>> example

Listing 19. Example of code lists encoded in JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "definitions": {
    "CodelistLinkObject": {
      "$ref": "http://example.org/jsonschema/link.json"
    },
    "CodelistNumeric": {
      "type": "number"
    },
    "CodelistString": {
      "type": "string"
    },
    "CodelistUriFormat": {
      "type": "string",
      "format": "uri"
    }
  }
}
```

NOTE A string based code list conversion is better suited for a subsequent mapping of JSON encoded code values to RDF using JSON-LD. For further details, see the future work item Conversion of JSON data to RDF using JSON-LD.

6.2.3.11. Basic Type

If a direct or indirect supertype of an application schema class is mapped to one of the simple JSON Schema types *string*, *number*, *integer*, or *boolean*, then under *rule-json-cls-basictype* that class represents a so called *basic type*.

A basic type does not define a JSON object. It represents a simple data value, e.g., a string. The JSON Schema definition of a basic type thus defines a simple JSON Schema type. A basic type can be restricted using a number of JSON Schema keywords. Table 4 defines which tagged values can be used to define these restrictions for a basic type, and which restrictions are available for which simple JSON Schema type.

Table 4. Basic type restrictions

JSON Schema keyword	**tagged value to define the restriction**	**applicable JSON Schema type(s)**
format	*jsonFormat*	string, number, integer
maxLength	*length*, *maxLength*, or *size*	string
pattern	*jsonPattern*	string
minimum (inclusive)	*rangeMinimum*	number, integer
maximum (inclusive)	*rangeMaximum*	number, integer

NOTE The JSON Schema keyword "format" is defined in chapter 7 of JSON Schema Validation: A Vocabulary for Structural Validation of JSON. The formats defined there (e.g., "date-time", "uri", and "json-pointer") apply to JSON values of type string. Custom formats could apply to JSON values of type number and integer.

NOTE JSON Schema Validation: A Vocabulary for Structural Validation of JSON defines the JSON Schema keyword "pattern". According to that specification, the value of the keyword should be a regular expression according to the ECMA 262 regular expression dialect. However, the specification does not reference a particular version or edition of ECMA 262. The regular expression dialect to be used in a JSON Schema "pattern" therefore is not exactly defined, and consequently depends on the implementation of a JSON Schema validator. In order to avoid issues with this diversity, JSON Schema: A Media Type for Describing JSON Documents defines a number of recommendations for writing regular expressions in JSON Schema.

NOTE

The regular expression dialect used by JSON Schema is the one used by ECMA 262 (though, as said before, no specific version or edition of that standard is referenced by JSON Schema). It is unlikely that this dialect will ever be the same as the one used by XML Schema 1.1 [https://www.w3.org/TR/xmlschema11-2/#regexs], which is for example used in XML Schema to define the pattern facet [https://www.w3.org/TR/xmlschema11-2/#rf-pattern]. XML Schema conversion rules supported by ShapeChange (rule-xsd-prop-constrainingFacets [https://shapechange.net/targets/xsd/extensions/#rule-xsd-prop-constrainingFacets] and rule-xsd-prop-length-size-pattern [https://shapechange.net/targets/xsd/extensions/#rule-xsd-prop-length-size-pattern]) use tagged value *pattern* to define the regular expression for the pattern facet in the XML Schema encoding. Due to the differences in regular expression dialects used by JSON Schema and XML Schema, *rule-json-cls-basictype* uses a different tagged value, namely *jsonPattern*, to define the regular expression for the "pattern" keyword in a JSON Schema. If the regular expression for a basic type must be different, in order to be valid in XML Schema and in JSON Schema, then both tagged value *pattern* and *jsonPattern* must be set.

However, for cases in which the regular expressions used to constrain string values within the application schema are known to be valid in the regular expression dialects of both JSON Schema and XML Schema, ShapeChange offers a way to define the regular expression only once per application schema element, and still derive XML Schema and JSON Schema from the conceptual model. Tag aliases [https://shapechange.net/get-started/config/input/#Tag_aliases] can be used in the ShapeChange configuration, to map the name of a tagged value to a different name. Tag *pattern* could thus be mapped to tag *jsonPattern*. However, tag aliases do not create copies of tagged values with different name. Tag aliasing results in renaming tags. Therefore, in order to derive both XML Schema and JSON Schema from an application schema, ShapeChange would have to be executed twice: once without mapping tag *pattern*, to create the XML Schema, and once with mapping tag *pattern* to tag *jsonPattern*, and then deriving JSON Schema.

NOTE

If the "format" keyword is used to restrict the structure of a JSON string, so that it matches a certain regular expression, then it is useful to add the "pattern" keyword as well, explicitly defining that regular expression (given that the regular expression follows an ECMA 262 regular expression dialect). The reason is that the "format" is first and foremost an annotation, so can be ignored by JSON Schema validators, whereas the "pattern" keyword will be evaluated by a JSON Schema validator. JSON Schema validators may treat the "format" keyword like an assertion, but that is not guaranteed. In any case, the "format" keyword helps to convey more information about the specific type of a JSON value (e.g., "date" instead of just "string"), and thus should not be omitted if a certain, well-known (i.e., defined by a JSON Schema vocabulary) format is applicable to a JSON value.

There are a number of cases which need to be considered when encoding a basic type.

- The basic type directly inherits from a type that is mapped to one of the simple JSON Schema types listed above: in that case, the JSON Schema definition of the basic type will include the "type" keyword with appropriate value, and potentially also the "format" keyword if the

mapping defines a specific format. In addition, restrictions defined for the basic type via tagged values are encoded using the appropriate JSON Schema keywords (as defined in Table 4).

- Otherwise, i.e., the basic type does not directly inherit from a type that is mapped to a simple JSON Schema type:
  - If no restrictions are defined for the basic type, then the JSON Schema definition of the basic type simply contains a "$ref" to the JSON Schema definition of the direct supertype.
  - Otherwise, i.e., restricitons are defined, an "allOf" is used to refer to the JSON Schema definition of the direct supertype, and to define a JSON Schema with the restrictions that apply to the basic type.

Figure 10 provides a detailed example that illustrates these cases. The JSON Schema encoding is shown in Listing 20.

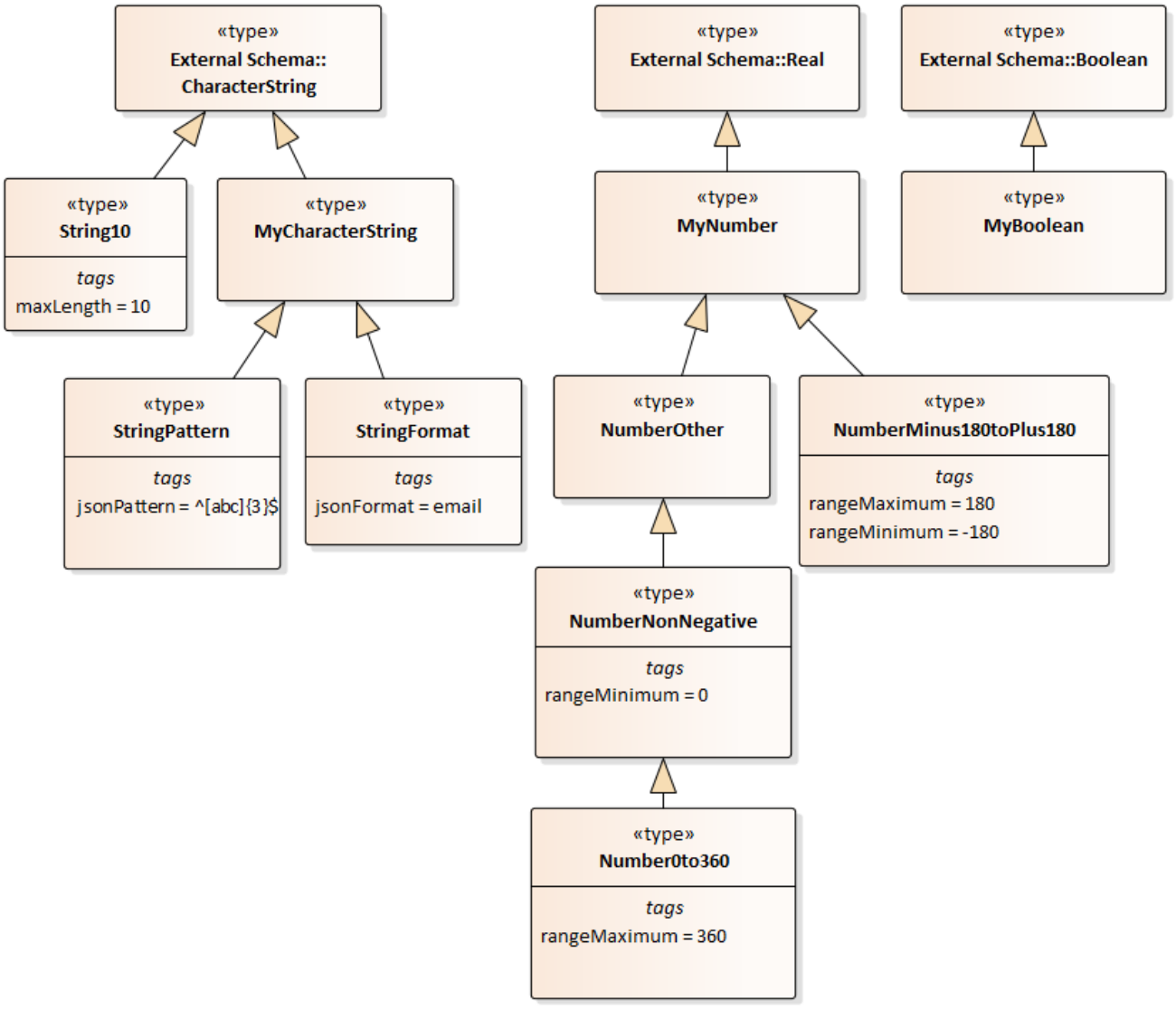

Figure 10. Basic types example

Listing 20. Example of basic types encoded in JSON Schema

```
1 {
2   "$schema": "http://json-schema.org/draft-07/schema#",
3   "definitions": {
```

```
    "MyBoolean": {
      "type": "boolean"
    },
    "MyCharacterString": {
      "type": "string"
    },
    "MyNumber": {
      "type": "number"
    },
    "Number0to360": {
      "allOf": [
        {
          "$ref": "#/definitions/NumberNonNegative"
        },
        {
          "maximum": 360.0
        }
      ]
    },
    "NumberMinus180toPlus180": {
      "allOf": [
        {
          "$ref": "#/definitions/MyNumber"
        },
        {
          "minimum": -180.0,
          "maximum": 180.0
        }
      ]
    },
    "NumberNonNegative": {
      "allOf": [
        {
          "$ref": "#/definitions/NumberOther"
        },
        {
          "minimum": 0.0
        }
      ]
    },
    "NumberOther": {
      "$ref": "#/definitions/MyNumber"
    },
    "String10": {
      "allOf": [
        {
          "type": "string"
        },
        {
          "maxLength": 10
        }
```

```
      ]
    },
    "StringFormat": {
      "allOf": [
        {
          "$ref": "#/definitions/MyCharacterString"
        },
        {
          "format": "email"
        }
      ]
    },
    "StringPattern": {
      "allOf": [
        {
          "$ref": "#/definitions/MyCharacterString"
        },
        {
          "pattern": "^[abc]{3}$"
        }
      ]
    }
  }
}
```

6.2.3.12. Default Geometry

By default, any property with a geometry type is converted as any other property, with the geometry type being mapped to a JSON Schema as defined via map entries (see section Mappings).

This may not always be desired. A GeoJSON Feature, for example, has a dedicated (and required) member - "geometry" - for encoding the feature geometry. Geometry properties defined within an application schema may therefore need to be mapped to this "geometry" member. Additional conversion behavior is required to define and achieve this mapping.

Two kinds of application schemas need to be distinguished:

1. application schemas where a class has at most one geometry property, typically a property with one of the ISO 19107 geometry types as value type; and
2. application schemas where classes can have more than one geometry property.

NOTE	When counting geometry properties per class, inheritance also needs to be considered. A class that, through inheritance, has multiple geometry typed properties with different name belongs to an application schema of the second category, whereas a class that only has at most one geometry typed property - also through inheritance - belongs to the first category.

Two conversion rules are available to support the two kinds of application schemas.

- *rule-json-cls-defaultGeometry-singleGeometryProperty* - for application schemas with classes that have at most one geometry property. With this rule, the geometry property of a class represents the default geometry, and is encoded as the top-level "geometry" member. If a class has multiple - potentially inherited - geometry properties with different names, none of them is selected as default geometry (because no informed choice can be made) and ShapeChange will log an error.
- *rule-json-cls-defaultGeometry-multipleGeometryProperties* - for application schemas with classes that can have multiple geometry properties. With this rule, a geometry property is identified as default geometry by setting tagged value *defaultGeometry* on the property to the value `true`. That property will then be encoded as a top-level "geometry" member. If multiple such properties exist (potentially inherited), none of them is selected as default geometry (because no informed choice can be made) and an error will be logged.

NOTE For a class that has multiple geometry properties with different names, all such properties except the one identified as default geometry are encoded just as any other property of the class. A different behavior would be to omit these other geometry properties altogether when deriving the JSON Schema. It is unclear at this point if that would really be useful behavior. After all, the modeling expert that designed the application schema with classes that have multiple geometry properties must have had good reasons for doing so, and must also be aware that in order to perform spatial computations on the different geometry properties, special software will be required.

If a UML property is identified as default geometry, then it is implemented via the top-level "geometry" member (and not as another member of the JSON object). The "geometry" member is constrained to the JSON Schema definition to which the value type of the default geometry property is mapped.

NOTE The schema should only use geometry types that are mapped in the configuration of the ShapeChange process. For example, in the case of a GeoJSON encoding rule, all geometry types should be mapped to GeoJSON geometry types. Otherwise the JSON Schema constraints of the GeoJSON Feature schema could not be satisfied.

NOTE One might think that it is beneficial to set the "geometry" member to just `null` if the type that is being converted does not have a default geometry. However, that must not be done in any schema conversion, because then a default geometry defined by a super- or (maybe defined in an external schema) subtype of that type could never be mapped to the "geometry" member. The only exception would be classes marked as "final," where none of the supertypes define a default geometry.

NOTE If the default geometry property has a maximum multiplicity greater than 1, then ShapeChange will log a warning and assume a maximum multiplicity of exactly 1.

NOTE Listing 26 illustrates the result of applying *rule-json-cls-defaultGeometry-singleGeometryProperty*, given the feature type in Figure 15.

NOTE Additional geometry properties are encoded like all other UML properties (in the GeoJSON case within the GeoJSON "properties" member).

NOTE In some application schemas, the geometry of a feature type is not defined directly, i.e., not via a property that has an ISO 19107 type as value type. For example, consider Figure 14, where property "place" indirectly defines the geometry of a feature type, through a complex *PlaceSpecification*. The place is either given by a point, a curve, a surface, or by some location identifier. Such a feature model, where the geometry of a feature can be one of several geometry and non-geometry types, is not suited for the conversion rules documented in this section.

6.2.4. Properties

A UML property of a class is converted to a member of a JSON object - unless the encoding rule defines a different behavior for the type that owns the property (e.g., for enumerations and code lists).

The default result of converting a UML property, therefore, is a key within the "properties" key of the JSON Schema definition for the type that owns the property, with the key name being the name of the UML property, and the value being a JSON Schema with constraints and annotations that define the property (value type, multiplicity, etc).

The following figure and listing provide an example: Figure 11 shows a feature type with a number of properties. Listing 21 illustrates how the UML properties are represented within the "properties" of the JSON Schema that defines that type.

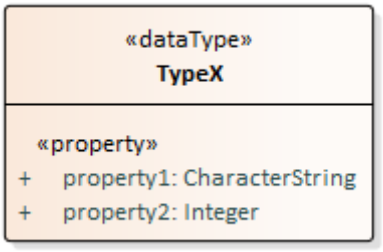

Figure 11. UML type used to exemplify JSON Schema encoding of UML properties

Listing 21. Encoding UML properties in JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "definitions": {
    "TypeX": {
      "type": "object",
      "properties": {
        "property1": { ... },
        "property2": { ... },
        ...
      }
    }
  },
  "$ref": "#/definitions/TypeX"
}
```

6.2.4.1. Value Type

If a mapping is defined for the value type of a UML property, then the JSON value type or JSON Schema defined by the mapping is used in the JSON Schema that constrains the property:

- If the value type maps to a simple JSON value type, i.e., "string", "number", or "boolean", then a "type" key is added to the JSON Schema, with the JSON value type as value;
- Otherwise, the mapping references a JSON Schema that defines the value type. In that case, a "$ref" key is added to the JSON Schema that constrains the property, with the reference defined by the mapping as value.

If no mapping is available, then:

- Typically the value type is defined by one of the application schemas that are converted, and the value type itself is also converted (encoding of a type or property can be prevented with an additional rule, see Additional rules). A "$ref" key is added to the JSON Schema that constrains the property. The value of that key references the definition of the type within the definitions schema that is produced for the schema to which the type belongs. The reference uses the identifier of that schema (see Schema Identifier) as a basis, and adds a fragment identifier to identify the definition of the type. If *rule-json-cls-name-as-anchor* (see Location Independent Schema Identifiers) applies to the type, then the type name is used as fragment identifier. Otherwise, a JSON Pointer will be created. For example:
  - using anchor: https://example.org/schemaA/schema_definitions.json#TypeX
  - using JSON Pointer:
    - For a draft 07 JSON Schema: https://example.org/schemaA/schema_definitions.json#/definitions/TypeX
    - For a 2019-09 JSON Schema: https://example.org/schemaA/schema_definitions.json#/$defs/TypeX
- If the value type is not defined by an application schema that is being converted, or the type

itself is not converted at all, then ShapeChange will log an error and omit the type definition for the property altogether.

A specific rule has been added to support value type restrictions for UML properties, which in the NAS are defined using OCL constraints (for further details - and especially an example, see section Constraints): *rule-json-cls-valueTypeOptions*. This rule looks for tagged value *valueTypeOptions* on a class. If the tag exists and has a value, it defines which types are allowed as value type for a given UML property. Note that this UML property can be directly defined on the class but also be inherited from a supertype. The property can also originally have been an association role that belonged to an association class. The conversion rule ensures that instead of the actual value type of the property, only one of the allowed types is encoded as type definition in the JSON Schema. The conversion also takes into account that the property may have been a role of an association class. The restriction to a set of allowed types uses an if-then-else construct, which depends on the presence of a type identifying member (see section Type Identification) in property values, and thus *rule-json-cls-valueTypeOptions* should always be used in combination with *rule-json-cls-name-as-entityType*. Note that value type restrictions (defined on a subtype) of inherited UML properties will result in these properties being explicitly defined in the JSON Schema definition of the subtype. The JSON Schema types of the allowed (UML) types are determined as described before. Further details on how the tagged value *valueTypeOptions* is structured and how it can be derived from OCL constraints are given in section Transforming OCL Constraints Defining Value Type Restrictions.

The behavior described so far covers the case of an inline encoding of the property value. In some cases, particularly if the value type is a type with identity, it can be preferable and maybe even necessary to encode the value by reference. In other cases, both options should be offered. That is similar to what the GML Application Schema encoding rules support (for further details, see OGC 07-036r1, Annex E, section E.2.4.11).

NOTE	An example where a reference to an object is needed is when the object is the value of properties from multiple other objects that are encoded within the same JSON document. For example, a feature referenced from several other features. In such a situation, it is often desirable not to encode the object inline multiple times - especially if that object also referenced other objects.

UML properties within an application schema typically have a tagged value *inlineOrByReference*, with one of three values: *inlineOrByReference*, *byReference*, or *inline*.

The default value (for an empty or missing *inlineOrByReference* tagged value) is defined via the ShapeChange JSON Schema target parameter *inlineOrByReferenceDefault*. The default value of that parameter is *byReference* . That default value is different to GML. *byReference* has been chosen as default in order to reduce the degrees of freedom and to reduce the schema complexity.

NOTE

If association roles within an application schema had tagged value *inlineOrByReference* all set to *inlineOrByReference*, then setting target parameter *inlineOrByReferenceDefault* would have no effect. A by reference encoding of association roles can still be achieved with ShapeChange, using a model transformation (e.g., the identity transformation), and setting tagged values during the post-processing phase of that transformation [https://shapechange.net/transformations/common-transformer-functionality/#Setting_Tagged_Values]. More specifically, one would instruct ShapeChange to set tagged value *inlineOrByReference* for all association roles to *byReference*.

When encoding the value type of a UML property in JSON Schema, the *inlineOrByReference* tagged value is taken into account:

- If the tag value is *inline*, then the behavior described above is applied;
- Otherwise, if the tag value is *byReference*, then by default the "type" key of the property will be defined with value "string" and `"format": "uri"`.

NOTE

The default behavior can be overridden by setting the ShapeChange JSON Schema target parameter *byReferenceJsonSchemaDefinition*. The parameter value is a URI to a JSON Schema definition of a link object, for example as shown in Listing 18. In actual JSON data, such a link object would be used to encode the reference.

- Otherwise, i.e., the tag value is *inlineOrByReference*, the two options above are combined using the "oneOf" keyword.

NOTE

The result is an XOR type of check, i.e., a value can either be given inline or by reference, but not both. This is different to GML, where in the case of *inlineOrByReference* and a complex value type a value can be given both inline and by reference.

This is restricted to properties where the value is a type with identity, which is not mapped to a simple JSON Schema type. Otherwise the value is always encoded inline.

NOTE

Some applications may prefer to reference types with identity using a code (of type string or number) instead of using a URI. That code could be seen as a foreign key. In such cases, a model transformation should be applied first, which, for all properties whose value type is a type with identity, replaces the value type with *CharacterString* or *Number*. The ShapeChange TypeConverter transformer could be enhanced to support such a transformation.

NOTE Current conversion behavior for value types does not enable by reference encoding for value types that are data types. In general, a data type does not have identity, and therefore a data type value should always be encoded inline, not by reference. The XML Schema encoding rule defined by ISO 19139:2007, typically used to encode metadata schemas (as defined by ISO 19115, and extensions thereof), on the other hand, allows by reference encoding for data type values. When comparing the previous version of ISO 19115 (from 2003/2006) against the current version (from 2014), we can see that some classes that were defined as data types in the previous version are now defined as object types, for example *CI_Citation*. This indicates that the assignment of the <<dataType>> stereotype has been corrected, in order to reflect in the conceptual model that the type shall be a type with identity.

NOTE If specialization needed to be supported (for further details, see section Class Specialization and Property Ranges), then the logic for determining the value type would need to be extended, to cover cases where the value type is a supertype. A particular example of such a situation can be found in the NAS, where an ISO type is used as value type, with the NAS actually defining a subtype of that ISO type.

6.2.4.2. Multiplicity

If the minimum cardinality of a UML property is 1 or greater, then the property will be listed under the "required" properties of the object to which the property belongs.

NOTE Specific conversion rules may override this behavior, for example the rules for converting a <<union>> (see section Union).

In addition, if the maximum cardinality of the property is greater than 1, then a JSON Schema will be created for the property as follows.

- The "type" of the JSON property is set to "array", with the "items" keyword containing the JSON Schema constraints that are created to represent the value type of the property.
- If the minimum cardinality is greater than 0, it is encoded using the "minItems" keyword.
- If the maximum cardinality is not unbounded, it is encoded using the "maxItems" keyword.
- If the values of the property must be unique (which is the default for UML properties), then that is represented by adding `"uniqueItems": true`.

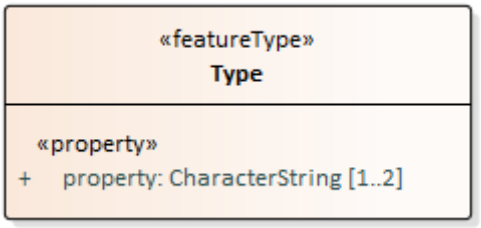

Figure 12. UML type used to exemplify JSON Schema encoding of multiplicity

Listing 22. Example for encoding multiplicity in JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "definitions": {
    "Type": {
      "type": "object",
      "properties": {
        "property": {
          "type": "array",
          "minItems": 1,
          "maxItems": 2,
          "items": {
            "type": "string"
          },
          "uniqueItems": true
        }
      },
      "required": [
        "property"
      ]
    }
  },
  "$ref": "#/definitions/Type"
}
```

This JSON object is valid against the schema from Listing 22:

```
{
"property": ["a","b"]
}
```

This JSON object is invalid (because "property" has three values, which exceeds the maximum amount of allowed values) against the schema from Listing 22:

```
{
"property": ["a","b",""]
}
```

NOTE All arrays in JSON are ordered, thus that the values of a UML property are ordered is always represented, and that the values of such a property are unordered cannot be represented. However, the latter should not matter to an application that does not expect ordered values for a certain property.

NOTE	An alternative approach for encoding a UML property with maximum multiplicity greater than one and a minimum multiplicity of 0 or 1 would be to allow either the type or an array of the type, so that a single value does not need to be encoded as an array. However, it is unclear if JSON tools generally support such an approach, i.e., encoding JSON member values as either a single value or within an array. Therefore, no conversion rule to support the alternative approach has been defined yet.

6.2.4.3. Voidable

With *rule-json-prop-voidable*, the JSON Schema of a UML property with stereotype <<voidable>>, or with tagged value *nillable* = true, is defined in a way that only allows either a null value or a(n array of) actual value(s).

- If the UML property has maximum multiplicity 1, then a simple "type" restriction with value "null" is added to the type definition that is produced for the property.
- Otherwise - the maximum multiplicity is greater than 1 - a choice (encoded using the "oneOf" keyword) between a "null" value and an array of actual values will be created.

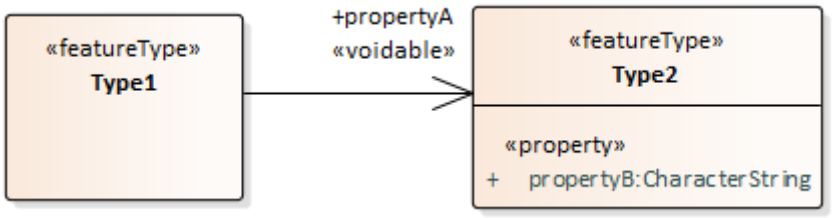

Figure 13. Example for JSON Schema encoding of a voidable property with max multiplicity 1

Listing 23. Encoding a voidable UML property with max multiplicity 1 in JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "definitions": {
    "Type1": {
      "type": "object",
      "properties": {
        "propertyA": {
          "oneOf": [
            {
              "type": "null"
            },
            {
              "$ref": "#/definitions/Type2"
            }
          ]
        }
      },
      "required": [
        "propertyA"
      ]
    },
    "Type2": {
      "type": "object",
      "properties": {
        "propertyB": {
          "type": "string"
        }
      },
      "required": [
        "propertyB"
      ]
    }
  },
  "$ref": "#/definitions/Type1"
}
```

The following two JSON objects are valid against the schema from Listing 23:

```
{
  "propertyA": null
}
```

```
{
  "propertyA": {
    "propertyB": "x"
  }
}
```

This JSON object is invalid (because "propertyB" is not allowed to be null) against the schema from Listing 23:

```
{
  "propertyA": {
    "propertyB": null
  }
}
```

If propertyA from the example shown in Figure 13 had maximum multiplicity of "*", then the resulting JSON Schema would be as in Listing 24

Listing 24. Encoding a voidable UML property with max multiplicity greater than 1 in JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "definitions": {
    "Type1": {
      "type": "object",
      "properties": {
        "propertyA": {
          "oneOf": [
            {
              "type": "null"
            },
            {
              "type": "array",
              "minItems": 1,
              "items": {
                "$ref": "#/definitions/Type2"
              },
              "uniqueItems": true
            }
          ]
        }
      },
      "required": [
        "propertyA"
      ]
    },
    "Type2": {
      "type": "object",
      "properties": {
        "propertyB": {
          "type": "string"
        }
      },
      "required": [
        "propertyB"
      ]
    }
  },
  "$ref": "#/definitions/Type1"
}
```

To encode the nil/null reason a separate, parallel property "xyz_nilReason" (or similar) could be added. This is implemented using a new transformation, not as part of the JSON Schema encoding rule.

6.2.4.4. Fixed / readOnly

With *rule-json-prop-readOnly*, the JSON Schema definition of a UML property that is read only or

fixed will include the "readOnly" annotation with JSON value true.

NOTE	With *rule-json-prop-derivedAsReadOnly*, a UML property marked as derived will also be encoded with `"readOnly": true`.

6.2.4.5. Initial Value

With *rule-json-prop-initialValueAsDefault*, the JSON Schema definition of a UML attribute that has an initial value, is not owned by an enumeration or code list, and whose value type is mapped to "string", "number", or "boolean", will include the "default" annotation with that value.

NOTE	The value of the annotation can have any JSON value type. The initial value is encoded accordingly: quoted, if the property type is "string", unquoted if the property type is "number", and true if the property type is "boolean" and the initial value is equal to, ignoring case, "true"; otherwise the value will be false. Theoretically, the default value can also be a JSON array or object, but that cannot be represented in UML and thus is not a relevant use case.

6.2.5. Association Class

There is no native represention for association classes in JSON or JSON Schema. For schemas that include association classes, a transformation of association classes as defined by GML 3.3 and implemented by the ShapeChange Association Class Mapper [https://shapechange.net/transformations/association-class-mapper/] should be used.

6.2.6. Constraints

OCL constraints can be used to enrich a conceptual model with requirements that cannot be expressed in UML alone. A full analysis of options for converting OCL expressions to something with which JSON data can be checked is out-of-scope for UGAS-2020, and therefore future work.

However, a particular type of OCL constraint defined in the NAS has been identified as critical for achieving a useful JSON Schema encoding of the NAS in UGAS-2020: constraints that disallow related entity types. Figure 14 provides an example.

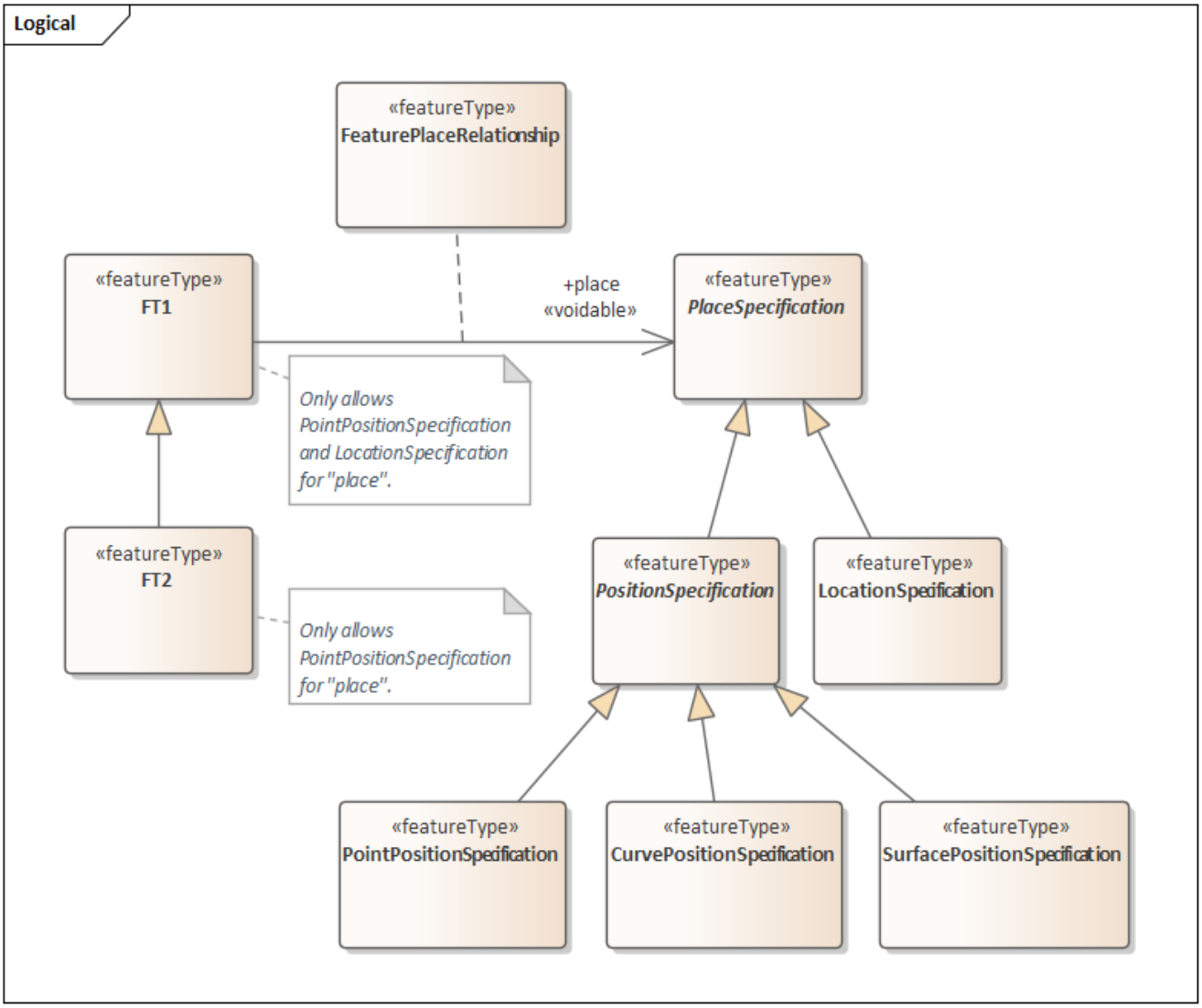

Figure 14. Example where OCL constraints restrict the set of allowed value types for property "place"

The value type of property *place* is the abstract type *PlaceSpecification*. That type is the root of an inheritance hierarchy, which contains four non-abstract classes. With typical UML semantics, the value of *place* can thus be any non-abstract subtype of *PlaceSpecification*. In the given example, that would be one of the types *PointPositionSpecification*, *CurvePositionSpecification*, *SurfacePositionSpecification*, and *LocationSpecification*. Note that further characteristics of these types, e.g., UML properties, have been omitted for brevity.

Diagram notes attached to the feature types *FT1* and *FT2* indicate that only a subset of non-abstract *PlaceSpecification* subtypes are actually allowed as value of property *place*. The according OCL constraints are defined as follows:

- FT1 - OCL constraint "Place Representations Disallowed": `inv: place→forAll(p| not(p.oclIsKindOf(CurvePositionSpecification) or p.oclIsKindOf(SurfacePositionSpecification)))`
- FT2 - OCL constraint "Place Representations Disallowed": `inv: place→forAll(p| not(p.oclIsKindOf(CurvePositionSpecification) or p.oclIsKindOf(SurfacePositionSpecification) or p.oclIsKindOf(LocationSpecification)))`

NOTE	Because the OCL constraint of FT2 has the same name as that of FT1, it overwrites the constraint that would be inherited from FT1.

In order to realize the value type restriction defined by such an OCL constraint, a model transformation - described in section Transforming OCL Constraints Defining Value Type Restrictions has been added to ShapeChange. In short, the transformation determines which value types are allowed for a UML property of a class (given that a value type restriction is defined for that property), and adds information about the allowed types to the model using a tagged value. That tagged value is used by *rule-json-cls-valueTypeOptions* (see section Value Type) to encode the value type restriction in JSON Schema.

NOTE

The model transformation also determines if the property is an association role whose association actually is an association class. If so, that information is added to the tagged value. As described in section Association Class, JSON Schema cannot directly represent association classes, and therefore such model constructs need to be transformed as defined by the GML 3.3 encoding rules. The resulting model structure is taken into account by *rule-json-cls-valueTypeOptions*, as shown in the following example.

Listing 25 shows how the value type restrictions for property *place* in the example used in this section are encoded in JSON Schema, assuming an inline encoding of *place* to achieve a more simple JSON Schema example.

Listing 25. Encoding a value type restriction in JSON Schema

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$defs": {
    "CurvePositionSpecification": { ... },
    "FeaturePlaceRelationship": {
      "$anchor": "FeaturePlaceRelationship",
      "type": "object",
      "properties": {
        "@type": {
          "type": "string"
        },
        "place": {
          "$ref": "#/$defs/PlaceSpecification"
        }
      },
      "required": [
        "@type",
        "place"
      ]
    },
    "LocationSpecification": { ... },
    "PlaceSpecification": { ... },
    "PointPositionSpecification": { ... },
    "PositionSpecification": { ... },
    "SurfacePositionSpecification": { ... },
    "FT1": {
      "$anchor": "FT1",
      "type": "object",
```

```
    "properties": {
      "@type": {
        "type": "string"
      },
      "place": {
        "oneOf": [
          {
            "type": "null"
          },
          {
            "allOf": [
              {
                "$ref": "#/$defs/FeaturePlaceRelationship"
              },
              {
                "type": "object",
                "properties": {
                  "place": {
                    "if": {
                      "properties": {
                        "@type": {
                          "const": "LocationSpecification"
                        }
                      }
                    },
                    "then": {
                      "$ref": "#/$defs/LocationSpecification"
                    },
                    "else": {
                      "if": {
                        "properties": {
                          "@type": {
                            "const": "PointPositionSpecification"
                          }
                        }
                      },
                      "then": {
                        "$ref": "#/$defs/PointPositionSpecification"
                      },
                      "else": false
                    }
                  }
                }
              }
            ]
          }
        ]
      }
    },
    "required": [
      "@type",
```

```
        "place"
      ]
    },
    "FT2": {
      "$anchor": "FT2",
      "allOf": [
        {
          "$ref": "#/$defs/FT1"
        },
        {
          "type": "object",
          "properties": {
            "place": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "allOf": [
                    {
                      "$ref": "#/$defs/FeaturePlaceRelationship"
                    },
                    {
                      "type": "object",
                      "properties": {
                        "place": {
                          "if": {
                            "properties": {
                              "@type": {
                                "const": "PointPositionSpecification"
                              }
                            }
                          },
                          "then": {
                            "$ref": "#/$defs/PointPositionSpecification"
                          },
                          "else": false
                        }
                      }
                    }
                  ]
                }
              ]
            }
          }
        }
      ]
    }
  },
  "$ref": "#/$defs/FT2"
}
```

The following JSON object is valid against the schema for FT2.

```
{
  "@type": "FT2",
  "place": {
    "@type" : "FeaturePlaceRelationship",
      "place": {
        "@type": "PointPositionSpecification"
      }
  }
}
```

The next JSON object is invalid against the schema for FT2, because the @type of the *PlaceSpecification* object that is the value of the *FeaturePlaceRelationship.place* property is *LocationSpecification* - which is allowed for FT1, but not for FT2.

```
{
  "@type": "FT2",
  "place": {
    "@type" : "FeaturePlaceRelationship",
      "place": {
        "@type": "LocationSpecification"
      }
  }
}
```

NOTE That *place* is a mandatory property results in "place" being added to the "required" member of the JSON Schema for FT1. There is no need to repeat this requirement in the encoding of FT2. However, other definitions would need to be repeated to achieve a consistent JSON Schema, such as that the value of the property is an array (if maximum multiplicity of the property is greater than 1), and that a null value is allowed (if the property is voidable).

6.2.7. Additional rules

If *rule-json-all-notEncoded* applies to an element of the application schema, then that element and all its components are not encoded.

NOTE How to define the encoding rule that applies to an application schema element is documented in more detail here. The ShapeChange configuration file StandardRules.xml [http://shapechange.net/resources/config/StandardRules.xml] defines an encoding rule named "notEncoded", which includes *rule-json-all-notEncoded*. When StandardRules.xml is included in the configuration of the JSON Schema target (typically using an xinclude XML element), then by setting tagged value *jsonEncodingRule* to "notEncoded", one would achieve that that model element is not encoded in the JSON Schema.

6.3. Instance Conversion Rules

This section documents recommendations and relevant aspects for encoding geospatial data in JSON.

6.3.1. Coordinate Reference System in JSON data

The GeoJSON standard requires that all GeoJSON coordinates use *urn:ogc:def:crs:OGC::CRS84* as coordinate reference system (CRS), with optional height in meters above or below the WGS 84 reference ellipsoid. Alternative CRSs are not allowed by this standard.

The OGC API - Features - Part 1: Core standard essentially has the same requirement:

> Unless the client explicitly requests a different coordinate reference system, all spatial geometries SHALL be in the coordinate reference system http://www.opengis.net/def/crs/OGC/1.3/CRS84 (WGS 84 longitude/latitude) for geometries without height information and http://www.opengis.net/def/crs/OGC/0/CRS84h (WGS 84 longitude/latitude plus ellipsoidal height) for geometries with height information.
>
> — OGC API - Features - Part 1: Core; section 7.11

A previous version of the GeoJSON standard did allow alternative CRSs, but that option has been removed *"because the use of different coordinate reference systems has proven to have interoperability issues"* (GeoJSON standard, chapter 4). However, the GeoJSON standard also states that if all involved parties have a prior arrangement, then alternative CRSs can be used.

The OGC API - Features - Part 1: Core standard also does not preclude the use of additional CRSs, but does not specify how to request features in such reference systems. That is the purpose of *OGC API - Features - Part 2: Coordinate Reference Systems by Reference*, which is currently under development.

To summarize: Coordinates of GeoJSON compliant data as well as spatial data accessed using the *OGC API - Features* standard must be given in OGC CRS84 (with optional height). That has implications for data publishers and consumers. A major benefit is increased interoperability, because spatial datasets that use the same CRS can easily be merged. If a use case requires other CRSs, then both GeoJSON and the OGC API - Features standards have options to support that. Data publisher and consumer only need to agree on the CRS in which the coordinates of spatial data, that is being exchanged between the two, is given in.

6.4. Conceptual Model Transformation Rules

The conceptual schema may need to be transformed, in order to deal with model elements:

- that cannot be represented in JSON at all (e.g., association classes);
- that cannot be represented in a certain JSON format (e.g., a Solid - a 3D geometry type - as value for the "geometry" member of a GeoJSON feature); or
- that are not (well) supported by client software (e.g., complex attribute values for styling,

processing, and filtering).

The following sections describe model transformations that can be useful to deal with these restrictions when encoding application schemas as JSON Schemas.

6.4.1. Flattening Inheritance

As mentioned in section Inheritance, JSON Schema does not directly support the concept of inheritance. There are ways to represent inheritance in JSON Schema to a certain extent. Generalization can be represented using an "allOf" that includes the schema of a subtype and the schema(s) of its supertype(s). Specialization can be represented as a relatively complex if-then-else construct, with which JSON objects - that encode types from a type hierarchy) can be validated based upon a property value that declares their type.

If a community does not want to apply these solutions for representing inheritance in JSON Schema, but still wants to use inheritance in their concpetual model and derive a JSON Schema encoding from it, then inheritance needs to be transformed on the conceptual level. Generalization (in the sense of Class Generalization and Property Inheritance) would be transformed by copying all properties of a supertype down to direct and indirect subtypes. For each supertype, a union of the non-abstract types in the hierarchy of that supertype (including the supertype itself) would be created, and the value type of each property that is that supertype would be switched to the union. That would allow the encoding of subtypes, instead of the supertype, as property value - and thus support specialization (in the sense of Class Specialization and Property Ranges).

The transformation of inheritance is implemented by the ShapeChange *Flattener* transformer, in *rule-trf-cls-flatten-inheritance*. The documentation of the rule [https://shapechange.net/transformations/flattener/#rule-trf-cls-flatten-inheritance] provides further details.

NOTE

The introduction of new unions as value types, for properties that have a supertype as value type, creates a level of indirection that may not be desirable. One approach to avoid the indirection would be to flatten the unions using another transformation, i.e., Flattening Complex Types (though that would create additional properties and remove the union semantics). Another approach would be to encode these unions as object references. This approach has been used in the creation of a GML-SF Level 0 XML Schema in OGC Testbed 13 (for further details, see the OGC Testbed-13: NAS Profiling Engineering Report, section 7.2.19. XML Schema encoding, rule-xsd-cls-union-omitUnionsRepresentingFeatureTypeSets [http://docs.opengeospatial.org/per/17-020r1.html#GMLSF0_XSD].

6.4.2. Flattening Multiplicity

Simple JSON formats may not support properties with a maximum multiplicity greater than 1. If the conceptual model contains such properties, they can be transformed to a set of properties, each with maximum multiplicity = 1.

This kind of model transformation is implemented in ShapeChange by rule-trf-prop-flatten-multiplicity [https://shapechange.net/transformations/flattener/#rule-trf-prop-flatten-multiplicity] of the *Flattener* transformer.

6.4.3. Flattening Complex Types

A community may want to keep their JSON formats simple by not allowing nested objects within a JSON object, or only a bare minimum (e.g., when their JSON shall be GeoJSON compliant). In such a case, nested objects that represent data types and unions from the conceptual model must be avoided. At the same time, the JSON formats used by the community must still be able to represent the information items that would usually be encoded via these complex types.

rule-trf-prop-flatten-types [https://shapechange.net/transformations/flattener/#rule-trf-prop-flatten-types], implemented by the ShapeChange *Flattener* transformer, "flattens" these complex types by copying their properties to the types that use the complex types. The names of the property copies are modified to reflect which information items they represent.

6.4.4. Mapping Association Classes

JSON Schema cannot directly represent association classes. An association class therefore needs to be transformed into a structure that can be represented with JSON Schema. The transformation of association classes defined by GML 3.3 is a suitable solution.

The ShapeChange Association Class Mapper [https://shapechange.net/transformations/association-class-mapper/] implements the transformation defined by GML 3.3.

6.4.5. Transforming Stereotype <<propertyMetadata>>

In the UGAS-2019 OGC Pilot, the <<propertyMetadata>> stereotype was developed. When assigned to a property, it indicates that the property can be associated with metadata. The metadata would provide additional information on the property value or values. Tagged value *metadataType* is defined for the stereotype, and used to identify the actual type that the property references as metadata.

The Property Stereotype for Metadata [https://shapechange.net/wp-content/uploads/2019/12/UGAS19-D100_property_stereotypes.pdf] document describes how the stereotype is encoded in XML Schema. Basically, a "metadata" XML attribute is added on the XML element that represent the property. The XML attribute can be used to reference the metadata object, much like an "xlink:href" XML attribute would be used to reference a "normal" object.

Some encodings - like JSON - do not have anything similar to XML attributes. Section 2.3.2 of the Property Stereotype for Metadata [https://shapechange.net/wp-content/uploads/2019/12/UGAS19-D100_property_stereotypes.pdf] document describes how the <<propertyMetadata>> stereotype can be handled in such an encoding. In essence, the stereotype is transformed to an additional property, with the type identified by tagged value *metadataType* as value type.

The transformation is implemented by the ShapeChange *Type Converter* transformer in *rule-trf-propertyMetadata-stereotype-to-metadata-property*.

- Rule behavior: Converts the <<propertyMetadata>> stereotype to an additional property, as follows: First, the metadata type that applies to the property with the stereotype is identified: The tagged value *metadataType* of the property is checked first. If the tagged value does not identify a metadata type, then the type defined by configuration parameter

defaultMetadataType is used.

NOTE

The identification of the metadata type by tagged value or by configuration parameter is as follows:

- definition by tagged value *metadataType*: If the type is defined by the schema that contains the property, then the tagged value simply provides the name of the type. Otherwise, the tagged value shall identify the type by its full package-qualified name, starting with the application schema package. For example: "Some Application Schema::Some Subpackage::Another Subpackage::MetadataType."
- definition by configuration parameter *defaultMetadataType*: If the name of the type is unique within the conceptual model, then simply providing the type name as parameter value is sufficient. Otherwise (or as a general alternative), the metadata type is identified by providing its full name (omitting packages that are outside of the schema the class belongs to - see the example above).

If the configuration parameter also does not identify a type within the conceptual model, an error message will be logged and the stereotype will simply be removed from the property. Otherwise, if the metadata type is a type with identity (feature or object type) then a directed association to the metadata type is created - else an attribute (with the metadatatype as value type) is created. The name of the new association role or attribute is the property name plus suffix defined by configuration parameter *metadataPropertyNameSuffix*. If a new association role was created, tagged value *inlineOrByReference* of the association role is set to the value defined by configuration parameter *metadataPropertyInlineOrByReference*. Otherwise, i.e., an attribute was created, tagged value *inlineOrByReference* is set to "inline." Tagged value *sequenceNumber* will be set in such a way that the new property is placed directly after the original property.

- Configuration parameter *defaultMetadataType*: Name of the type from the conceptual model, which shall be used as metadata type for all properties with stereotype <<propertyMetadata>> that do not define a metadata type via tagged value *metadataType*. The value can be the pure type name, if it is unique within the conceptual model. Otherwise, the correct type is identified by providing its full name (omitting packages that are outside of the schema the class belongs to). The default value for this parameter is 'MD_Metadata' (which typically refers to the type defined by ISO 19115).
- Configuration parameter *metadataPropertyNameSuffix*: Defines the suffix that shall be added to the name of a new property created by *rule-trf-propertyMetadata-stereotype-to-metadata-property*. Default is'_metadata'.
- Configuration prameter *metadataPropertyInlineOrByReference*: Defines the value for tag *inlineOrByReference* of a new association role created by *rule-trf-propertyMetadata-stereotype-to-metadata-property*. Default is 'inlineOrByReference'. Other allowed values are 'byReference' and 'inline'.

6.4.6. Generating NilReason Properties for Nillable Properties

A UML property that is defined for a feature, object, data, or union type within an application schema, by default cannot have a null value. In order to model that a property can have a null value (instead of actual value(s)), stereotype <<voidable>> must be added to the property, or tagged value *nillable* with value 'true'. Such a property is called a *nillable* property.

The recommendation from the UGAS-2019 OGC Pilot for NAS modelling of nillable properties (for further details, see the Property Stereotype for Metadata [https://shapechange.net/wp-content/uploads/2019/12/UGAS19-D100_property_stereotypes.pdf] document, section 2.2) is to use the <<voidable>> stereotype, and to also set tagged value *voidReasonType* on the property, in order to define the enumeration that defines the reasons for a null value.

In the XML Schema encoding, nillable properties are represented by XML elements for which the XML attribute "nilReason" can be set. As also mentioned in section Transforming Stereotype <<propertyMetadata>>, some encodings - like JSON - do not have anything similar to XML attributes. In order to encode the reason why a nillable property has a null value in such an encoding, the application schema can be transformed, adding a new property for each nillable property (to the class that owns the nillable property), with the new property having the void reason type as value type.

The according transformation is implemented by the ShapeChange *Type Converter* transformer in *rule-trf-nilReason-property-for-nillable-property*.

- Rule behavior: For each property that is nillable (has stereotype <<voidable>> or tagged value *nillable* set to 'true'), create a new attribute, as follows: First, the void reason type that applies to the nillable property is identified: The tagged value *voidReasonType* of the nillable property is checked first. If the tagged value does not exist or does not identify a type, then the type defined by configuration parameter *defaultVoidReasonType* is used.

  NOTE

  The identification of the void reason type by tagged value or by configuration parameter is as follows.

  - Definition by tagged value *voidReasonType*: If the type is defined by the schema that contains the property, then the tagged value simply provides the name of the type. Otherwise, the tagged value shall identify the type by its full package-qualified name, starting with the application schema package. For example: "Some Application Schema::Some Subpackage::Another Subpackage::VoidReasonType".
  - Definition by configuration parameter *defaultVoidReasonType*: If the name of the type is unique within the conceptual model, then simply providing the type name as parameter value is sufficient. Otherwise (or as a general alternative), the void reason type is identified by providing its full name (omitting packages that are outside of the schema the class belongs to - see the example above).

  If the configuration parameter also is not set or does not identify a type within the conceptual model, an error message will be logged and the value type of the new attribute will be CharacterString. Otherwise, the identified type will be set as value type of the new attribute. The

name of the new attribute is the name of the nillable property plus suffix defined by configuration parameter *nilReasonPropertyNameSuffix*. Tagged value *inlineOrByReference* of the new attribute is set to inline. Tagged value *sequenceNumber* will be set in such a way that it is placed directly after the nillable property.

- Configuration parameter *defaultVoidReasonType*: Name of the type from the conceptual model, which shall be used as void reason type for all nillable properties that do not define a void reason type via tagged value *voidReasonType*. The value can be the pure type name, if it is unique within the conceptual model. Otherwise, identify the correct type by providing its full name (omitting packages that are outside of the schema the class belongs to). No default value is defined for this parameter.
- Configuration parameter *nilReasonPropertyNameSuffix*: Defines the suffix that shall be added to the name of a new property created by *rule-trf-nilReason-property-for-nillable-property*. Default is'_nilReason'.

6.4.7. Transforming OCL Constraints Defining Value Type Restrictions

An OCL constraint such as `inv: place→forAll(p|not(p.oclIsKindOf(CurvePositionSpecification) or p.oclIsKindOf(SurfacePositionSpecification)))`, and - for the sake of the example used in this section - name "Value Type Representations Disallowed", restricts the set of allowed value types for a property. In the example, property *place* must not have a value of type *CurvePositionSpecification* or *SurfacePositionSpecification*. The example is described in more detail in section Constraints.

With *rule-trf-cls-constraints-valueTypeRestrictionToTV-exclusion* - defined for the ShapeChange *ConstraintConverter* transformation [https://shapechange.net/transformations/constraintconverter/], the value type restrictions defined by OCL constraints can be extracted from the OCL expression, and converted into a tagged value, to be used by subsequent transformation and conversion processes.

Configuration parameter *valueTypeRepresentationConstraintRegex* is used to identify the relevant OCL constraints. The parameter value contains a regular expression - for example `.*Value Type Representations Disallowed.*`, which matches the names of OCL constraints that define value type restrictions. The according OCL expressions must thereby be structured as in the example (with *oclIsTypeOf(..)* also being supported).

The name of the property that is restricted is parsed from the begin of the OCL expression: `inv: {propertyName}->forAll`... The property name may thereby be preceded by `self.`, i.e., `inv: self.{propertyName}->forAll`... is a valid alternative way to structure the value type restricting OCL expression.

Required configuration parameter *valueTypeRepresentationTypes* specifies the types that are used as value type by the UML properties identified in the value type restricting OCL constraints. For each such type, a list of names of the generally allowed types within the inheritance hierarchy of that type must be provided, which may include the type itself and abstract types. For example, for the value type *PlaceSpecification* of property *place* shown in Figure 14, the value of the configuration parameter could be: `PlaceSpecification{PointPositionSpecification, CurvePositionSpecification, SurfacePositionSpecification, LocationSpecification}`. The transformation will automatically add all subtypes of generally allowed types to the set of generally allowed types. That is important for creating a tagged value that explicitly lists the types that are allowed for a property, regardless of inheritance structures, because the OCL constraint may

exclude a specific subtype of a generally allowed supertype.

NOTE If multiple value types need to be described by configuration parameter *valueTypeRepresentationTypes*, then a semicolon is used to separate the descriptions in the parameter value.

The transformation will parse a value type restricting OCL constraint in order to determine the (potentially inherited) UML property to which the constraint applies. The OCL expression is structured so that any type mentioned in the expression is disallowed/excluded. The transformation can therefore determine the value types that are disallowed - also taking into account all subtypes of a type that is mentioned within an *oclIsKindOf(..)*. The set of disallowed types will then be subtracted from the set of generally allowed types, resulting in the set of types that are allowed as value types of the property.

The allowed types for the property are documented in the model by adding (also: overwriting, if it already exists) tagged value *valueTypeOptions* to the class on which the UML property is defined. The tagged value is structured as follows:

```
{propertyName}(\(associationClassRole\))?={allowedTypeName}(,{allowedTypeName})(;{propertyName}
(\(associationClassRole\))?={allowedTypeName}(,{allowedTypeName}))*
```

For the example OCL constraint, that would result in: `place(associationClassRole)=PointPositionSpecification,LocationSpecification`.

NOTE The example shows that the tagged value may contain a qualifier - *associationClassRole* - for a property, which, if set, indicates that the property is an association role whose association actually is an association class. That information can be relevant for subsequent processes, for example the JSON Schema encoding, when a previous model transformation has transformed association classes as defined by the GML 3.3 encoding rules.

NOTE Configuration parameter *valueTypeRepresentationTypes* can also define an alias for the name of an allowed type. For example: `PlaceSpecification{PointPositionSpecification=P, CurvePositionSpecification=C, SurfacePositionSpecification=S, LocationSpecification=L}`. The alias is used when constructing tagged value *valueTypeOptions*. That can be useful in case that the names of UML types contained in the model are flattened, i.e., replaced by a short name or code, by a subsequent model transformation. Processes that convert such a flattened model and use the information from tagged value *valueTypeOptions* then have the correct names of allowed value types.

6.5. Encoding Rules

This section documents two JSON Schema encoding rules, one for achieving a GeoJSON compliant JSON Schema encoding, and one for producing plain JSON Schemas (typically for non-geospatial schemas).

NOTE

Each of these two rules is implemented in ShapeChange, the first with name "defaultGeoJson", and the second with name "defaultPlainJson". Additional conversion rules can easily be added to such an encoding rule by defining a new encoding rule in the ShapeChange JSON Schema target that extends "defaultGeoJson" or "defaultPlainJson", and setting the new rule as the default encoding rule (using the ShapeChange JSON Schema target parameter *defaultEncodingRule*). If, on the other hand, conversion rules from "defaultGeoJson" or "defaultPlainJson" need to be removed or replaced, for any reason, then these encoding rules cannot be used (ShapeChange supports extending a named encoding rule, but not restricting it), and instead a new encoding rule must be defined that is patterned after the applicable existing rule.

NOTE

For some application schemas, it is useful to know that different encoding rules can be applied to the subpackages, classes, and properties defined by the schema. Typically, a single encoding rule applies to all application schema elements. In ShapeChange, that rule is identified by setting the JSON Schema target parameter *defaultEncodingRule*, with the unique name defined for the encoding rule in the target configuration. The target configuration, however, can contain multiple encoding rules (with different names). By setting tagged value *jsonEncodingRule* on an application schema element, using the name of another encoding rule, the model element will be encoded as defined by that rule. For example, if an application schema used unions in both ways described in section Union, then the default encoding rule could include *rule-json-cls-union-propertyCount*, and encoding rule "TypeDiscriminatorUnionRule" could instead include *rule-json-cls-union-typeDiscriminator*. By setting tagged value *jsonEncodingRule=TypeDiscriminatorUnionRule* on each type discriminator union, these unions would be encoded using *rule-json-cls-union-typeDiscriminator* and all other unions would be encoded using *rule-json-cls-union-propertyCount*.

6.5.1. GeoJSON Schema Encoding Rule

In order to achieve a GeoJSON compliant encoding using ShapeChange, set "defaultGeoJson" as the default encoding rule for the JSON Schema target (using the target parameter *defaultEncodingRule*).

This encoding rule consists of the following conversion rules:

- rule-json-cls-defaultGeometry-singleGeometryProperty
- rule-json-cls-ignoreIdentifier
- rule-json-cls-name-as-anchor
- rule-json-cls-nestedProperties
- rule-json-cls-virtualGeneralization
- rule-json-prop-derivedAsReadOnly
- rule-json-prop-initialValueAsDefault
- rule-json-prop-readOnly

- rule-json-prop-voidable

Furthermore, the following parameters need to be added to the configuration of the ShapeChange JSON Schema target:

- baseJsonSchemaDefinitionForFeatureTypes = https://geojson.org/schema/Feature.json
- baseJsonSchemaDefinitionForObjectTypes = https://geojson.org/schema/Feature.json

Geometry types used in the conceptual model (e.g., types from ISO 19107) must be mapped to one of the GeoJSON geometry types (see Table 5).

Table 5. Mapping ISO 19107 types to GeoJSON geometry types

Conceptual geometry type	**GeoJSON geometry type**
GM_Point	https://geojson.org/schema/Point.json
GM_Curve	https://geojson.org/schema/LineString.json
GM_Surface	https://geojson.org/schema/Polygon.json
GM_MultiPoint	https://geojson.org/schema/MultiPoint.json
GM_MultiCurve	https://geojson.org/schema/MultiLineString.json
GM_MultiSurface	https://geojson.org/schema/MultiPolygon.json
GM_Object	https://geojson.org/schema/Geometry.json

NOTE In order to achieve an OGC JSON encoding, the ShapeChange JSON Schema target parameters *baseJsonSchemaDefinitionForFeatureTypes* and *baseJsonSchemaDefinitionForObjectTypes* would have to be set to reference the schema for AnyFeature, shown in Listing 57. Furthermore, the geometry types would need to be mapped as defined in Table 8.

The feature type illustrated in Figure 15 is used as example for a GeoJSON based encoding, which is shown in Listing 26.

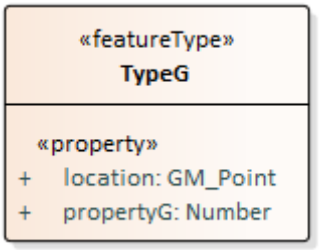

Figure 15. Example of a feature type in UML, which will be converted to a GeoJSON feature

Listing 26. JSON Schema example of a feature type that is converted using the default GeoJSON encoding rule

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "definitions": {
    "TypeG": {
      "$id": "#TypeG",
      "allOf": [
        {
          "$ref": "https://geojson.org/schema/Feature.json"
        },
        {
          "type": "object",
          "properties": {
            "properties": {
              "type": "object",
              "properties": {
                "propertyG": {
                  "type": "number"
                }
              },
              "required": [
                "propertyG"
              ]
            },
            "geometry": {
              "$ref": "http://geojson.org/schema/Point.json"
            }
          },
          "required": [
            "properties"
          ]
        }
      ]
    }
  },
  "$ref": "#/definitions/TypeG"
}
```

This JSON object is valid against the schema from Listing 26:

```
{
  "id": "42445fdasd7asd6f7",
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [8.195669, 51.903589]
  },
  "properties": {
    "propertyG": 3
  }
}
```

This JSON object is invalid against the schema from Listing 26 (because the JSON value of "geometry" is not valid according to the JSON Schema of a GeoJSON point):

```
{
  "id": "42445fdasd7asd6f7",
  "type": "Feature",
  "geometry": {
    "type": "LineString",
    "coordinates": [
      [102.0,0.0],
      [103.0,1.0],
      [104.0,0.0],
      [105.0,1.0]
    ]
  },
  "properties": {
    "propertyG": 3
  }
}
```

6.5.2. Plain JSON Schema Encoding Rule

Some communities have schemas where "geospatial" plays a minor or no role at all. For such cases, the use of GeoJSON features is not relevant. The property nesting and the restrictions of the "type" and "geometry" members defined by the GeoJSON schema could even be a hindrance.

In order to achieve a plain JSON Schema encoding using ShapeChange, set "defaultPlainJson" as the default encoding rule for the JSON Schema target (using the target parameter *defaultEncodingRule*).

This encoding rule consists of the following conversion rules:

- rule-json-cls-name-as-anchor
- rule-json-prop-derivedAsReadOnly
- rule-json-prop-initialValueAsDefault
- rule-json-prop-readOnly

- rule-json-prop-voidable

The geometry types used in the conceptual model should be mapped to the JSON Schema implementations defined by the Features Core Profile.

A plain JSON Schema encoding of the feature type that was used as example for the GeoJSON based encoding in the previous section (see Figure 15) is shown in Listing 27.

Listing 27. JSON Schema example of a feature type that is converted using the default plain encoding rule

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "definitions": {
    "TypeG": {
      "$id": "#TypeG",
      "type": "object",
      "properties": {
        "location": {
          "$ref": "http://geojson.org/schema/Point.json"
        },
        "propertyG": {
          "type": "number"
        }
      },
      "required": [
        "location",
        "propertyG"
      ]
    }
  },
  "$ref": "#/definitions/TypeG"
}
```

This JSON object is valid against the schema from Listing 27:

```
{
  "location": {
    "type": "Point",
    "coordinates": [
      8.195669,
      51.903589
    ]
  },
  "propertyG": 3
}
```

Chapter 7. Features Core Profile of Key Community Conceptual Schemas

7.1. Overview

Most application schemas for geospatial information, including the NAS, build on the comprehensive conceptual schemas from the OGC Abstract Specifications and the ISO 19100 series. This creates a challenge for JSON encodings of those application schemas. XML encodings can build on GML, the ISO-19139-based XML schemas for the metadata standards and the available tools supporting these schemas and standards, but only very few of the classes in the conceptual schemas have associated JSON schemas or JSON support in implementations. While the work documented in chapter UML to JSON Schema Encoding Rule provides the capability to create JSON schemas for the content defined in the application schema packages, JSON schemas for the classes that are imported from other ISO/OGC schemas do not exist yet and therefore cannot be referenced.

Existing JSON encodings for ISO/OGC schemas include the following.

- Features: GeoJSON and Esri JSON implement features and feature collections, although with limitations, for example, only one geometry property per feature. GeoJSON has a JSON schema and very good tool support. Esri JSON is also widely used in the ArcGIS platform; however, no official JSON schema exists.
- Spatial geometries: GeoJSON and Esri JSON implement the simple feature geometries. GeoJSON is essentially restricted to WGS84.
- Coordinate reference systems: PROJJSON [https://proj.org/specifications/projjson.html] is a JSON encoding of WKT2:2019 / ISO-19162:2019, which itself implements the model of OGC Topic 2: Referencing by coordinates, supported by the frequently used PROJ library.
- SWE Common: A JSON encoding exists as an OGC Best Practice document [http://docs.opengeospatial.org/bp/17-011r2/17-011r2.html], but without JSON schema. One implementation is known.

No known JSON schemas exist for other schemas, in particular the metadata and data quality schemas (ISO 19115-1, ISO 19115-2, ISO 19157), for additional spatial geometries beyond Simple Features (ISO 19107) or how time instances and intervals (ISO 19108) should be represented in features consistent with the General Feature Model (ISO 19109).

In order to specify a JSON encoding for the NAS or other application schemas, additional JSON schemas for the imported classes from base schemas defined by ISO/OGC are required.

At the same time, these base schemas are comprehensive and full support requires a significant implementation effort that is often difficult to justify since in practice only a small subset of the complete schema(s) are needed or used. For example, while Geography Markup Language (GML) implements only a subset of all curve segments or surface patches specified by ISO 19107, it is still a comprehensive subset and most of that subset is not, or is only very rarely, used in practice. To address this, the GML Simple Features Profile has been specified to identify a small subset that is sufficient for most use cases. Likewise, many of the metadata elements specified in the ISO 19115-3 XML encoding standards are optional and not used in most metadata XML instance documents –

and most communities have defined profiles of the comprehensive ISO metadata schemas to meet their narrower requirements.

Historically it has been the case that defining complete implementation schemas in XML Schema for these ISO standards, and then specifying a multiplicity of community-specific simpler, tailored profiles has limited the extent to which those complete implementation schemas are supported in software and thus contribute to effective interoperability.

Consistent with the approach taken by most of the recent OGC standards, any approach to generating JSON schemas for the base standards should start with a small core encoding of the classes supported by most datasets and implementations. Once proven in practice through implementations, the core can be extended based on community needs.

The definition of this core in the UGAS-2020 pilot has been based on this context. The profile is, therefore, called the "Features Core Profile of Key Community Conceptual Schemas", from now on simply called "Features Core Profile."

The NAS requirements with respect to spatial geometries, temporal information and metadata as it is typically included in existing NAS data (to the extent that this information is publicly available) have been taken into account.

The definition of the JSON schemas for the Features Core Profile has considered existing encodings as a starting point, where they exist, as well as the JSON encoding rule, but with optimizations for the data representation in JSON, similar to how gml:posList is an optimized XML implementation of a GM_PointArray from ISO 19107.

7.2. Scope of the Features Core Profile

Three categories of classes are used by almost all geospatial application schemas, including the NAS.

- Basic types defined by ISO 19103 - especially literal types for representing values such as CharacterString, Integer, Real, Boolean, Date, and DateTime. In addition, Measure and Any are frequently used, too.
- Spatial types defined by ISO 19107 – especially geometry types such as GM_Point, GM_Curve, and GM_Surface.

NOTE	The NAS baseline X-3 conceptual model does not directly contain ISO 19107. However, a relationship to the schema indirectly exists, through the conceptual schema for elements from ISO 19136. Table 8 defines a mapping between types from ISO 19107 and their counterparts in the ISO 19136 schema contained in the NAS conceptual model.

- Most spatial data has a temporal aspect, too. This may be represented using literal values (Date, DateTime) or using the temporal types defined by ISO 19108 - especially temporal geometry types such as TM_Instant and TM_Period.

The Features Core Profile is comprised of key types from these schemas – see Table 7, Table 8, and Table 9. This set of types represents a small, but useful basis for building geospatial applications

that exchange and process JSON encoded spatial data.

For the JSON encoding, the encoding of the feature types based on the General Feature Model defined by ISO 19109 had to be considered as well, building on the discussion in the chapter UML to JSON Schema Encoding Rule. For the Features Core Profile, a consistent approach to encode thematic, spatial, and temporal attributes was required. Since metadata or quality attributes, as well as association roles are also commonly used in application schemas, the JSON feature encoding also includes extensions for those feature property types.

The Features Core Profile does not include classes from the ISO metadata and data quality schemas (ISO 19115-1, 19115-2 and ISO 19157). This has several reasons.

- Classes from these schemas are less frequently used in geospatial application schemas.
- In the application schemas where they are used (for example, in the NAS), the profile is often comprehensive and it seems unlikely that a small core that is useful and sufficient for many use cases can be identified.
- Other conceptual metadata schemas are in broad use that should also be supported. To name a few: Dublin Core, DCAT and schema.org.

The approach taken by the UGAS-2020 pilot therefore was to not include classes from ISO 19115-1, ISO 19115-2 or ISO 19157 in the Features Core Profile, but to specify how JSON instances of those classes could be included in feature and feature collection data.

Nevertheless, application schemas like the NAS have a need to import JSON schemas for classes from these three ISO standards. To support such workflows, Appendix B documents how JSON schemas can be produced for these schemas using ShapeChange and the UML to JSON Schema Encoding Rule. These JSON schemas could be used as-is in a JSON encoding of the NAS or the result could be the starting point for additional optimizations. With such JSON Schemas, as well as the JSON schema definitions for the Features Core Profile, a JSON-Schema-based encoding for the NAS can be produced.

7.3. The Features Core Profile and its encoding in JSON

This section identifies the types from the ISO/TC 211 Harmonized Model that are part of the Features Core Profile.

It also specifies the mapping of these types to JSON Schema. In some cases this mapping is specified as a JSON Schema implementation of the type that can be referenced from an application schema in JSON Schema; these JSON schemas have an "$id" that starts with "http://www.opengis.net/tbd/" and could become part of the schema definitions of a future OGC JSON specification. In other cases, the JSON Schema implementation is always included inline and the mapping extends the UML-to-JSON-Schema encoding rule.

7.3.1. Overview

Table 6 provides an overview of the types from the ISO/TC 211 Harmonized Model included in the Features Core Profile.

Table 6. Types in the Features Core Profile

Basic types	Spatial types	Temporal types	Feature types
ISO 19103:2015	**ISO 19107:2003**	**ISO 19108:2002**	**ISO 19109:2015**
Boolean	GM_Point	TM_Instant	AnyFeature
Character	GM_MultiPoint	TM_Period	
CharacterString	GM_Curve		
Date	GM_MultiCurve		
DateTime	GM_Surface		
Decimal	GM_MultiSurface		
Integer	GM_Solid		
Number	GM_MultiSolid		
Real	GM_Object		
URI			
Measure	**ISO 19107:2019**		
Any	Point		
Record	Line		
RecordType	Polygon		
	Solid		
	Collection		
	Geometry		

While the table uses the types from the ISO/TC 211 harmonized model, this is not meant to imply that any valid instance of these types are covered by the profile.

For the purposes of defining a Features Core Profile of spatial types, the representations of all spatial 1d, 2d, and 3d geometric primitives in ISO 19107 and all temporal geometric primitives in ISO 19108 are restricted to the simple representations that are widely supported in software tools and libraries.

The following constraints apply for the spatial geometric primitive types.

- GM_Curve is constrained to a single curve segment with linear interpolation.
- GM_Surface is constrained to a single surface patch that is a GM_Polygon with planar interpolation where each ring is a closed GM_Curve.
- Each shell of a GM_Solid is constrained to be a closed GM_PolyhedralSurface where each patch is a GM_Polygon.

These constraints apply to all instances including geometric primitives that are part of a geometric aggregate.

For ISO 19107:2019, any Collection is constrained to be homogenuous, i.e., it is a collection of geometric primitives of the same dimension.

NOTE	GM_Aggregate has not been included in the profile, since it does not seem to be used much in practice.

The following constraints apply for the temporal geometric primitive types:

- TM_Instant and TM_Period are constrained to the temporal positions in the Gregorian calendar.

7.3.2. Basic Types

This section lists the types from ISO 19103 that belong to the Features Core Profile and documents their JSON Schema definition. The following sections provide more detail and specify the JSON encoding.

7.3.2.1. Simple Types

Most of the frequently used basic types are simple, i.e., the value can be represented by a single literal value. The mapping to the JSON Schema types and - where applicable - the JSON Schema format specifier and maybe additional JSON Schema keywords is shown in Table 7.

Table 7. JSON Schema implementation for simple types

Type	JSON Schema type	JSON Schema format	Additional JSON Schema validation keyword(s)
Boolean	boolean		
Character	string		"minLength" : 1 and "maxLength" : 1
CharacterString	string		
Date	string	date	
DateTime	string	date-time	
Decimal	number		
Integer	integer		
Number	number		
Real	number		
URI	string	uri	

7.3.2.2. Non-simple Types

In addition to the simple types, two additional non-simple types are included in the Features Core Profile since they are frequently used in application schemas:

- Measure (or any of its subtypes): Measure is a combination of a numeric value and the unit of measurement of the value;
- Any

7.3.2.2.1. Measure

Three alternative mappings to JSON Schema are specified. In the discussion below we use a property `length : Measure [0..1]` as an example.

Default representation

In the default mapping, the property could be represented as an object-valued property in JSON. The object should have two properties, one for the numeric value ("value") and one for the unit ("uom").

Listing 28. JSON Schema for Measure

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/Measure.json",
  "title": "A measure",
  "type" : "object",
  "required" : [ "value", "uom" ],
  "properties" : {
    "value" : {
      "type" : "number"
    },
    "uom" : {
      "type" : "string"
    }
  }
}
```

Listing 29. JSON example of the length property

```
{
  "length" : {
    "value" : 45.6,
    "uom" : "m"
  }
}
```

Listing 28 may be referenced directly from application schemas. However, it may also be desirable to specify constraints on the ranges of "value" and "uom". In that case, the Measure schema can be used as a template that is modified to include the constraints.

Listing 30. Example JSON schema for a customized Measure type

```
{
  "type" : "object",
  "required" : [ "value" ],
  "properties" : {
    "value" : {
      "type" : "number",
      "minimum" : 0
    },
    "uom" : {
      "type" : "string",
      "enum" : [ "m", "ft" ],
      "default" : "m"
    }
  }
}
```

Fixed units

For cases where the application schema uses a fixed value, the JSON Schema type "number" could be used directly. If desired, a suffix could be added to the name of the property to indicate the fixed unit. In this case, the symbol of the unit should be added, separated with an underscore. Example:

Listing 31. JSON Schema for a property with a fixed unit

```
{
  "type" : "object",
  "properties" : {
    "length_m" : {
      "type" : "number"
    }
  }
}
```

Listing 32. JSON example of the length property with a fixed unit

```
{
  "length_m" : 45.6
}
```

Flat representation

For cases where the unit of measurement may differ from feature to feature, where the maximum multiplicity is one (at least for all measure typed properties), and where a flat property structure is important for clients, the property could be represented as two properties in JSON, one for the numeric value and one for the unit. The property for the unit should have a suffix of "_uom" added to the property name. If the list of allowed units is known, these should be added as enum values. A dependency can be used to ensure that a unit is provided, if the numeric value is provided.

Listing 33. JSON Schema for a property with a flattened measure representation

```
{
  "type" : "object",
  "properties" : {
    "length" : {
      "type" : "number"
    },
    "length_uom" : {
      "type" : "string",
      "enum" : [ "m", "ft" ]
    }
  },
  "dependencies" : {
    "length" : [ "length_uom" ]
  }
}
```

Listing 34. JSON example of a property with a flattened measure representation

```
{
  "length" : 45.6,
  "length_uom" : "m"
}
```

In order to convert an application schema that has properties with a maximum multiplicity greater than one to an implementation schema where all properties have a maximum multiplicity of one, the ShapeChange Flattener [https://shapechange.net/transformations/flattener/] transformation can be used - more specifically, rule-trf-prop-flatten-multiplicity [https://shapechange.net/transformations/flattener/#rule-trf-prop-flatten-multiplicity].

NOTE

A new Flattener transformation rule has been implemented in UGAS-2020, rule-trf-prop-flatten-measure-typed-properties [https://shapechange.net/transformations/flattener/#rule-trf-prop-flatten-measure-typed-properties], which can be used to transform measure typed properties (essentially changing the type to *Numeric* and creating a "*..uom" property with value type _CharacterString*. The resulting model is closer to the flat representation. However, uom enums and property dependencies are not created. If it turns out that this is important, then future work can design and implement a solution for achieving the full encoding for a flat measure representation.

7.3.2.2.2. Any

Since any value is a valid instance of the type Any, the type is mapped to JSON Schema (see Listing 35) so that all values - simple JSON types and objects - are valid.

Listing 35. JSON Schema for Any

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/Any.json",
  "title": "Any value",
  "type": ["boolean", "number", "integer", "string", "object"]
}
```

The conversion of other property characteristics (multiplicity, voidable, inline or by reference encoding, uniqueness) of a property with value type *Any* should follow the same rules as described in Properties.

For example, an object with a property `metadata : Any [0..*]` would be implemented as

Listing 36. JSON Schema example for `metadata : Any [0..*]`

```
{
  "type" : "object",
  "properties" : {
    "metadata" : {
      "type" : "array",
      "items" : {
        "$ref" : "http://www.opengis.net/tbd/Any.json"
      },
      "uniqueItems": true
    }
  }
}
```

The following values are all valid:

- `"metadata" : []`
- `"metadata" : [ true ]`
- `"metadata" : [ "abc", "def" ]`
- `"metadata" : [ { "abc" : "def" } ]`
- `"metadata" : [ { "abc" : [ 123, 456 ] } ]`

7.3.2.2.3. Record

The basic type Record is implemented in JSON by a JSON object with one property per field.

Listing 37 shows a generic JSON schema for Record instances with no defined record type. The schema of a Record with a record type is specified by a more specific JSON schema, see Record Type.

Listing 37. JSON Schema for Record

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/Record.json",
  "title": "A record value",
  "type": "object"
}
```

7.3.2.2.4. Record Type

RecordType is implemented in JSON as a JSON schema of type "object". The properties of the object are the fields. Example:

Listing 38. JSON Schema example for a record type for a temperature measurement

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.example.com/MyRecordType.json",
  "title": "My record value",
  "type" : "object",
  "properties" : {
    "timestamp" : {
      "type" : "string",
      "format" : "date-time"
    },
    "temperature_Cel" : {
      "type" : "number",
      "minimum" : -273.15
    }
  }
}
```

NOTE In this case "Cel" is used as the unit for degrees Celsius, using the symbol for this unit from [UCUM](https://ucum.org/ucum.html).

7.3.3. Spatial Types

The Features Core Profile includes:

- geometric primitives for dimensions 0d, 1d, 2d and 3d with linear (1d) or planar (2d) interpolation; and
- geometric aggregates of these primitives.

Overview lists these types (for both ISO 19107:2003 and ISO 19107:2019) and documents the constraints on these types.

Table 8. JSON Schema implementation for spatial types

Type in ISO 19107:2003	Interface in ISO 19107:2019	NAS equivalent	JSON Schema implementation reference
GM_Point	Point	GeometryPoint	Listing 39
GM_MultiPoint	Collection	MultiPointGeometry	Listing 42
GM_Curve	Line	GeometryCurve	Listing 43
GM_MultiCurve	Collection	MultiCurveGeometry	Listing 46
GM_Surface	Polygon	GeometrySurface	Listing 47
GM_MultiSurface	Collection	MultiSurfaceGeometry	Listing 48
GM_Solid	Solid	GeometrySolidRep (abstract)	Listing 49
GM_MultiSolid	Collection	MultiSolidGeometry (abstract)	Listing 51
GM_Object	Geometry	GeometryObjectRepresentation (abstract)	Listing 52

7.3.3.1. JSON Schema implementation

The JSON schemas in this Features Core Profile are specified so that point, line string and polygon geometries, including aggregates, continue to conform to GeoJSON.

In addition, polyhedra, including collections of polyhedra, are supported. A polyhedron is a solid that has closed polyhedral surfaces as exterior and interior boundaries.

NOTE The term "polyhedron" is used to reflect that only solids with closed polyhedral surfaces as boundaries are supported, just like line strings are a subset of all curves and polygons are a subset of all surfaces.

The JSON representations of the geometries included in the Features Core Profile extend the GeoJSON definitions with a property "crs" to specify the spatial coordinate reference system (CRS) of the geometry. The value of the "crs" property in the Features Core Profile is the URI of the CRS, consistent with "OGC API - Features - Part 2: Coordinate Reference Systems by Reference."

NOTE Extensions could be defined to also support an array of CRS URIs (e.g., for a compound CRS) or a PROJJSON object as a value of the property. Clients should be prepared to receive and handle "crs"-values that are not a single URI.

If the geometry is embedded in a context where a default CRS is stated, that default CRS is the CRS of the geometry, if no "crs" property is included in the geometry. If no default CRS is specified and no "crs" property is provided, the CRS is the GeoJSON default CRS, i.e., CRS84 or CRS84h depending on the coordinate dimension. This follows the current practice about the scope of a CRS declaration as it is, for example, used in GML 3.2.

For geometries with a coordinate dimension of one, a default CRS should be specified, too, but no decision was taken during the pilot. Two candidates are: [MSL height](http://www.opengis.net/def/

crs/EPSG/0/5714) or [EGM2008 height](http://www.opengis.net/def/crs/EPSG/0/3855).

The coordinate dimension is restricted to one, two or three. That is, no "M" dimension is supported. Surfaces require a coordinate dimension of at least two, solids require coordinate dimension three.

While it is a key principle to conform to GeoJSON, where possible, it is equally important to avoid that geometries that do not conform to GeoJSON are mistaken by tools as GeoJSON.

For example, geometries with a coordinate dimension of one are not in scope of GeoJSON, i.e., such geometries do not conform to GeoJSON. Using a value of "Point" for "type" (as used in GeoJSON) for such instances would create interoperability issues. Clients would need to use the Content-Type header (if the file is received as an HTTP response) or other information (file extension or information provided by the user) to disambiguate a GeoJSON geometry from a Features Core Profile JSON geometry that does not meet the GeoJSON requirements. It seems safer to restrict the use of "Point" etc. to geometries that conform to GeoJSON, but require the use of a different value, e.g., "ogc:Point", for other cases. The schemas and examples in this section use an "ogc" prefix in cases where a geometry does not conform to GeoJSON.

The JSON schemas specified in this section include a property "bbox", which has been introduced since the GeoJSON geometries include the option to add a [bounding box in addition to the geometry](https://tools.ietf.org/html/rfc7946#section-5). However, it is unclear how often such a property is used in practice and it could also be discussed to drop the property from the schemas.

NOTE

There are a number of constraints that cannot be expressed in JSON Schema:

- All coordinates have the same dimension; and
- The number of items in a "bbox" property is twice the coordinate dimension of the geometry.

If an application required that the JSON Schemas for geometry types be restricted to either 2D or 3D, but not both, then 2D and 3D specific JSON Schemas could be developed. Such schemas would be able to express the two constraints.

General pattern

Geometries are specified as nested arrays, where each array represents a geometry. The level of nesting for each geometry is shown in square brackets.

- *point* [1]: A coordinate array with a value for each axis of the CRS. The number of elements (one, two, or three) depends on the CRS.
- *multi-point* [2]: An array of *point* arrays. The order of the elements is not significant.
- *line string* [2]: An array of *point* arrays with a minimum of 2 elements. The order reflects the sequence of the points along the line string.
- *multi-line-string* [3]: An array of *line string* arrays. The order of the elements is not significant.
- *polygon* [3]: An non-empty array of *line string* arrays. Each *line string* is a ring and must be closed (the first and the last *point* are identical). The first ring is the exterior boundary all other rings are holes.

- *multi-polygon* [4]: An array of *polygon* arrays. The order of the elements is not significant.
- *polyhedron* [5]: An non-empty array of *multi-polygon* arrays. Each *multi-polygon* is a shell and must be closed. The first shell is the exterior boundary all other shells are holes.
- *multi-polyhedron* [6]: An array of *polyhedron* arrays. The order of the elements is not significant.

Point

A *point* is a coordinate array with a value for each axis of the CRS. The number of elements (one, two, or three) depends on the CRS.

Listing 39. JSON Schema for a point geometry

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/Point.json",
  "title": "A point geometry",
  "type": "object",
  "required": [
    "type",
    "coordinates"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "Point",
        "ogc:Point"
      ]
    },
    "crs": {
      "type": "string",
      "format": "uri"
    },
    "coordinates": {
      "type": "array",
      "minItems": 1,
      "maxItems": 3,
      "items": {
        "type": "number"
      }
    },
    "bbox": {
      "type": "array",
      "oneOf": [
        { "minItems": 2, "maxItems": 2 },
        { "minItems": 4, "maxItems": 4 },
        { "minItems": 6, "maxItems": 6 }
      ],
      "items": {
        "type": "number"
      }
    }
  }
}
```

Listing 40. Example of a 3D point geometry

```
{
  "type":"Point",
  "crs":"http://www.opengis.net/def/crs/OGC/0/CRS84h",
  "coordinates":[
     36.0964929,
     32.6020818,
     436.46
  ]
}
```

Listing 41. Example of a 1D point geometry

```
{
  "type":"ogc:Point",
  "crs":"http://www.opengis.net/def/crs/EPSG/0/5714",
  "coordinates":[
     436.34
  ]
}
```

MultiPoint

A *multi-point* is an array of *point* arrays. The order of the points is not significant.

Listing 42. JSON Schema for a multi-point geometry

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/MultiPoint.json",
  "title": "A multi-point geometry",
  "type": "object",
  "required": [
    "type",
    "coordinates"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "MultiPoint",
        "ogc:MultiPoint"
      ]
    },
    "crs": {
      "type": "string",
      "format": "uri"
    },
    "coordinates": {
      "type": "array",
      "items": {
        "type": "array",
        "minItems": 1,
        "maxItems": 3,
        "items": {
          "type": "number"
        }
      }
    },
    "bbox": {
      "type": "array",
      "oneOf": [
        { "minItems": 2, "maxItems": 2 },
        { "minItems": 4, "maxItems": 4 },
        { "minItems": 6, "maxItems": 6 }
      ],
      "items": {
        "type": "number"
      }
    }
  }
}
```

LineString

A *line string* is an array of *point* arrays, with a minimum of 2 elements. The order reflects the sequence of the points along the line string.

Listing 43. JSON Schema for a line string geometry

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/LineString.json",
  "title": "A line string geometry",
  "type": "object",
  "required": [
    "type",
    "coordinates"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "LineString",
        "ogc:LineString"
      ]
    },
    "crs": {
      "type": "string",
      "format": "uri"
    },
    "coordinates": {
      "type": "array",
      "minItems": 2,
      "items": {
        "type": "array",
        "minItems": 1,
        "maxItems": 3,
        "items": {
          "type": "number"
        }
      }
    },
    "bbox": {
      "type": "array",
      "oneOf": [
        { "minItems": 2, "maxItems": 2 },
        { "minItems": 4, "maxItems": 4 },
        { "minItems": 6, "maxItems": 6 }
      ],
      "items": {
        "type": "number"
      }
    }
  }
}
```

Listing 44. Example of a 3D line string geometry

```
{
   "type":"LineString",
   "crs":"http://www.opengis.net/def/crs/OGC/0/CRS84h",
   "coordinates":[
       [
          36.4251993,
          32.7137029,
          436.34
       ],
       [
          36.4270026,
          32.7114543,
          436.12
       ]
   ]
}
```

Listing 45. Example of a 1D line string geometry

```
{
  "type":"ogc:LineString",
  "crs":"http://www.opengis.net/def/crs/EPSG/0/5714",
  "coordinates":[
     [
       436.12
     ],
     [
       436.34
     ]
  ]
}
```

MultiLineString

A *multi-line-string* is an array of *line string* arrays. The order of the line strings is not significant.

Listing 46. JSON Schema for a multi-line-string geometry

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/MultiLineString.json",
  "title": "A multi-line-string geometry",
  "type": "object",
  "required": [
    "type",
    "coordinates"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "MultiLineString",
        "ogc:MultiLineString"
      ]
    },
    "crs": {
      "type": "string",
      "format": "uri"
    },
    "coordinates": {
      "type": "array",
      "items": {
        "type": "array",
        "minItems": 2,
        "items": {
          "type": "array",
          "minItems": 1,
          "maxItems": 3,
          "items": {
            "type": "number"
          }
        }
      }
    },
    "bbox": {
      "type": "array",
      "oneOf": [
        { "minItems": 2, "maxItems": 2 },
        { "minItems": 4, "maxItems": 4 },
        { "minItems": 6, "maxItems": 6 }
      ],
      "items": {
        "type": "number"
      }
    }
  }
}
```

Polygon

A *polygon* is an non-empty array of *line string* arrays. Each *line string* is a ring and must be closed (the first and the last *point* are identical). The first rings is the exterior boundary all other rings are holes.

Listing 47. JSON Schema for a polygon geometry

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/Polygon.json",
  "title": "A polygon geometry",
  "type": "object",
  "required": [
    "type",
    "coordinates"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "Polygon"
      ]
    },
    "crs": {
      "type": "string",
      "format": "uri"
    },
    "coordinates": {
      "type": "array",
      "minItems": 1,
      "items": {
        "type": "array",
        "minItems": 4,
        "items": {
          "type": "array",
          "minItems": 2,
          "maxItems": 3,
          "items": {
            "type": "number"
          }
        }
      }
    },
    "bbox": {
      "type": "array",
      "oneOf": [
        { "minItems": 4, "maxItems": 4 },
        { "minItems": 6, "maxItems": 6 }
      ],
      "items": {
        "type": "number"
      }
    }
  }
}
```

MultiPolygon

A *multi-polygon* is an array of *polygon* arrays. The order of the polygons is not significant.

Listing 48. JSON Schema for a multi-polygon geometry

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/MultiPolygon.json",
  "title": "A multi-polygon geometry",
  "type": "object",
  "required": [
    "type",
    "coordinates"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "MultiPolygon"
      ]
    },
    "crs": {
      "type": "string",
      "format": "uri"
    },
    "coordinates": {
      "type": "array",
      "items": {
        "type": "array",
        "minItems": 1,
        "items": {
          "type": "array",
          "minItems": 4,
          "items": {
            "type": "array",
            "minItems": 2,
            "maxItems": 3,
            "items": {
              "type": "number"
            }
          }
        }
      }
    },
    "bbox": {
      "type": "array",
      "oneOf": [
        { "minItems": 4, "maxItems": 4 },
        { "minItems": 6, "maxItems": 6 }
      ],
```

```
      "items": {
        "type": "number"
      }
    }
  }
}
```

Polyhedron

A *polyhedron* is an non-empty array of *multi-polygon* arrays. Each *multi-polygon* array is a shell and must be closed. The first shell is the exterior boundary, all other shells are holes.

Listing 49. JSON Schema for a polyhedron geometry

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/Polyhedron.json",
  "title": "A polyhedron geometry",
  "type": "object",
  "required": [
    "type",
    "coordinates"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "Polyhedron"
      ]
    },
    "crs": {
      "type": "string",
      "format": "uri"
    },
    "coordinates": {
      "type": "array",
      "minItems": 1,
      "items": {
        "type": "array",
        "minItems": 1,
        "items": {
          "type": "array",
          "minItems": 1,
          "items": {
            "type": "array",
            "minItems": 4,
            "items": {
              "type": "array",
              "minItems": 3,
              "maxItems": 3,
              "items": {
```

```
                "type": "number"
              }
            }
          }
        }
      },
      "bbox": {
        "type": "array",
        "minItems": 6,
        "maxItems": 6,
        "items": {
          "type": "number"
        }
      }
    }
}
```

Listing 50. Example of a polyhedron

```
{
   "type":"Polyhedron",
   "crs":"http://www.opengis.net/def/crs/OGC/0/CRS84h",
   "coordinates": [
      [
         [
            [
               [ 36.1005693, 32.6345205, 436.3],
               [ 36.1006509, 32.6345642, 436.3],
               [ 36.1006133, 32.6346141, 436.3],
               [ 36.1005316, 32.6345704, 436.3],
               [ 36.1005693, 32.6345205, 436.3]
            ]
         ]
      ],
      [
         [
            [
               [ 36.1005693, 32.6345205, 439.8 ],
               [ 36.1006509, 32.6345642, 439.8 ],
               [ 36.1006133, 32.6346141, 439.8 ],
               [ 36.1005316, 32.6345704, 439.8 ],
               [ 36.1005693, 32.6345205, 439.8 ]
            ]
         ]
      ],
      [
         [
            [
               [ 36.1005693, 32.6345205, 436.3],
               [ 36.1006509, 32.6345642, 436.3],
```

```
                [ 36.1006509, 32.6345642, 439.8 ],
                [ 36.1005693, 32.6345205, 439.8 ],
                [ 36.1005693, 32.6345205, 436.3]
            ]
        ],
        [
            [
                [ 36.1006509, 32.6345642, 436.3],
                [ 36.1006133, 32.6346141, 436.3],
                [ 36.1006133, 32.6346141, 439.8 ],
                [ 36.1006509, 32.6345642, 439.8 ],
                [ 36.1006509, 32.6345642, 436.3]
            ]
        ],
        [
            [
                [ 36.1006133, 32.6346141, 436.3],
                [ 36.1005316, 32.6345704, 436.3],
                [ 36.1005316, 32.6345704, 439.8 ],
                [ 36.1006133, 32.6346141, 439.8 ],
                [ 36.1006133, 32.6346141, 436.3]
            ]
        ],
        [
            [
                [ 36.1005316, 32.6345704, 436.3],
                [ 36.1005693, 32.6345205, 436.3],
                [ 36.1005693, 32.6345205, 439.8 ],
                [ 36.1005316, 32.6345704, 439.8 ],
                [ 36.1005316, 32.6345704, 436.3]
            ]
        ]
    ]
}
```

MultiPolyhedron

A *multi-polyhedron* is an array of *polyhedron* arrays. The order of the polyhedra is not significant.

Listing 51. JSON Schema for a multi-polyhedron geometry

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/MultiPolyhedron.json",
```

```
  "title": "A multi-polyhedron geometry",
  "type": "object",
  "required": [
    "type",
    "coordinates"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "MultiPolyhedron"
      ]
    },
    "crs": {
      "type": "string",
      "format": "uri"
    },
    "coordinates": {
      "type": "array",
      "items": {
        "type": "array",
        "minItems": 1,
        "items": {
          "type": "array",
          "minItems": 1,
          "items": {
            "type": "array",
            "minItems": 1,
            "items": {
              "type": "array",
              "minItems": 4,
              "items": {
                "type": "array",
                "minItems": 3,
                "maxItems": 3,
                "items": {
                  "type": "number"
                }
              }
            }
          }
        }
      }
    },
    "bbox": {
      "type": "array",
      "minItems": 6,
      "maxItems": 6,
      "items": {
        "type": "number"
      }
```

```
    }
  }
}
```

Geometry

Listing 52. JSON Schema for any geometry type covered by the Features Core Profile

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/Geometry.json",
  "title": "A Features Core Profile geometry",
  "type": "object",
  "oneOf": [
    {"$ref": "http://www.opengis.net/tbd/Point.json"},
    {"$ref": "http://www.opengis.net/tbd/MultiPoint.json"},
    {"$ref": "http://www.opengis.net/tbd/LineString.json"},
    {"$ref": "http://www.opengis.net/tbd/MultiLineString.json"},
    {"$ref": "http://www.opengis.net/tbd/Polygon.json"},
    {"$ref": "http://www.opengis.net/tbd/MultiPolygon.json"},
    {"$ref": "http://www.opengis.net/tbd/Polyhedron.json"},
    {"$ref": "http://www.opengis.net/tbd/MultiPolyhedron.json"}
  ]
}
```

7.3.4. Temporal Types

On the conceptual level, Date and DateTime are data types for literal values in the Gregorian calendar. These are basic types and their mapping has already been specified in Table 7.

TM_Instant is a temporal 0d geometry, TM_Period a temporal 1d geometry. Both are in general not restricted to the Gregorian calendar, but the Features Core Profile is restricted to temporal information in the Gregorian calendar. The JSON representations of TM_Instant and TM_Period for the Features Core Profile are, therefore, based on literal values based on the representations of Date and DateTime.

In addition, special values are supported.

- Unknown: Only supported for time intervals (unknown start or end). Represented as a blank/empty string (""). This is based on ISO 8601-2.
- Open: Only supported for time intervals (open start or end). Represented as a double-dot (".."). This is based on ISO 8601-2.
- Now: Supported for time instants and time intervals. Represented as "now". When processed, "now" should be resolved to the system date-time consistent with RFC 3339. Every subsequent use of the same temporal instant or interval in the same processing step (e.g., an API call) should use the same value.

Since the special values are not always needed, there are two JSON Schema variants, one restricted

to date/time values according to RFC 3339 and one with the additional special values. See Table 9.

Table 9. JSON Schema implementation for temporal types

Type	JSON Schema implementation reference
TM_Instant	Listing 53 or Listing 54
TM_Period	Listing 55 or Listing 56

Listing 53. JSON Schema for a temporal instant restricted to RFC 3339

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/InstantRfc3339.json",
  "title": "A temporal instant in the Gregorian calendar according to RFC 3339",
  "type": "string",
  "pattern": "^\\d{4}(-(0[1-9]|1[0-2])(-(0[1-9]|[12][0-9]|3[01])(T([01][0-9]|2[0-3]):([0-5][0-9]):([0-5][0-9]|60)(\\.[0-9]+)?(Z|(\\+|-)([01][0-9]|2[0-3]):([0-5][0-9])))?)?)?$"
}
```

This pattern allows the following values:

- year
- year, and month
- year, month, and day
- year, month, day, timestamp (fractional seconds are optional), and time zone

Additional constraints:

- the day must be valid for the month
- the seconds must be valid according to the rules for leap seconds

Listing 54 adds the special value "now".

Listing 54. JSON Schema for a temporal instant that also supports 'now'

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/InstantGregorian.json",
  "title": "A temporal instant in the Gregorian calendar according to RFC 3339 or 'now'",
  "type": "string",
  "pattern": "^\\d{4}(-(0[1-9]|1[0-2])(-(0[1-9]|[12][0-9]|3[01])(T([01][0-9]|2[0-3]):([0-5][0-9]):([0-5][0-9]|60)(\\.[0-9]+)?(Z|(\\+|-)([01][0-9]|2[0-3]):([0-5][0-9])))?)?)?|now$"
}
```

Listing 55. JSON Schema for a simple temporal interval in the Gregorian calendar, without special values

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/SimpleIntervalRFC3339.json",
  "title": "A temporal interval in the Gregorian calendar where start and end instant conform to RFC 3339",
  "type": "string",
  "pattern": "^\\d{4}(-(0[1-9]|1[0-2])(-(0[1-9]|[12][0-9]|3[01])(T([01][0-9]|2[0-3]):([0-5][0-9]):([0-5][0-9]|60)(\\.[0-9]+)?(Z|(\\+|-)([01][0-9]|2[0-3]):([0-5][0-9])))?)?)?\\/\\d{4}(-(0[1-9]|1[0-2])(-(0[1-9]|[12][0-9]|3[01])(T([01][0-9]|2[0-3]):([0-5][0-9]):([0-5][0-9]|60)(\\.[0-9]+)?(Z|(\\+|-)([01][0-9]|2[0-3]):([0-5][0-9])))?)?)?$"
}
```

Listing 56 adds the special value "now" for start/end dates that are now, ".." for open start/end dates and "" for unknown start/end dates.

Listing 56. JSON Schema for a simple temporal interval in the Gregorian calendar, with special values

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/SimpleIntervalGregorian.json",
  "title": "A temporal interval in the Gregorian calendar with support for start or end dates that are open, unknown or now",
  "type": "string",
  "pattern": "^(\\d{4}(-(0[1-9]|1[0-2])(-(0[1-9]|[12][0-9]|3[01])(T([01][0-9]|2[0-3]):([0-5][0-9]):([0-5][0-9]|60)(\\.[0-9]+)?(Z|(\\+|-)([01][0-9]|2[0-3]):([0-5][0-9])))?)?)?|(\\.\\.)?|now)\\/(\\d{4}(-(0[1-9]|1[0-2])(-(0[1-9]|[12][0-9]|3[01])(T([01][0-9]|2[0-3]):([0-5][0-9]):([0-5][0-9]|60)(\\.[0-9]+)?(Z|(\\+|-)([01][0-9]|2[0-3]):([0-5][0-9])))?)?)?|(\\.\\.)?|now)$"
}
```

Some examples:

Table 10. Examples of instants and intervals

Instants	
`2020`	the year 2020
`2020-07`	July 2020
`2020-07-22`	July 22, 2020
`2020-07-22T14:06:34+02:00`	2:06pm on July 22, 2020 in the CEST time zone (for example)
`now`	the current date/time
Intervals	
`2020-01-01/2020-12-31`	the year 2020 as a date interval

`2020-07-23T08:00:00-04:00/2020-07-23T09:00:00-04:00`	8am to 9am EDT (for example) on July 23, 2020
`2020-07-01/`	an interval starting on July 1, 2020; end is unknown
`2020-07-01/..`	an open interval starting on July 1, 2020
`2020-01-01T00:00:00Z/now`	an interval starting on January 1, 2020 until now
`2000/2010`	from 2000 until 2010

7.3.5. Features

This section specifies how the GeoJSON feature encoding can be extended to support the additional JSON schema capabilities of the Features Core Profile described above.

7.3.5.1. General remarks

By default, all properties are encoded in the "properties" object in the JSON feature object unless specified otherwise. See Nested Properties for more details.

7.3.5.2. AnyFeature

The JSON schema for the AnyFeature type from ISO 19109 in Listing 57 is based on the GeoJSON feature schema with the following changes based on the spatial types (see Spatial Types).

- A "crs" property has been added with the same semantics.
- The "bbox" property has been changed to express the constraint that the number of elements must be twice the coordinate dimension.
- The "geometry" property also supports (multi-)polyhedron geometries.

Listing 57. JSON Schema for AnyFeature

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/Feature.json",
  "title": "A feature",
  "type": "object",
  "required": [
    "type",
    "properties",
    "geometry"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "Feature",
        "ogc:Feature"
      ]
    },
    "id" : {
```

```
      "oneOf": [
        {
          "type": "string"
        },
        {
          "type": "integer"
        }
      ]
    },
    "properties": {
      "oneOf": [
        {
          "type": "null"
        },
        {
          "type": "object"
        }
      ]
    },
    "crs": {
      "type": "string",
      "format": "uri"
    },
    "geometry": {
      "oneOf": [
        {
          "type": "null"
        },
        {
          "$ref": "http://www.opengis.net/tbd/Point.json"
        },
        {
          "$ref": "http://www.opengis.net/tbd/MultiPoint.json"
        },
        {
          "$ref": "http://www.opengis.net/tbd/LineString.json"
        },
        {
          "$ref": "http://www.opengis.net/tbd/MultiLineString.json"
        },
        {
          "$ref": "http://www.opengis.net/tbd/Polygon.json"
        },
        {
          "$ref": "http://www.opengis.net/tbd/MultiPolygon.json"
        },
        {
          "$ref": "http://www.opengis.net/tbd/Polyhedron.json"
        },
        {
          "$ref": "http://www.opengis.net/tbd/MultiPolyhedron.json"
```

```
        }
      ]
    },
    "bbox": {
      "type": "array",
      "oneOf": [
        { "minItems": 2, "maxItems": 2 },
        { "minItems": 4, "maxItems": 4 },
        { "minItems": 6, "maxItems": 6 }
      ],
      "items": {
        "type": "number"
      }
    }
  }
}
```

7.3.5.3. Thematic attributes

Thematic properties are encoded in the "properties" object.

7.3.5.4. Spatial attributes

The General Feature Model in ISO 19109 supports multiple geometry properties per feature, while GeoJSON is restricted to a single spatial geometry.

If a feature type has a single geometry property, it is mapped to the top-level property "geometry" in the JSON feature object.

In case of multiple geometry properties in a feature type, one property (the default geometry) is mapped to the "geometry" property. The additional properties can be suppressed or included in the "properties" object like other feature properties. See also Default Geometry.

7.3.5.5. Temporal attributes

Temporal properties of the feature are encoded in the "properties" object.

7.3.5.6. Metadata or data quality attributes

Metadata or data quality properties are encoded in the "properties" object.

7.3.5.7. Association roles

If the value of a property is another feature / object, the UML-to-JSON-Schema encoding rule offers three options: always inline, always by-reference or either inline or by-reference.

The by-reference value may either be just a URI string or a link object, depending on the encoding preferences. The URI string is simpler to use for clients that prefer a simple structure, and would also lend itself better to a JSON to RDF conversion, but the link object has additional information (e.g., a title describing the link) that may be useful to clients.

The pilot has not specified, if all options should be supported in the JSON encoding of the Features Core Profile or if it should be more restrictive. Supporting all options leaves more flexibility, which in turn adds complexity to client implementations that have to be prepared for all options. Finding the sweet spot should be subject to testing in implementations and broader discussion. Note that there is a related discussion in the OGC API Records SWG in the context of querying the records (issue 28 [https://github.com/opengeospatial/ogcapi-records/issues/28#issuecomment-628430946]).

Listing 58. JSON Schema for an object that is referenced using a URI

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/ByReference.json",
  "title": "A reference encoded as a URI",
  "type": "string",
  "format": "uri"
}
```

Listing 59. JSON Schema for an object that referenced using a link object

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/Link.json",
  "title": "A reference encoded as a web link according to RFC 8288",
  "type": "object",
  "required": [
    "href",
    "rel"
  ],
  "properties": {
    "href": {
      "type": "string",
      "format": "uri"
    },
    "rel": {
      "type": "string"
    },
    "type": {
      "type": "string"
    },
    "hreflang": {
      "type": "string"
    },
    "title": {
      "type": "string"
    },
    "length": {
      "type": "integer"
    }
  }
}
```

Listing 60. JSON Schema for an object that is encoded inline

```
{
  "$ref" : "{$id of the schema of the value type}"
}
```

Listing 61. JSON Schema for an object that is encoded inline or by reference (using a URI)

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "title": "Example of and association role value that is encoded inline or by reference (using a URI)",
  "oneOf": [
    {
      "type": "string",
      "format": "uri"
    }, {
      "$ref" : "{$id of the schema of the value type}"
    }
  ]
}
```

Listing 62. JSON Schema for an object that is encoded inline or by reference (using a link object)

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "title": "Example of and association role value that is encoded inline or by reference (using a link object)",
  "oneOf": [
    {
      "$ref" : "http://www.opengis.net/tbd/Link.json"
    }, {
      "$ref" : "{$id of the schema of the value type}"
    }
  ]
}
```

7.3.5.8. Feature collections

Feature collections are not specified in ISO 19109 and they are, therefore, not part of the Features Core Profile. However, for sharing feature data in JSON, feature collections are important and need to be included at least in the JSON representation of the Features Core Profile.

Like with the feature schema, the GeoJSON schema for a feature collection is used as the basis and extended / amended.

Listing 63. JSON Schema for a feature collection

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/FeatureCollection.json",
  "title": "A feature collection",
  "type": "object",
  "required": [
    "type",
    "features"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "FeatureCollection",
        "ogc:FeatureCollection"
      ]
    },
    "crs": {
      "type": "string",
      "format": "uri"
    },
    "type": {
      "type": "array",
      "items": {
        "$ref" : "http://www.opengis.net/tbd/Feature.json"
      }
    },
    "bbox": {
      "type": "array",
      "oneOf": [
        { "minItems": 2, "maxItems": 2 },
        { "minItems": 4, "maxItems": 4 },
        { "minItems": 6, "maxItems": 6 }
      ],
      "items": {
        "type": "number"
      }
    }
  }
}
```

7.3.5.9. Example

The schema for a particular feature type, a subtype of AnyFeature, would then be encoded as a combination of the AnyFeature schema and the details for the specific feature type using "allOf". The example Listing 64 illustrates this and constrains the feature to have a solid geometry plus it defines four known properties.

Listing 64. Example of a JSON Schema for a feature type "building"

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.example.com/some/path/Building.json",
  "title": "A building",
  "allOf": [
    {
      "$ref": "http://www.opengis.net/tbd/Feature.json"
    }, {
      "type": "object",
      "properties": {
        "geometry": {
          "$ref": "http://www.opengis.net/tbd/Polyhedron.json"
        },
        "properties": {
          "type": "object",
          "properties": {
            "function" : {
              "type" : "string",
              "enum" : [ "residential", "public use", "commercial", "other" ]
            },
            "floors" : {
              "type" : "integer"
            },
            "parcel" : {
              "type": "string",
              "format": "uri"
            },
            "lastUpdate" : {
              "type" : "string",
              "format" : "date-time"
            }
          }
        }
      }
    }
  ]
}
```

7.4. Extensions

This section documents extensions of the Features Core Profile, which, if standardized, should be in separate conformance classes. These extensions can serve as starting points for the development of additional extensions.

7.4.1. Spatio-temporal geometries

The new edition of ISO 19111:2019 supports spatio-temporal coordinate referencing, which enables

the use of, for example, the Point or Line types from ISO 19107:2019 with spatio-temporal coordinates (spatio-temporal points and trajectories).

In the JSON encoding, two CRSs could be specified (spatial and temporal) and coordinates could consist of numbers (spatial coordinates) or RFC 3339 values (temporal coordinates).

Listing 65. JSON Schema for a point geometry with spatio-temporal coordinates

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/PointST.json",
  "title": "A spatio-temporal point geometry",
  "type": "object",
  "required": [
    "type",
    "coordinates"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "ogc:PointST"
      ]
    },
    "crs": {
      "oneOf" : [ {
        "type": "string",
        "format": "uri"
      }, {
        "type": "array",
        "minItems": 2,
        "maxItems": 2,
        "items": {
          "type": "string",
          "format": "uri"
        }
      } ]
    },
    "coordinates": {
      "type": "array",
      "minItems": 2,
      "maxItems": 4,
      "items": {
        "oneOf": [
          {
            "type": "number"
          }, {
            "type": "string"
          }
        ]
      }
```

```
    },
    "bbox": {
      "type": "array",
      "oneOf": [
        { "minItems": 4, "maxItems": 4 },
        { "minItems": 6, "maxItems": 6 },
        { "minItems": 8, "maxItems": 8 }
      ],
      "items": {
        "oneOf": [
          {
            "type": "number"
          }, {
            "type": "string"
          }
        ]
      }
    }
  }
}
```

Listing 66. Example: an observation with a spatio-temporal point geometry

```
{
  "type" : "Feature",
  "geometry" : {
    "type" : "ogc:PointST",
    "crs" : [ "http://www.opengis.net/def/crs/OGC/0/CRS84h",
"http://www.opengis.net/def/uom/ISO-8601/0/Gregorian" ],
    "coordinates" : [ 6.9299617, 50.000008, 105.2, "2019-07-01T10:00:00Z" ]
  },
  "properties" : {
    "observedProperty" : "temperature",
    "result" : 22.3,
    "result_uom" : "Cel"
  }
}
```

Listing 67. JSON Schema for a line geometry with spatio-temporal coordinates (trajectory)

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/LineStringST.json",
  "title": "A spatio-temporal line string geometry",
  "type": "object",
  "required": [
    "type",
    "coordinates"
  ],
  "properties": {
```

```
    "type": {
      "type": "string",
      "enum": [
        "ogc:LineStringST"
      ]
    },
    "crs": {
      "oneOf" : [ {
        "type": "string",
        "format": "uri"
      }, {
        "type": "array",
        "minItems": 2,
        "maxItems": 2,
        "items": {
          "type": "string",
          "format": "uri"
        }
      } ]
    },
    "coordinates": {
      "type": "array",
      "items": {
        "type": "array",
        "minItems": 2,
        "maxItems": 4,
        "items": {
          "oneOf": [
            {
              "type": "number"
            }, {
              "type": "string"
            }
          ]
        }
      }
    },
    "bbox": {
      "type": "array",
      "oneOf": [
        { "minItems": 4, "maxItems": 4 },
        { "minItems": 6, "maxItems": 6 },
        { "minItems": 8, "maxItems": 8 }
      ],
      "items": {
        "oneOf": [
          {
            "type": "number"
          }, {
            "type": "string"
          }
```

```
        ]
      }
    }
  }
}
```

Listing 68. Example: the trajectory of a tornado

```
{
  "type" : "Feature",
  "geometry" : {
    "type" : "ogc:LineStringST",
    "crs" : [ "http://www.opengis.net/def/crs/OGC/1.3/CRS84",
"http://www.opengis.net/def/uom/ISO-8601/0/Gregorian" ],
    "coordinates" : [[ 6.9299617, 50.000008, "2019-07-01T10:07:00Z" ],
                     [ 6.9546373, 49.998403, "2019-07-01T10:08:00Z" ],
                     [ 6.9834923, 50.004252, "2019-07-01T10:09:00Z" ],
                     [ 7.0134923, 50.015566, "2019-07-01T10:10:00Z" ],
                     [ 7.0285356, 50.025566, "2019-07-01T10:11:00Z" ],
                     [ 7.0354923, 50.036248, "2019-07-01T10:12:00Z" ]]
  },
  "properties" : {
    "maximumWindSpeed_kph" : "126.5"
  }
}
```

Note that OGC has recently published a standard OGC Moving Features Encoding Extension - JSON (MF-JSON) [https://docs.opengeospatial.org/is/19-045r3/19-045r3.html] that includes a GeoJSON extension for trajectories that follows a different approach. MF-JSON adds the time series as a property `datetimes` in the `properties` object with an array of the time steps in the time series. The geometry of the feature is a line string and the number of positions in the line string must match the number of time steps. The extension above intentionally follows a different approach to explore how a spatio-temporal geometry extension could look like.

7.4.2. Non-linear Geometries

Curves with non-linear curve segments are important for the NAS, in particular, arcs, circles, elliptic arcs and ellipses as defined in ISO 19107 (old and new edition). Probably the best approach would be to define a new geometry type for each new curve type. For example, for (circular) arcs this could be defined using three positions on the arc (start, intermediate, and end position).

Listing 69. An arc

```
{
  "type" : "Feature",
  "geometry" : {
    "type" : "ogc:Arc",
    "coordinates" : [[ 7.0134923, 50.001248 ], [ 7.0234923, 50.021248 ], [ 7.0354923, 50.033248 ]]
  },
  "properties" : {
    "attribute" : "value"
  }
}
```

Similar definitions could be specified for other curve types using the CurveData data type from ISO 19107:2019 and its subtypes (the CurveData types "contain copies of the attributes needed for every Curve construction").

This would also allow to define a general curve type that consists of the connected sequence of curves. For example:

Listing 70. Example: A refueling track (a "race track" geometry)

```
{
  "type" : "Feature",
  "geometry" : {
    "type" : "ogc:Curve",
    "crs" : "http://www.opengis.net/def/crs/OGC/1.3/CRS84",
    "segments" : [ {
      "type" : "LineString",
      "coordinates" : [[ 6.9299617, 50.000008 ], [ 7.0134923, 50.000008 ]]
    }, {
      "type" : "ogc:Arc",
      "coordinates" : [[ 7.0134923, 50.000008 ], [ 7.0294923, 50.010008 ], [ 7.0134923, 50.020008 ]]
    }, {
      "type" : "LineString",
      "coordinates" : [[ 7.0134923, 50.020008 ], [ 6.9299617, 50.020008 ]]
    }, {
      "type" : "ogc:Arc",
      "coordinates" : [[ 6.9299617, 50.020008 ], [ 6.9139617, 50.010008 ], [ 6.9299617, 50.000008 ]]
    } ]
  },
  "properties" : {
    "altitude_ft" : "12000"
  }
}
```

7.4.3. Temporal geometries in another temporal coordinate reference system

In the Features Core Profile, all time instants or time intervals are in the (proleptic) Gregorian calendar, which supports a JSON representation as a literal value using the ISO 8601 encoding. To support other temporal coordinate reference systems, an object representation as a (temporal) geometry is required. The general pattern for (spatio-temporal) geometry objects can be used.

There are two options. The first is to simply reuse the spatial geometry types, just with only a temporal CRS. For example:

Listing 71. Time instant as a temporal geometry with explicit temporal coordinate reference system, option 1

```
{
  "type" : "ogc:Point",
  "crs" : "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian",
  "coordinates" : [ "2019-07-01T10:34:00Z" ]
}
```

The next example is a time interval in the POSIX system used in UNIX (seconds since epoch). In this case, the interval is "2017-01-01T00:00:15Z/2017-01-01T07:46:39Z".

Listing 72. Time interval as a temporal geometry with explicit temporal coordinate reference system, option 1

```
{
  "type" : "ogc:LineString",
  "crs" : "http://www.example.com/def/crs/posix",
  "coordinates" : [ [ 1483228815 ], [ 1483299999 ] ]
}
```

Alternatively, a more specific encoding with explicit types for time instants and intervals could be used:

Listing 73. Time instant as a temporal geometry with explicit temporal coordinate reference system, option 2

```
{
  "type" : "ogc:Instant",
  "crs" : "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian",
  "coordinates" : [ "2019-07-01T10:34:00Z" ]
}
```

Listing 74. Time interval as a temporal geometry with explicit temporal coordinate reference system, option 2

```
{
  "type" : "ogc:Interval",
  "crs" : "http://www.example.com/def/crs/posix",
  "coordinates" : [ 1483228815, 1483299999 ]
}
```

Whether explicit types would be beneficial requires analysis and testing. One benefit could be easier mapping to RDF in JSON-LD contexts.

Chapter 8. Using SHACL for Validation of Linked Data

8.1. Overview

One of the tasks in the OGC UGAS-2020 Pilot project was to analyze the potential use of the Shapes Constraint Language (SHACL) to validate linked data.

Previous work related to this topic by the OGC Interoperability Program includes, but is not limited to the following.

- The OGC Testbed-12 ShapeChange Engineering Report [http://docs.opengeospatial.org/per/16-020.html] documents rules for converting a UML-based application schema to OWL ontologies.
- The OGC Testbed-14: Application Schema-based Ontology Development Engineering Report [http://docs.opengeospatial.org/per/18-032r2.html] complements the work performed in OGC Testbed-12, in that it defines additional rules for converting application schema elements to OWL ontologies. These rules cover the topics of property enrichment and property generalization. Furthermore, an analysis of how OCL constraints can be translated to OWL expressions was conducted. That analysis resulted in detailed conversion instructions for most OCL language constructs that were being investigated, but also revealed that some of these language constructs cannot be translated to OWL.
- The OGC Testbed-14: Characterization of RDF Application Profiles for Simple Linked Data Application and Complex Analytic Applications Engineering Report [http://docs.opengeospatial.org/per/18-094r1.html] worked on defining a concept of application profiles for ontologies, using OWL and SHACL.

All of these reports identified that future work should include an analysis of how UML-based application schemas, including OCL constraints defined in such schemas, can be converted to SHACL, as a means to perform validation of linked data. These work items have been addressed by the UGAS-2020 Pilot. This chapter documents the results of the according analysis.

The remaining sections of this chapter are structured as follows: The Validation of Linked Data section explains what validation of Linked Data actually means, when considering ontologies and SHACL as validation technologies. The Shapes Constraint Language (SHACL) section gives an introduction to SHACL, explaining key concepts and SHACL language constructs. The SHACL Conversion Rules section defines rules for converting a UML-based application schema, including OCL constraints, to SHACL validation constructs. Finally, a Summary of the results documented in this chapter is provided.

NOTE

Within this chapter, the following namespace prefix bindings are used:

Table 11. Prefix bindings used in the SHACL chapter

Prefix	Namespace
dash	http://datashapes.org/dash#
ex	http://example.org/shacl/data#
exshacl	http://example.org/shacl/shapes#
exschema	http://example.org/shacl/schema/
geosparql	http://www.opengis.net/ont/geosparql#
owl	http://www.w3.org/2002/07/owl#
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
sc	http://www.interactive-instruments.de/ShapeChange/Configuration/1.1
sh	http://www.w3.org/ns/shacl#
skos	http://www.w3.org/2004/02/skos/core#
xsd	http://www.w3.org/2001/XMLSchema#

8.2. Validation of Linked Data

In order to understand how SHACL can be used for validation of *Linked Data*, it is crucial to achieve a common understanding of what *Linked Data* and validation of such data actually means.

The W3C provides a useful introduction to *Linked Data* on https://www.w3.org/standards/semanticweb/data. That page explains the term *Linked Data* in more detail, and how it relates to *Semantic Web*, *Vocabularies*, *Queries*, *Inference*, and applications. Quoting directly from that page:

> What is Linked Data?
>
> The Semantic Web [https://www.w3.org/standards/semanticweb/] is a Web of Data — of dates and titles and part numbers and chemical properties and any other data one might conceive of. The collection of Semantic Web technologies (RDF, OWL, SKOS, SPARQL, etc.) provides an environment where applications can query [https://www.w3.org/standards/semanticweb/query] that data, draw inferences [https://www.w3.org/standards/semanticweb/inference] using vocabularies [https://www.w3.org/standards/semanticweb/ontology], etc.
>
> However, to make the Web of Data a reality, it is important to have the huge amount of data on the Web available in a standard format, reachable and manageable by Semantic Web tools. Furthermore, not only does the Semantic Web need access to data, but relationships among data should be made available, too, to create a Web of Data (as opposed to a sheer collection of datasets). This collection of interrelated datasets on the Web can also be referred to as Linked Data.
>
> To achieve and create Linked Data, technologies should be available for a common format (RDF), to make either conversion or on-the-fly access to existing databases (relational, XML, HTML, etc). It is also important to be able to setup query [https://www.w3.org/standards/semanticweb/query] endpoints to access that data more conveniently. W3C provides a palette of technologies (RDF, GRDDL, POWDER, RDFa, the upcoming R2RML, RIF, SPARQL) to get access to the data.
>
> What is Linked Data Used For?
>
> Linked Data lies at the heart of what Semantic Web is all about: large scale integration of, and reasoning on, data on the Web. [...]
>
> — Linked Data [https://www.w3.org/standards/semanticweb/data]

A more terse definition of *Linked Data* is given on the W3C Semantic Web Wiki:

> Linked Data is the data format that supports the Semantic Web. The basic rules for Linked Data are defined as:
>
> - Use Uniform Resource Identifiers (URIs) to identify things;
> - Use HTTP URIs so that these things can be referred to and looked up ("dereferenced") by people and user agents;
> - Provide useful information about the thing when its URI is dereferenced, using standard formats such as RDF/XML; and
> - Include links to other, related URIs in the exposed data to improve discovery of other related information on the Web.
>
> — W3C Semantic Web Wiki [https://www.w3.org/2001/sw/wiki/Semantic_Web_terminology#linked_data]

For the purpose of this ER, the primary format for encoding linked data is the Resource Description Framework (RDF).

NOTE	For an introduction to RDF, read the RDF 1.1 Primer [https://www.w3.org/TR/rdf11-primer/], which has links to the normative W3C specifications.

Another format for encoding linked data is JSON-LD [https://www.w3.org/TR/json-ld11/]. JSON-LD is a concrete RDF syntax, meaning that most JSON-LD documents can be interpreted as RDF.

NOTE	The JSON-LD extensions to the RDF data model are documented in the chapter Relationship to RDF [https://www.w3.org/TR/json-ld11/#relationship-to-rdf] of the JSON-LD 1.1 specification.

Using so-called JSON-LD context documents, JSON data can be interpreted as JSON-LD, and thus typically also as RDF.

NOTE	OGC Testbed-14 conducted an analysis on how the semantics of JSON data can be defined using JSON-LD. More specifically, it was investigated if and how GeoJSON data could be transformed to NEO RDF data, using JSON-LD context documents. The results of that analysis are documented in the OGC Testbed-14: Application Schemas and JSON Technologies Engineering Report [http://docs.opengeospatial.org/per/18-091r2.html].

For the purpose of validating linked data, RDF data - where resources are identified by HTTP URIs - can be regarded as linked data. The RDF data may have been created in a number of ways. For example, it could have been created by transforming JSON data to RDF using JSON-LD context documents. Another example is that a web service provides RDF data from some data store, such as a relational database, or even another web service.

Validation of linked data can be done in two different ways.

- Checking the logical consistency of the data, based upon a set of vocabularies [https://www.w3.org/

standards/semanticweb/ontology] that are used in the linked data. Formats to define such vocabularies are, for example, RDF Schema (RDFS) [https://www.w3.org/TR/rdf-schema/] and the Web Ontology Language (OWL) [https://www.w3.org/TR/owl-primer/]. These vocabularies are typically used to infer new information. A so-called *reasoner* is used to perform the according inferencing. Such a reasoner will also detect if the given RDF data is inconsistent to some vocabulary definition(s). In that case, inferences would not be produced. However, that the RDF data is inconsistent is a useful result. RDF data that is known to be inconsistent against its vocabularies can also be interpreted as being invalid.

- Checking that RDF data satisfies a set of structural constraints. This is where SHACL comes into play.

The following example shows how validation of linked data can be performed using OWL and SHACL. It also highlights a number of Semantic Web concepts, such as open-world assumption (OWA), closed-world assumption (CWA), and inferencing.

Consider the following RDF data, written in the Turtle syntax [https://www.w3.org/TR/turtle/].

```
<http://example.org/shacl/schema>
  a owl:Ontology .

exschema:Building
  a owl:Class ;
  rdfs:subClassOf [
      a owl:Restriction ;
      owl:cardinality "1"^^xsd:nonNegativeInteger ;
      owl:onProperty exschema:numberOfStorys ;
    ] .

exschema:numberOfStorys
  a owl:DatatypeProperty ;
  rdfs:domain exschema:Building ;
  rdfs:range xsd:nonNegativeInteger .
```

The data shows an OWL ontology that defines a class `exschema:Building` and a property `exschema:numberOfStorys`. OWL and RDFS language constructs are used to define:

- that a `exschema:Building` has exactly one value for property `exschema:numberOfStorys`; and
- that the property `exschema:numberOfStorys` is used by resources of type `exschema:Building` and generally has a non-negative integer value.

Given the following RDF data:

```
ex:A exschema:numberOfStorys 0 .

ex:B a exschema:Building .

ex:C
  a exschema:Building ;
  exschema:numberOfStorys -1 .
```

An OWL reasoner will conclude that the data is inconsistent, because resource `ex:C` is defined to be of type `exschema:Building` but it has a negative integer value for property `exschema:numberOfStorys`.

If we correct `ex:C` and change the data as follows:

```
ex:A
  exschema:numberOfStorys 0 .

ex:B a exschema:Building .

ex:C
  a exschema:Building ;
  exschema:numberOfStorys 2 .
```

And let the reasoner run over the updated data, then no inconsistency is reported. This may be surprising, because `ex:B` is declared to be a `exschema:Building`, but it has no `exschema:numberOfStorys`. The explanation for this behavior is that an OWL application works under the so-called *open-world assumption* (OWA), where facts not declared in the data are simply assumed to be unknown at present. That is different to an application with so-called *closed-world assumption* (CWA) - examples being SQL databases - where the available data is assumed to be complete, and anything not contained in the data is false. `ex:B` is not marked as inconsistent by the reasoner because it could be the case that the `exschema:numberOfStorys` is simply not yet known rather than being erroneously missing. In order to ensure that `ex:B` has a value for the number of storys, SHACL can be used.

First, however, another important feature of OWL applications should be discussed, and that is inferencing. An OWL reasoner can not only detect inconsistencies in the data. As mentioned before, it can also produce new facts, based upon the axioms and expressions defined in the vocabularies, and given some RDF data.

NOTE

The OWL 2 Web Ontology Language Primer [https://www.w3.org/TR/owl-primer/], chapter Modeling Knowledge: Basic Notions [https://www.w3.org/TR/owl-primer/#Modeling_Knowledge:_Basic_Notions], explains axioms and expressions in the context of OWL.

In the example, one of the facts produced by inferencing is that the resource `ex:A` also is a `exschema:Building`. That is because the ontology defines that an instance of `exschema:Building` is any resource that has exactly one (non-negative integer) value for `exschema:numberOfStorys` - which is the case for `ex:A`. RDFS and OWL are well suited for use cases where resources need to be classified.

Healthcare is one application domain: a classification tool could help diagnose a disease based upon observed symptons, and given an OWL ontology that defines diseases with certain values for such symptoms. In much the same way, a classification tool could generate geospatial intelligence.

NOTE	Advanced features of SHACL, more specificall *SHACL Rules*, also support generating new facts. However, that feature of SHACL is not discussed further in this ER.

Coming back to the example, how can the data be checked to ensure that all buildings have a meaningful value for the property `exschema:numberOfStorys`? That is where SHACL comes into play. Consider the following RDF data:

```
exshacl:BuildingShape
  a sh:NodeShape ;
  sh:property [
      sh:path exschema:numberOfStorys ;
      sh:datatype xsd:integer ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:minInclusive 1 ;
    ] ;
  sh:targetClass exschema:Building .
```

A SHACL shape for all resources that are of type `exschema:Building` is defined, with constraints on property `exschema:numberOfStorys` to ensure that each such resource has exactly one integer value for the property, and that that value is greater than or equal to 1.

NOTE	In the definition of property `exschema:numberOfStorys`, the range has been set to xsd:nonNegativeInteger on purpose, to demonstrate validation with SHACL. If the range had been defined as xsd:positiveInteger, a meaningful value would already have been achieved, because xsd:positiveInteger excludes 0, whereas xsd:nonNegativeInteger includes 0.

When a SHACL processor evaluates the shape against the corrected RDF data, `ex:B` will be marked as invalid - because it is declared to be of type `exschema:Building` but does not have a value for property `exschema:numberOfStorys`, and SHACL does not operate under the open-world assumption.

The reason why `ex:A` is not flagged as being invalid is that the RDF data does not explicitly declare `ex:A` to be of type `exschema:Building`. Since a SHACL processor by default does not perform any inferencing, it does not recognise `ex:A` as being a resource to which the SHACL shape should be applied. However, as explained before, inferencing can produce the missing fact. If a SHACL processor also has the inferred information that `ex:A` indeed is of type `exschema:Building`, it would apply the SHACL shape to that resource and recognize that it is invalid (since the value of property `exschema:numberOfStorys` is 0).

When SHACL validation identifies invalid structures in a set of linked data, these validation results can be used to correct the data, and to complement missing information. That can be useful for subsequent reasoning on the data, because of the following.

- A reasoner cannot generate new facts if an inconsistency has been detected in the data. The cause of an inconsistency may be complex, and hard for a reasoner to report and for a user to pinpoint exactly. If invalid data structures are causing the inconsistency, then SHACL validation can be a way to identify them. Subsequent correction of the data may then remove the cause of the inconsistency, allowing the reasoner to perform inferencing and generate new facts.
- When SHACL validation reveals that some information that is typically expected for a certain type of resource, such as the `exschema:numberOfStorys`, is missing, then a process can be started to provide that information. With the additional information available in the linked data, a reasoner may be able to infer new facts.

These are examples of how SHACL based validation can be used to support inferencing, and derivation of new information/intelligence.

8.3. Shapes Constraint Language (SHACL)

SHACL is a language for describing and validating RDF graphs.

A number of SHACL-related specifications exist.

- Shapes Constraint Language (SHACL) [https://www.w3.org/TR/shacl/]: This W3C Recommendation defines SHACL Core and the extension SHACL SPARQL. SHACL Core defines *shapes*, which define *constraints* that apply to RDF data nodes that are identified by *targets*.
- SHACL Advanced Features [https://www.w3.org/TR/shacl-af/]: This W3C Working Group Note defines custom SHACL targets, annotation properties, user-defined functions, node expressions and rules. Many of these features are realized using SPARQL, but the specification allows realization with other implementations languages as well, for example JavaScript.
- SHACL JavaScript Extensions [https://www.w3.org/TR/shacl-js/]: This W3C Working Group Note defines a JavaScript-based extension mechanism for SHACL.
- DASH Data Shapes Vocabulary [http://datashapes.org/dash]: This document extends the range of SHACL constraints and target types, and defines a number of additional SHACL extensions.

For the purpose of validating linked data using SHACL, with validation instructions derived from application schemas in UML, SHACL Core is of primary interest. SHACL Core defines *shapes*, which are the main elements used for validating RDF data. A *shape* defines a set of *constraints* that apply to a set of RDF nodes. These nodes are identified by the shape using *target* predicates. A SHACL processor applies a set of *shapes* - which is called the *shapes graph* - to an RDF dataset - the *data graph*.

An important aspect of SHACL is that it uses RDF and RDFS vocabularies. By default, a SHACL processor evaluates `rdf:type` and `rdfs:subClassOf` predicates, provided that RDF triples with these predicates are made known to the processor as part of the data graph. For example, if an RDF resource is of type T1 (defined via an `rdf:type` predicate), the SHACL processor will also know that that resource has `rdf:type` T2 and T3, if it has found the following triples in the data graph: `T1 rdfs:subClassOf T2` and `T2 rdfs:subClassOf T3`. It is not necessary that the data graph contains the triples `T1 rdf:type T2` and `T1 rdf:type T3` - these triples will automatically be inferred by the SHACL processor. That is useful whenever the *target* of a *shape* are resources of a particular type T (e.g., T3 in the preceding example), since the SHACL processor will automatically apply the *shape* to all RDF

resources whose declared type is a subtype of type T (e.g., T1 and T2). The same is true for checking the value types of properties - which is something that JSON Schema v2019-09, for example, does not support (for further details, see section Class Specialization and Property Ranges in the JSON Schema chapter).

Note that a SHACL processor does not automatically perform full RDFS based inferencing, nor any other kind of inferencing. For example, SHACL does not consider owl:sameAs predicates, which state (to an OWL reasoner) that two resources are actually the same thing. However, SHACL has a way to instruct a SHACL processor to apply certain kinds of inferencing, as mentioned before.

The following subsections describe key SHACL concepts in further detail, so that the reader of this chapter can get a better understanding of how SHACL features can be used in general, and which SHACL features are especially of interest for validation of linked data, and the definition of conversion rules for SHACL.

8.3.1. Shapes

A SHACL shape defines how a so-called focus node shall be validated. Typically, the shape defines a set of SHACL constraints that will be evaluated against the focus node. Validation of a shape with multiple constraints uses an implicit *and*, i.e., all constraints defined in a shape must be satisfied by the focus node.

SHACL Core defines two types of shapes:

- node shapes - which define the shape of the focus node itself; and
- property shapes - which define the shape of the values of a particular property (or set of properties identified by a property path).

8.3.1.1. Focus Nodes

An RDF term - an IRI, a literal, or a blank node - that is validated by a SHACL processor against a shape is called a *focus node*.

8.3.1.2. Targets

SHACL provides a number of predicates to identify the targets - i.e., the focus nodes - of a shape. A SHACL processor will apply a shape to all RDF terms (IRI, literal, blank node) that match the target of the shape.

NOTE	Using shape-based constraint components, the focus node is determined by evaluating that component. In other words, the target of the shape that is defined as object of a shape-based constraint component is implied by that component. For example, if `Shape1 sh:node Shape2` then the focus node of Shape1 is the target of Shape2.

SHACL Core defines the following target predicates.

- `sh:targetClass` - targets RDF nodes that are instances of a given RDFS/OWL class, either directly or as subclasses

    - Note that SHACL automatically recognizes `rdfs:subClassOf` relationships that are defined in the data graph (as explained in more detail here).
    - If an RDFS/OWL class C also is a `sh:NodeShape` then the target declaration for that node shape is implicitly defined as `sh:targetClass C`.
- `sh:targetNode` - targets individual RDF nodes (identified by IRI) and literals.
- `sh:targetSubjectOf` - targets RDF nodes that are subjects of a given predicate.
- `sh:targetObjectOf` - targets RDF nodes (identified by IRI) and literals that are objects of a given predicate.

Of the given target predicates, the first one is of primary interest for defining SHACL conversion rules.

8.3.1.3. Declaring Messages for a Shape

The predicate `sh:message` can be used to define messages that shall be reported if validation for a shape failed. The value of `sh:message` is a string or a language tagged string (multiple `sh:message` values in the same shape should have different language tags).

This predicate could be useful when translating OCL constraints.

8.3.1.4. Deactivating a Shape

By setting the predicate `sh:deactivated` to the boolean value true, a shape can be deactivated, meaning that a SHACL processor will not apply it. This can be useful for debugging, but also for deactivating certain shapes from imported shapes graphs.

8.3.2. Node Shapes

A node shape is a shape that applies to the focus node itself. A node shape cannot have the `sh:path` predicate.

8.3.3. Property Shapes

A property shape is a shape that applies to the values of the properties that can be reached from the focus node, by following the SHACL property path defined by the `sh:path` predicate.

8.3.3.1. SHACL Property Paths

A property path defines how certain properties within the data graph can be selected, starting from the focus node. The following kinds of paths are available:

- Predicate path: identifies a single RDF/OWL property by its IRI
    - Example: `ex:parent` - will select the parents of the focus node
- Sequence path: a list of two or more property paths, typically predicate paths
    - Example: `( ex:parent ex:firstName )` - will select the first names of the parents of the focus node
- Alternative path: a list of two or more alternative property paths

- Example: `[ sh:alternativePath ( ex:father ex:mother ex:sister ex:brother ) ]` - will select the father, the mother, as well as sisters and brothers of the focus node

- Inverse path: navigates to the subject of the RDF/OWL property that is identified by its IRI
  - Example: `[ sh:inversePath ex:parent ]` - will select the children of the parents of the focus node (thus including the focus node itself)
- Zero-or-more path: a path that connects the subject and object of the path by zero or more matches of the given RDF/OWL property IRI
  - Example: `( [ sh:zeroOrMorePath rdf:rest ] rdf:first )` - here, the first member of the sequence path is a zero-or-more path (for property `rdf:rest`), which - in combination with the predicate path of the second sequence path member (property `rdf:first`) - selects all members of an RDF list
- One-or-more path: a path that connects the subject and object of the path by one or more matches of the given RDF/OWL property IRI
  - Example: `[ sh:oneOrMorePath foaf:knows ]` - will select resources that are directly or indirectly known (via `foaf:knows`) by the focus node

NOTE	SHACL property paths are a subset of SPARQL 1.1 property paths. Thus, the SPARQL 1.1 Query Language [https://www.w3.org/TR/sparql11-query/#propertypaths] is a useful source to learn more about property paths.

NOTE	SHACL predicate paths will be useful for converting UML properties. SHACL sequence paths could be useful for converting OCL constraints.

8.3.3.2. Non-Validating Property Shape Characteristics

SHACL defines a number of predicates that are not used for validating. Instead, they can be used for documentation purposes, and for building GUIs.

- `sh:name` and `sh:description` - Can be used (only) in a property shape to provide a human readable label and description for a property that is being validated by the shape.
- `sh:order` - Can be used to define an order (using literals - typically integers - in ascending order) for shapes on the same level, typically property shapes. One use case is the creation of a form for data insertion, with individual fields for each property, arranged in the specified order.
- `sh:group` - Can be used to define that certain shapes belong to the same group (defined as a resource of type `sh:PropertyGroup`, which may have labels etc).
- `sh:defaultValue` - Can be used to define a default value within a property shape. That value can be displayed in an input form, for example. The value type should conform to any value type constraint defined in the shape.

8.3.4. Graphs

A typical SHACL validation workflow consists of taking a set of SHACL Shapes and evaluating them against some RDF data.

NOTE	Often, the shapes are stored separately in a Shapes Graph, and the data is stored in a Data Graph. However, shapes and data can be combined in a single file, and linking between and to shapes graphs is possible. Therefore, whenever the term *shapes graph* is mentioned, it refers to the sum of all SHACL shapes to be validated, and if a *data graph* is mentioned, it refers to all other RDF data - regardless of which actual RDF graph contains the corresponding triples.

8.3.4.1. Shapes Graph

The shapes graph contains the SHACL shapes that shall be used for validating RDF data.

Shapes can be defined in multiple separate RDF graphs (typically stored in different files, or made available at a different URL). The shapes from another RDF graph can be imported using an `owl:imports` predicate. A SHACL validator will automatically follow all `owl:imports` defined in imported RDF graphs in order to construct the shapes graph for validation.

The shapes graph may also contain the `sh:entailment` predicate. This predicate defines which kind of inferencing is required by a shapes graph.

Some background first: SHACL automatically uses information from `rdf:type` and `rdfs:subClassOf` predicates defined in the data graph. However, the shapes in the shapes graph may require specific information to be present for validation. If that information can be computed using inferencing, and an entailment regime supports this kind of inferencing, then the designer of the SHACL shapes may decide to require this entailment using `sh:entailment`.

The value of `sh:entailment` is an IRI. Some values are defined by SPARQL 1.1 Entailment Regimes [https://www.w3.org/TR/sparql11-entailment/]. Additional entailment regimes are possible. If a SHACL implementation does not support a certain entailment, then validation will result in a failure.

NOTE	SHACL validation does not add triples to the shapes or data graph. Thus inferred triples are only available during the validation process.

8.3.4.2. Data Graph

The data graph contains the RDF data that shall be validated. It is expected to include relevant vocabulary definitions (such as the definitions of classes and subclasses [using the `rdfs:subClassOf` predicate]), and statements that define the type(s) of RDF resources (via the `rdf:type` predicate).

NOTE	The predicate `sh:ShapesGraph` can be used within a data graph to identify one or more graphs with shapes that should be added to the shapes graph for validation. In other words, the predicate can be used to link from a data graph to one or more graphs containing shapes.

8.3.5. Core Constraint Components

The SHACL core constraint components are supported by all SHACL processors. The following subsections give a brief overview of each of these components.

NOTE Constraint components can be used in both node and property shapes, unless explicitly stated otherwise.

NOTE

The definitions of constraint components use the term *value node*, which is defined in the context of a shape.

- For a node shape, the value node is the focus node.
- For a property shape with `sh:path` *p*, the value nodes are the set of nodes that the path *p* leads to, starting at the focus node.

8.3.5.1. Value Type Constraint Components

- `sh:class` - Checks that each value node is of a given type.
  - NOTE: automatically takes into account explicitly defined `rdfs:subClassOf` relationships
  - Multiple values for `sh:class` are automatically combined using 'and', i.e., the focus nodes must belong to all specified types.
- `sh:datatype` - Checks that each value is of a given datatype.
  - Multiple values for `sh:datatype` are not allowed.
  - In order for a value node to satisfy the `sh:datatype` constraint, the datatype IRI of the value node - as computed by the SPARQL datatype() function - must match the specified datatype IRI.
- `sh:nodeKind` - Restricts each value node to be one of (a combination of) general node kinds: blank node, IRI, and literal. The according SHACL identifiers are `sh:BlankNode`, `sh:IRI`, `sh:Literal`, `sh:BlankNodeOrIRI`, `sh:BlankNodeOrLiteral`, and `sh:IRIOrLiteral`.
  - There can only be one value for `sh:nodeKind`.
  - Can be useful to ensure that blank nodes are not used as property values.

8.3.5.2. Cardinality Constraint Components

- `sh:minCount` - To define the minimum number of values that are expected for the SHACL property path (which is given via `sh:path`).
- `sh:maxCount` - Same as `sh:minCount`, just defining the maximum number of values.

NOTE The default cardinality in SHACL for a property shape is {0,unbounded}.

NOTE `sh:minCount` and `sh:maxCount` cannot be used in node shapes.

8.3.5.3. Value Range Constraint Components

- `sh:minInclusive` - Checks that a value is >= the value defined by this constraint component.
- `sh:maxInclusive` - Same as before, just with <=.
- `sh:minExclusive` - Same as before, just with >.
- `sh:maxExclusive` - Same as before, just with <.

These constraint components can be useful to define constraints for basic types.

NOTE	The implied comparison (>=, <=, >, <) works as in SPARQL, and thus not only supports comparison of numeric types, but also comparison of, for example, strings and dates (for further details, see section "Operator Mapping" [https://www.w3.org/TR/sparql11-query/#OperatorMapping] in the SPARQL 1.1 standard).

8.3.5.4. String-based Constraint Components

- `sh:minLength` - Defines the minimum length of a value node.
- `sh:maxLength` - Defines the maximum length of a value node.
- `sh:pattern` - Defines a regular expression that a value node must match.
  - The property `sh:flags` can be added, in order to define flags for the interpretation of the regular expression.
  - Internally, matching is performed as defined in section "Regex" [https://www.w3.org/TR/sparql11-query/#func-regex] of the SPARQL 1.1 standard (which, according to the SPARQL standard, actually invokes the `fn:matches` function defined in the XPath and XQuery Functions and Operators [https://www.w3.org/TR/xpath-functions/#func-matches] standard).
- `sh:languageIn` - Can be used to restrict the set of allowed language tags.
  - Will fail validation if the value is not an `rdf:langString`.
- `sh:uniqueLang` - Can be set to true in order to enforce that no pair of value nodes has the same language tag.
  - Cannot be used in a node shape.

NOTE	`sh:minLength`, `sh:maxLength`, and `sh:pattern` can be applied to literals as well as IRIs, but not to blank nodes.

8.3.5.5. Property Pair Constraint Components

- `sh:equals` - Check that the set of values identified using `sh:path` is equal to the set of values of the property identified by `sh:equals`.
- `sh:disjoint` - Same as `sh:equals`, but the sets of values must be disjoint.
- `sh:lessThan` - Check that all values identified using `sh:path` are less than - using SPARQL's < operator (which supports, for example, comparison of numbers, strings, dates; for further details, see this note) - any of the values of the property identified by `sh:lessThan`.
- `sh:lessThanOrEquals` - Same as `sh:lessThan`, just with <= operator instead of <.

NOTE	All property pair constraint components can only be used by property shapes.

NOTE	The properties identified by the property pair components are evaluated in the context of the focus node of the shape that has the property constraint. In other words, the set of values identified using `sh:path` cannot be compared to values from a property other than one of the focus node.

NOTE	While sh:path can contain a property path, sh:equals etc. cannot (a single IRI is expected as value of a property pair constraint).

8.3.5.6. Logical Constraint Components

- sh:not
- sh:and
- sh:or
- sh:xone

These constraint components implement the common logical operators *and*, *or*, *not*, and *exclusive or*.

The subect and the object(s) of a logical constraint component are shapes in general, i.e., there is no restriction to either node shape or property shape.

NOTE	For sh:xone, exactly one shape must match - even if sh:xone consists of more than two shapes.

8.3.5.7. Shape-based Constraint Components

- sh:node - Specifies the shape that each value node must conform to.
  - Can be used to realize type discriminator unions.
  - Potentially useful to realize the OCL forAll() iterator call.
  - Can be used to realize generalization for basic types. For types that are represented as classes, with rdfs:subClassOf relationships between supertypes and subtypes, using sh:node to explicitly require testing against the shape of a supertype is not necessary, because shapes that apply to supertypes are automatically applied to subtypes, if a rdfs:subClassOf relationship (chain) exists between super- and subtype.
  - In theory, sh:node could be used to enforce shape validation of supertypes for values that do not declare a type, and if rdfs:subClassOf relationships are lacking. However, relevant ontologies or RDFS schemas and thus rdfs:subClassOf relationships should always be made available to a SHACL validator before validation of a certain RDF data graph, and thus rdfs:subClassOf relationships can be expected to be present for validation. rdf:type predicates, on the other hand, may very well be missing in the RDF data that shall be validated (for example because for some resources, their specific type is not known yet).
  - An alternative to using sh:node would be to use sh:and with the supertype shape as one condition and all type specific constraints as additional conditions. That would be similar to how generalization is implemented in the JSON Schema encoding. Keep in mind, though, that SHACL can automatically validate a subtype instance against the shapes defined for all its supertypes, if the rdf:type and rdfs:subClassOf relationships are correctly defined - which should be the goal for a linked data application (maybe using inferencing as a pre-processing step).
- sh:property - To define that each value node must be valid against a certain property shape.

- `sh:qualifiedValueShape` - To define that a specific number of value nodes must conform to a given shape. The number of required matches is defined using `sh:qualifiedMinCount` and `sh:qualifiedMaxCount`.
  - Cannot be used in node shapes.
  - It is possible to define multiple property shapes for the same property (using the same `sh:path`) within a single shape, and set `sh:qualifiedValueShapesDisjoint` in each of these property shapes to true in order to define that validation can only succeed if a value conforms to a single shape defined using `sh:qualifiedValueShape` in the set of property shapes.

8.3.5.8. Other Constraint Components

- `sh:closed`, `sh:ignoredProperties` - `sh:closed` can be used to ensure that an RDF resource only has values for the properties for which property shapes are defined. `sh:ignoredProperties` can be used to define a set of additional properties which may still occur on such a resource (e.g., `rdf:type`).
  - Can be useful to achieve an RDF encoding that only contains RDF terms as defined by the application schema. However, this kind of restriction should be used with care, since one of the major characteristics of linked data is its openness, i.e., that additional triples can be defined for an RDF resource. Typically, an RDF resource should be considered valid if it passes the validation rules derived from the application schema - even if the resource has additional predicates not covered by these rules.
- `sh:hasValue` - Checks that the property has at least the specified value, which is either an IRI or a literal.
  - Could be useful for encoding specific uses of the OCL iterator call *exists()* (e.g., "inv: self.property→exists(x|x = value)").
- `sh:in` - Checks that each value of a property is one of a given list (of IRIs and/or literals).
  - Literals need to be matched exactly. "4"^^xsd:integer is not equal to "4"^^xsd:decimal.
  - Can be useful to encode enumerations.

8.3.6. SPARQL-based Constraints

SHACL SPARQL is an extension of SHACL Core, which supports using SPARQL SELECT and ASK queries for SHACL validation.

SPARQL queries can either be defined explicitly, using predicate `sh:sparql`, or using SPARQL based constraint components. The latter represent building blocks for reusable SHACL constraints.

A full introduction of SPARQL-based constraints for SHACL validation is out of scope for this report. For details about SPARQL-based constraints, please refer to the W3C Shapes Constraint Language standard [https://www.w3.org/TR/shacl/].

8.4. SHACL Conversion Rules

This section documents rules for converting an application schema to a set of SHACL shapes.

NOTE

The rules are typically identified using a string that follows the pattern rule-shacl-{UML model element type}-{specific rule identifier suffix}, where the UML model element type is either "pkg" (for UML packages), "cls" (for UML classes), "prop" (for UML properties, or "all" (for UML packages, classes, and properties). The rule identifier suffix results in a unique rule ID. Ideally, it encodes the intent of the encoding rule, in one or more words. Examples for SHACL conversion rule identifiers are *rule-shacl-cls-encode-featuretypes* and *rule-shacl-cls-union-typeDiscriminator*.

Generation of SHACL shapes for validating RDF data needs to take into account that RDF encoded information may come in different formats. As shown in chapter 6 of the OGC Testbed-14: Application Schemas and JSON Technologies Engineering Report [http://docs.opengeospatial.org/per/18-091r2.html], RDF derived from JSON using JSON-LD can be different to RDF that is expected by an OWL ontology. The SHACL conversion rules therefore should be designed in a way that supports multiple RDF structural forms.

At the moment, the only ShapeChange target that generates such a format is the ShapeChange ontology target. However, in the future, ShapeChange may be extended to produce JSON-LD context documents, with which JSON data - for example defined using the new ShapeChange JSON Schema target (implementing the UML to JSON Schema Encoding Rule) - can be transformed to RDF. One benefit of transforming JSON data to RDF using JSON-LD, and validating that RDF data using SHACL, would be that SHACL supports checking of class specialization and property ranges, which is not fully supported by JSON Schema (as explained here).

NOTE

In order to check that the value object of a JSON member has a valid type (the value type of the UML property represented by the JSON member, or a subtype of that type), the object must have a type declaration in RDF. Either the JSON-LD transformation assigns such a type, or the type is inferred before the SHACL validation is executed. In addition, type hierarchies must be defined correctly in order for the SHACL validation to recognise that a certain type is a subtype / `rdfs:subClassOf` another type.

The SHACL conversion rules should likely be implemented in a new ShapeChange target, and not in existing targets (such as the ShapeChange ontology and JSON Schema targets). Having one dedicated target for generation of SHACL would improve usability and maintainability of the SHACL conversion.

Some general considerations related to such a target.

- The RDFS/OWL implementation of application schema elements - types and properties - will need to be made known to the target using RDF map entries. Such map entries could be produced by other ShapeChange targets, the ShapeChange ontology target in particular but also the ShapeChange JSON Schema target, for example in case of a 1:1 transformation to RDF using ad-hoc created JSON-LD.

NOTE

  - There is one crucial aspect that needs to be considered when validating such an ad-hoc RDF format: type hierarchies would not be validated because the format has no RDF schema, and thus no `rdfs:subClassOf` relationships between the types used in that format are available. If a JSON-LD encoding leads to an ad-hoc RDF format, and that format shall be validated using SHACL, then a capability for producing a rudimentary RDF schema is needed. From a conceptual point of view, the place to realize that capability would be the component responsible for the JSON-LD encoding - because it is responsible for creating the ad-hoc RDF format.

- Potential variations in RDF structure produced by the ShapeChange ontology and JSON Schema targets will need to be handled by defining SHACL conversion rules that support such variations. SHACL encoding rules can then be defined, using subsets of these conversion rules as needed to validate a specific RDF format.

8.4.1. Documentation

SHACL Core does not define general elements with which the documentation of UML packages, classes, properties, and associations could be represented. It would be possible to encode descriptive information using RDF/OWL properties, for example the human readable name of a model element as `rdfs:label`. That could be useful when displaying the generated SHACL shapes. However, a SHACL processor is not required to include such information in a SHACL validation report. Therefore, a mechanism for converting the documentation of model elements when generating SHACL has not been defined. If actual requirements regarding such a mechanism are identified in the future, then such a mechanism can be developed. The descriptor targets supported by the ShapeChange ontology target [https://shapechange.net/targets/ontology/uml-rdfowl-based-isois-19150-2/#Descriptor_Targets] could be used as a basis for that mechanism.

NOTE

Non-Validating Property Shape Characteristics can be used for the documentation of UML properties in SHACL. For further details, see the property documentation section.

8.4.2. Package

A single OWL ontology - from now on called *SHACL ontology* - is created per application schema package. SHACL shapes derived from the content of the application schema will be defined in this ontology.

NOTE	A SHACL ontology does not define OWL classes or properties, but SHACL shapes. Therefore, a SHACL ontology is an ontology only in a very broad sense.

NOTE	The SHACL standard recommends, but does not require that SHACL shapes are defined as part of an OWL ontology. Nevertheless, looking at the RDF encoding of the DASH Data Shapes Vocabulary [http://datashapes.org/dash.ttl] and the RDF encoding of SHACL itself [http://www.w3.org/ns/shacl.ttl], that appears to be a best practice.

8.4.2.1. Name and Namespace

The SHACL ontology must have a name and a namespace.

The ontology name is constructed as follows.

- If *rule-shacl-pkg-ontologyName-byTaggedValue* is included in the encoding rule and a UML tagged value - identified by the configuration parameter *ontologyNameTaggedValue* (default value: ontologyName) - is set for the package, use its value as the base ontology name.
    - Note that this rule heavily leans on *rule-owl-pkg-ontologyName-byTaggedValue*, a rule supported by the ShapeChange ontology target.
- Otherwise, append the normalized application schema package name to a base URI, using "/" as separator, to create the base ontology name. The base URI is defined by the target parameter *URIbase*, if not blank. Otherwise, the *targetNamespace* of the application schema package is used as base URI.
- If *rule-shacl-pkg-ontologyName-appendVersion* is enabled, and the *version* UML tagged value of the application schema package is not blank, append the version to the base ontology name, using "/" as separator.
- Finally, append the value of the target parameter *shaclOntologyNameSuffix* (default value is: "Shapes"), using "/" as separator - unless the parameter is defined with a blank value.

The namespace is constructed by appending a separator to the name (default separator is '#', but the target parameter *rdfNamespaceSeparator* can be used to set a different separator, e.g., '/').

8.4.2.2. Imports

Validation may depend on SHACL shapes defined for types external to the application schema. Such a situation occurs if:

- A shape-based constraint component is created, which refers to the SHACL shape definition of the external type;
- A value type constraint component is created, which refers to the RDFS/OWL class or datatype defined for the external type, and a SHACL shape exists for the external type as well; or

- A node shape is created for a type from the application schema, a supertype of that type is from an external schema, and a SHACL shape exists for that supertype.

NOTE That a SHACL shape exists for an external type should be defined in the configuration of the ShapeChange SHACL target using map entries. The ShapeChange SHACL target produces an OWL ontology (though only in a very broad sense: the ontology does not define OWL classes or properties, but SHACL shapes), and - what is more important - it depends on RDF mappings for UML types and properties (to identify their RDFS/OWL implementation). The ShapeChange configuration of the ShapeChange SHACL target may thus use the <sc:TargetOwl> configuration element to define the target configuration items (parameters, map entries, namespaces, rules, etc). The <sc:TargetOwl> element can contain a set of <sc:RdfTypeMapEntry> and <sc:RdfPropertyMapEntry> elements. <sc:RdfTypeMapEntry> elements would have to be extended, so that it is possible to indicate that the map entry target is a SHACL shape. This could be achieved by extending the set of allowed values for the *targetType* XML attribute. Currently, the only allowed values are 'datatype' and 'class'. New values could be 'shaclshape' and 'class/shaclshape'. The latter would cover cases in which the RDFS/OWL definition of a class also is a SHACL shape. These new values would need to be taken into account by the ShapeChange ontology target as well.

In any of the aforementioned situations, the SHACL ontology defining the shape for the external type needs to be imported, using `owl:imports`.

NOTE The <sc:RdfTypeMapEntry> that defines the SHACL shape for the external type identifies the <sc:Namespace> element that contains the information needed to create the `owl:imports`. The namespace is identified by matching the prefix of the QName in sc:RdfTypeMapEntry/@target with the abbreviation defined for the namespace (sc:Namespace/@nsabr). The location defined for the namespace (in sc:Namespace/@location) will be used in the `owl:imports`.

NOTE The ShapeChange SHACL target may likely be implemented as a so-called ShapeChange *SingleTarget*. In that case, multiple schemas can be processed at once by the target, resulting in multiple SHACL ontologies. SHACL shapes and their ontology namespaces can then automatically be determined by ShapeChange for types defined by these schemas (and the user does not need to define map entries for such types to convey that information; however, map entries defining the RDFS/OWL class/datatype/property implementation would still be needed).

The import of an ontology or RDF schema (defined in the configuration of the Shapechange SHACL target using <sc:Namespace> elements) can also be enforced by setting UML tagged value *shaclForcedImports* on the application schema package, with the value being a comma-separated list of namespace abbreviations defined for these namespaces (via XML attribute nsabr of the <sc:Namespace> element defined in the target configuration).

NOTE

Enforcing the import of certain SHACL ontologies can be useful, for the following reasons.

- Testing with TopBraidComposer Maestro Edition v6.3.2 revealed that the tool only validated SHACL if the SHACL ontology imported the TopBraid Data Shapes Library [https://topbraid.org/tosh] (TOSH) - either directly, or indirectly through the Data Shapes Vocabulary [http://datashapes.org/dash] (DASH).
- When custom subClassOf mappings [http://docs.opengeospatial.org/per/16-020.html#_custom_subclassof_mappings] to certain RDFS/OWL classes have been added to the original ontology, and SHACL validation shall include validating SHACL shapes defined for these classes, then the SHACL ontologies that contain these shapes need to be imported.
- When the value type constraint `sh:class` identifies an RDFS/OWL class for which subclasses are known to exist, SHACL validation should include the shapes defined for these subtypes, and these shapes would not be imported automatically.

In the latter two cases, it is important to force import not only the SHACL shapes for the RDFS/OWL classes, but also the RDF schemas and ontologies that define these classes, so that SHACL validation has access to relevant `rdfs:subClassOf` relationships.

8.4.2.3. Entailment

SHACL validation may depend on triples that are not contained in the data graph, but which may be inferred using some entailment regime. As described in the Shapes Graph section, the `sh:entailment` predicate can be used to tell the SHACL processor which entailment regimes are required for SHACL validation.

Values of `sh:entailment` are IRIs, with some values being defined by SPARQL 1.1 Entailment Regimes [https://www.w3.org/TR/sparql11-entailment/] (see Table 12).

Table 12. Entailment regimes defined for SPARQL 1.1

Name	IRI
RDF	http://www.w3.org/ns/entailment/RDF
RDFS	http://www.w3.org/ns/entailment/RDFS
D-Entailment	http://www.w3.org/ns/entailment/D
OWL 2 RDF-Based Semantics	http://www.w3.org/ns/entailment/OWL-RDF-Based
OWL 2 Direct Semantics	http://www.w3.org/ns/entailment/OWL-Direct
(Simple) RIF Core Entailment Regime	http://www.w3.org/ns/entailment/RIF

NOTE

RDFS entailment can be used to make inferences based upon `rdfs:subPropertyOf` relationships, which can be quite useful as explained in more detail here.

In order to add the IRI of an entailment regime to a SHACL ontology, set the ShapeChange SHACL target parameter *entailmentRegimes*. Multiple parameter values are separated by spaces.

NOTE	Tests with TopBraidComposer (TBC) Maestro edition v6.3.2 conducted during the analysis of SHACL in UGAS-2020 showed that TBC supports the RDFS entailment regime. Other entailment regimes, like D-Entailment, do not appear to be supported.

8.4.3. Types

8.4.3.1. General

A UML type is converted to a SHACL node shape (`sh:NodeShape`), having `sh:targetClass` predicate with the RDFS/OWL class that represents that type as object, unless:

- the conversion rule *rule-shacl-all-notEncoded* applies to the type,
- the type is mapped to an existing SHACL shape, or
- one of the following sections explicitly states a different encoding for a certain type category.

A property of the type is converted as described in the Property section - unless explicitly stated otherwise in the following sections.

8.4.3.2. Mapping

As outlined in the section on mapping types in the JSON Schema chapter, application schemas typically use types from external schemas, as value types of properties, and as supertypes. Whenever a specific target implementation for such an external type is needed, ShapeChange looks it up.

The lookup is typically done as follows.

- If a map entry is defined for the type, use the existing external implementation of the type defined by the map entry.
- Otherwise, if the type belongs to the application schema itself - in case of a normal ShapeChange target -, or if it belongs to one of the schemas selected for processing - in case of a ShapeChange "SingleTarget" -, use the implementation that is created by the target.
- Otherwise ShapeChange should log an error (that no implementation was found for the external type).

NOTE	As outlined in the Imports section, the ShapeChange SHACL target will use map entries that define the RDFS/OWL implementation of a type. Accordingly, RdfTypeMapEntries as defined for the ShapeChange ontology target [https://shapechange.net/targets/ontology/uml-rdfowl-based-isois-19150-2/#RdfTypeMapEntry] will be needed. Such map entries will also be used to identify the SHACL shapes that apply for mapped types.

For the ShapeChange SHACL target, the lookup procedure described above only applies for the SHACL implementation of an external type. Map entries defining RDFS/OWL implementations of

types and properties will have to be provided for all types and properties defined and used by the application schema.

8.4.3.3. Type Name

The name of the SHACL node shape is a combination of the RDF namespace of the SHACL ontology and the UML type name:

```
nodeShapeName = rdfNamespace + UmlTypeName
```

The UML type name is given in upper camel case. Punctuation characters other than dash and underscore (i.e., the following ASCII characters: `] [ ! "# $ % & ' * + , . / : ; < = > ? @ \ ^ ` { | } ~`) are replaced by underscore characters. Whitespace characters are removed.

8.4.3.4. Abstractness

An abstract type is encoded as a SHACL node shape like any non-abstract type. That will ensure that RDF resources that are of that abstract type are validated according to the SHACL constraints derived from the abstract type.

With *rule-shacl-cls-dashAbstract*, the predicate `dash:abstract = true` will be added to the node shape of an abstract type. The predicate is defined by the Data Shapes Vocabulary [http://datashapes.org/dash#abstract-classes], and while it does not have any validating effect, it can be used by software tools to prevent a user from creating instances of the abstract type.

8.4.3.5. Generalization/Inheritance

A SHACL processor automatically applies all constraints defined in a node shape with a certain type as `sh:targetClass` to instances of that type as well as instances of all its subtypes. Thus, generalization / inheritance - even multiple inheritance - is automatically handled by a SHACL processor.

NOTE As described in the section on Shape-based Constraint Components, a conversion rule to explicitly encode generalization using `sh:node`, with the node shape of the supertype as value, would only be needed if `rdfs:subClassOf` relationships were not available. If a use case was identified where SHACL validation under such conditions was necessary, such a conversion rule could be defined.

NOTE The ShapeChange ontology target can add custom subClassOf mappings [http://docs.opengeospatial.org/per/16-020.html#_custom_subclassof_mappings], i.e., `rdfs:subClassOf` predicates which do not have a direct representation within the UML model. This can be used to - for example - define that any OWL class that encodes a feature type is an `rdfs:subClassOf geosparql:Feature`. SHACL shapes may exist for validating the corresponding RDFS/OWL classes. Such shapes need to be imported explicitly. For further details, see the section on Imports.

8.4.3.6. Feature and Object Types

If *rule-shacl-cls-encode-featuretypes* is enabled, feature types will be converted to SHACL node shapes. Likewise, if *rule-shacl-cls-encode-objecttypes* is enabled, object types will be converted to SHACL node shapes.

The ShapeChange ontology target does not declare specific conversion rules that influence how properties of feature and object types - both types with identity - are encoded. The ShapeChange JSON Schema target, on the other hand, does define such rules.

- An identifier property may be added to the JSON encoding of a type with identity using *rule-json-cls-identifierForTypeWithIdentity*. For further details, see Identifier.
- Using *rule-json-cls-nestedProperties*, the properties of a type with identity may be JSON encoded within a "properties" member. For further details, see Nested Properties.

As described in the OGC Testbed-14: Application Schemas and JSON Technologies Engineering Report [http://docs.opengeospatial.org/per/18-091r2.html#JSON_LD_recommendations_and_best_practices_keywords] on best practices regarding JSON-LD keywords, the JSON-LD keyword *@nest* can and should be used to remove unnecessary property nesting in JSON data before transforming that data to RDF. Therefore, no specific SHACL conversion rule has been defined to cope with nested properties.

It should be possible, however, to validate an additional identifier property. With *rule-shacl-cls-identifierForTypeWithIdentity*, a SHACL property shape is added to the node shape that represents the type with identity. The following ShapeChange SHACL target parameters are used to provide information about that property (they are similar to the ones defined for *rule-json-cls-identifierForTypeWithIdentity*):

- *objectIdentifierName*: the name of the identifier property; default value is "id";
- *objectIdentifierType*: the type of the identifier property - either "string" (the default), "number", or "string, number";
  - NOTE: "string, number" can be represented in SHACL using `sh:or ( [sh:datatype xsd:string;] [sh:datatype xsd:integer;] )`
- *objectIdentifierRequired*: "true" or "false" (the default) is used to define if the property is required or optional.

The property shape for the identifier is constructed following the conversion rules documented in the Property section.

8.4.3.7. Mixin Types

ShapeChange supports the notion of mixin type (for further details, see the JSON Schema chapter).

If *rule-shacl-cls-encode-mixintypes* is enabled, then a SHACL node shape will be created for a mixin type.

NOTE	This supports cases in which mixins are represented as RDFS/OWL classes, for example using *rule-owl-cls-encode-mixintypes* (see the OGC Testbed-12 report, section "Mixin Types" [http://docs.opengeospatial.org/per/16-020.html#rdf_cr_class_mixin]). If mixins are not represented as RDFS/OWL classes - typically because they have been dissolved, copying their properties down to non-mixin subtypes and removing the mixin types from the conceptual model - then a SHACL node shape would not be able to identify these mixins using `sh:targetClass`, and validation with that shape would never occur.

Otherwise, the ShapeChange SHACL target will ignore the mixin. In order to ensure that mixin properties are available for non-mixin subtypes, the ShapeChange Flattener transformation should be used in a pre-processing step, using *rule-trf-cls-dissolve-mixins* *[https://shapechange.net/transformations/flattener/#rule-trf-cls-dissolve-mixins]*.

NOTE	As of July 2020, *rule-trf-cls-dissolve-mixins* does not support copying of association roles. If association roles are relevant for mixins within a given application schema, *rule-trf-cls-dissolve-mixins* needs to be enhanced to support association roles.

8.4.3.8. Data Types

If *rule-shacl-cls-encode-datatypes* is enabled, data types will be converted to SHACL node shapes.

8.4.3.9. Basic Types

If a direct or indirect supertype of an application schema class is mapped to an RDFS/OWL datatype, then that class represents a so called *basic type*.

NOTE	A similar logic exists for the JSON Schema encoding rules (see the JSON Schema chapter, section Basic Type). The ShapeChange ontology target, however, currently encodes basic types as OWL classes (see the OGC Testbed-12 ShapeChange ER, section Basic Types [http://docs.opengeospatial.org/per/16-020.html#rdf_cr_class_basictype]). A future work item addresses the revision of the ShapeChange ontology target with respect to encoding of basic types.

In RDF, a basic type is represented by a literal value, typically using one of the XSD datatypes. Basic types may be defined in an inheritance tree (see Figure 10 in the JSON Schema chapter). A basic type usually defines some kind of restriction, for example a numerical range. These are typically modelled using UML tagged values.

A basic type is encoded as a SHACL node shape. No `sh:targetClass` predicate is defined for that shape. Instead, any property shape that intends to check the value type of a property with a basic type as value type, will use `sh:node` instead of a value type constraint, referencing the node shape defined for the basic type.

NOTE	Referencing the SHACL node shape defined for the basic type allows us to define the value restrictions in a single place - that node shape. An alternative would be to repeat the restrictions in each property shape that needs to check the value type of a property with a basic type as value type. However, that would result in a verbose shapes graph.

If the basic type has no other basic type as supertype - only a supertype that is mapped to an RDFS/OWL datatype -, then the value type constraint `sh:datatype` is added to the node shape, with the RDFS/OWL datatype as value. Otherwise - the basic type does have another basic type as supertype - an `sh:node` predicate is added to the node shape, with the node shape defined for the supertype as value.

If a restriction is defined for the basic type via UML tagged value, it is encoded using the appropriate SHACL keyword - see Table 13.

NOTE	In general, the restrictions listed in Table 13 only work for XSD datatypes - not for custom data types. Furthermore, some restrictions only work for certain XSD datatypes.

NOTE	A ShapeChange SHACL target implementation should re-use functionality that is already available in the ShapeChange XmlSchema target, in order to identify if a certain restriction is supported for a given datatype.

Table 13. Basic type restrictions

SHACL keyword	UML tagged value to define the restriction
`sh:maxLength`	*length* and *maxLength*
`sh:pattern`	*pattern* and *jsonPattern* NOTE: The *jsonPattern* UML tagged value needs to have a syntax supported by `sh:pattern` - for further details, see the description of the keyword in String-based Constraint Components.
`sh:minInclusive`	*rangeMinimum*
`sh:maxInclusive`	*rangeMaximum*

Example:

```
exshacl:ClassShape
  a sh:NodeShape ;
  sh:targetClass exschema:Class ;
  sh:property [
    sh:path exschema:property ;
    sh:node exshacl:BasicTypeBShape ;
  ] .

exshacl:BasicTypeAShape
  a sh:NodeShape ;
  sh:datatype xsd:integer ;
  sh:minInclusive 0 .

exshacl:BasicTypeBShape
  a sh:NodeShape ;
  sh:node exshacl:BasicTypeAShape ;
  sh:maxExclusive 360 .
```

Given this RDF schema and RDF data:

```
exschema:Class a owl:Class .

ex:A a exschema:Class ;
  exschema:property "11"^^xsd:integer ;
  exschema:property "x" ;
  exschema:property "13"^^xsd:nonNegativeInteger ;
  exschema:property -1 ;
  exschema:property 361 .
```

Then all values of `exschema:property`, except for the first one shown in the example, are invalid:

- `exschema:property "x"` and `exschema:property "13"^^xsd:nonNegativeInteger` are invalid because the datatypes are not equal to `xsd:integer`
- `exschema:property -1` violates the `sh:minInclusive` constraint
- `exschema:property 361` violates the `sh:maxInclusive` constraint

NOTE Using `sh:node` in property shapes to ensure that a property value fulfills all constraints defined by the referenced node shape comes at a cost: if the value is invalid, the validation result does not tell us which actual constraint of the shape was violated. Instead, a general validation report is given, such as: "*Value does not have shape exshacl:BasicTypeBShape*".

8.4.3.10. Unions

As described in the JSON Schema chapter on Union types, unions can be used in two different ways: as type discriminators, and as choices between multiple options that are represented by the attributes of the union.

8.4.3.10.1. Type Discriminator

With *rule-shacl-cls-union-typeDiscriminator*, a type discriminator union can be encoded in SHACL. The union itself is represented as a node shape with a logical union of value type constraints, one for each distinct value type defined by the union attributes. Any SHACL property shape that represents a UML property with the union as value type will use `sh:node` to refer to that node shape, instead of using a value type constraint to validate the property values.

Example:

```
exshacl:Type1Shape
  a sh:NodeShape ;
  sh:targetClass exschema:Type1 ;
  sh:property [
    a sh:PropertyShape ;
    sh:path exschema:typeDiscrUnionProp ;
    sh:node exshacl:TypeDiscriminatorUnionShape;
  ] .

exshacl:TypeDiscriminatorUnionShape
  a sh:NodeShape ;
  sh:or (
    [sh:class exschema:Type2]
    [sh:class exschema:Type3]
    [sh:datatype xsd:string]
  ) .
```

Given this RDF schema and RDF data:

```
exschema:Type1 a owl:Class .
exschema:Type2 a owl:Class .
exschema:Type3 a owl:Class .
exschema:Type4 a owl:Class .

ex:A
  a exschema:Type1 ;
  exschema:typeDiscrUnionProp "Test" ; # 1
  exschema:typeDiscrUnionProp 42; # 2
  exschema:typeDiscrUnionProp ex:B ; # 3
  exschema:typeDiscrUnionProp ex:D ; # 4
.

ex:B a exschema:Type2 .
ex:C a exschema:Type3 .
ex:D a exschema:Type4 .
```

`ex:A` would be invalid because the second and fourth value of `exschema:typeDiscrUnionProp` do not match the value type constraints defined by `exshacl:TypeDiscriminatorUnionShape`.

8.4.3.10.2. Attribute Choices

With *rule-shacl-cls-union-attributeChoices*, a union that defines a choice between multiple options - represented by the attributes of the union - can be encoded in SHACL.

The union is represented as a node shape that uses a logical intersection of two `sh:xone` (see Logical Constraint Components) to achieve the choice between the properties defined by the union. The first `sh:xone` is used to ensure that one and only one of the options is used.

NOTE	This is necessary because an `sh:xone` checks that exactly one of the contained shapes is valid, so if an invalid option occurred, together with a different valid option, the `sh:xone` would be valid. But that is not the intent of a union.

The second `sh:xone` encodes each union property as a property shape, as usual.

Properties that have the union as value type use `sh:class` to ensure that the value has the correct type. A union instance is expected to have a correct type declaration. That declaration is sufficient to identify the node shape to use for validating the instance.

Example:

```
exshacl:Type1Shape
  a sh:NodeShape ;
  sh:targetClass exschema:Type1 ;
  sh:property [
    a sh:PropertyShape ;
    sh:path exschema:unionProp ;
    sh:class exschema:Union ;
    sh:minCount 1 ;
  ] .

exshacl:UnionShape
  a sh:NodeShape ;
  sh:targetClass exschema:Union ;
  sh:and (
    [sh:xone (
      [sh:property [
        sh:path exschema:option1 ;
        sh:minCount 1 ;
      ] ]
      [sh:property [
        sh:path exschema:option2 ;
        sh:minCount 1 ;
      ] ]
      [sh:property [
        sh:path exschema:option3 ;
        sh:minCount 1 ;
      ] ]
    )]
    [sh:xone (
```

```
      [sh:property [
        a sh:PropertyShape ;
        sh:path exschema:option1 ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
      ] ]
      [sh:property [
        a sh:PropertyShape ;
        sh:path exschema:option2 ;
        sh:class exschema:Type2 ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
      ] ]
      [sh:property [
        a sh:PropertyShape ;
        sh:path exschema:option3 ;
        sh:class exschema:Type3 ;
        sh:minCount 2 ;
        sh:maxCount 4 ;
      ] ]
    )]
  )
.
```

Given this RDF schema and RDF data:

```
exschema:Type1 a owl:Class .
exschema:Type2 a owl:Class .
exschema:Type3 a owl:Class .
exschema:Type4 a owl:Class .
exschema:Union a owl:Class .

exschema:unionProp a owl:ObjectProperty ;
    rdfs:range exschema:Union .

exschema:option1 a owl:DataProperty ;
    rdfs:range xsd:string .

exschema:option2 a owl:ObjectProperty ;
    rdfs:range exschema:Type2 .

exschema:option3 a owl:ObjectProperty ;
    rdfs:range exschema:Type3 .

ex:A a exschema:Type1 ;
  exschema:unionProp ex:Union1 ;
  exschema:unionProp ex:Union2 ;
  exschema:unionProp ex:Union3 .

ex:B a exschema:Type2 .
ex:C a exschema:Type3 .
ex:D a exschema:Type4 .

ex:Union1 a exschema:Union ;
  exschema:option1 "Test" ;
  exschema:option1 1 .

ex:Union2 a exschema:Union ;
  exschema:option2 ex:B .

ex:Union3 a exschema:Union ;
  exschema:option2 ex:D ;
  exschema:option3 ex:C .
```

`ex:Union1` is invalid because `exschema:option1 1` has a value type that is not allowed for `exschema:option1`. Furthermore, `ex:Union3` is invalid because both `exschema:option2` and `exschema:option3` occur. Without the first `sh:xone` in `exshacl:UnionShape`, `ex:Union3` would be valid, because `exschema:option2 ex:D` is invalid (wrong value type) while `exschema:option3 ex:C` is valid, and so the condition that exactly one of the shapes in `sh:xone` is valid would be fulfilled. Note that `ex:Union3` would also be invalid even if it did not have the `exschema:option2` - because it does not have at least two values for `exschema:option3`, as required by the shape for that property in the second `sh:xone`.

8.4.3.11. Enumerations

Enumerations are typically encoded as a set of allowed literals with a specific datatype. However, the ShapeChange ontology target also supports a conversion rule to treat an enumeration like a code list.

8.4.3.11.1. Choice of Literals

The case of an enumeration that is encoded as a choice of literal values is supported by *rule-shacl-cls-enumeration-choiceOfLiterals*. Here, a node shape is created for the enumeration, which has a value type constraint `sh:datatype`, as well as an `sh:in` constraint that lists the allowed literals.

The datatype of the enumeration is defined using UML tagged value *literalEncodingType*. The UML tagged value contains the name of a type from the conceptual schema that is mapped to a datatype. If the UML tagged value is missing or has a blank value, the datatype is assumed to be `xsd:string`.

NOTE	The same UML tagged value is used in the JSON Schema encoding of an enumeration to identify the type with which the enums defined by that enumeration shall be encoded.

Example:

```
exshacl:ClassShape
    a sh:NodeShape ;
    sh:targetClass exschema:Class ;
    sh:property [
        sh:path exschema:propertyA ;
        sh:node exshacl:EnumerationAShape ;
    ] ;
    sh:property [
        sh:path exschema:propertyB ;
        sh:node exshacl:EnumerationBShape ;
    ] .

exshacl:EnumerationAShape
    a sh:NodeShape ;
    sh:datatype xsd:integer ;
    sh:in ("1"^^xsd:integer "3"^^xsd:integer "5"^^xsd:integer "7"^^xsd:integer) .

exshacl:EnumerationBShape
    a sh:NodeShape ;
    sh:datatype xsd:string ;
    sh:in ("a"^^xsd:string "b"^^xsd:string) .
```

Given this RDF schema and RDF data:

```
exschema:Class a owl:Class .

ex:A a exschema:Class ;
    exschema:propertyA "1"^^xsd:integer ; # 1
    exschema:propertyA 5 ; # 2
    exschema:propertyA "7"^^xsd:nonNegativeInteger ; # 3
    exschema:propertyA "9"^^xsd:integer ; # 4
    exschema:propertyB "a" ; # 5
    exschema:propertyB "b"@en ; # 6
    exschema:propertyB "c" ; # 7
.
```

`ex:A` is invalid because:

- the third and sixth exschema predicates have a wrong datatype, and
- the fourth and seventh exschema predicates have an invalid enum value.

8.4.3.11.2. As Code List

To encode an enumeration like a code list, use *rule-shacl-cls-enumerationAsCodelist*. Then the conversion rules for Code Lists apply to the enumeration.

8.4.3.12. Code Lists

No particular SHACL shape is encoded for a code list. Instead, shapes for properties that have a code list as value type have a value type constraint - `sh:class` or `sh:datatype`, depending on the situation.

- If an <sc:RdfTypeMapEntry> is defined for the code list, it is mapped accordingly. The map entry defines the target and its type (class or datatype).
  - This covers the cases allowed by the ShapeChange ontology target, as defined in the OGC Testbed-12 ER [http://docs.opengeospatial.org/per/16-020.html#_code_lists] for code lists.
  - This would also cover cases in which code values are JSON-encoded as URIs, and a JSON-LD @context document maps these to individuals of a particular class.
- Otherwise, UML tagged value *literalEncodingType* - the same as described before - is used to identify the type of code values (with default being CharacterString). In this case, the value of the tag would map to an RDFS/OWL datatype, which would be used in an `sh:datatype` constraint.
  - This covers a situation that may occur in JSON encodings, where code lists may be mapped to simple JSON value types.

8.4.4. Property

8.4.4.1. General

A UML property is converted to a SHACL property shape (`sh:PropertyShape`), with `sh:path` predicate with the RDF/OWL property that represents that UML property as object, unless *rule-shacl-all-notEncoded* applies to the UML property.

The property shape is encoded as a blank node, which is referenced from a node shape via the `sh:property` predicate.

8.4.4.2. Documentation

As described in the section Non-Validating Property Shape Characteristics, a number of SHACL predicates can be used in property shapes that are not directly used for validation, but are nevertheless useful for descriptive purposes, for example structuring and populating data input forms. The following rules are available to generate these predicates.

- With *rule-shacl-prop-shname* the *alias* (one of the ShapeChange descriptor sources [https://shapechange.net/get-started/config/input/#Descriptor_sources]) of the property is encoded in the `sh:name` predicate. If no alias is defined for the property, use its name instead.
- With *rule-shacl-prop-shdescription* the *documentation* (one of the ShapeChange descriptor sources [https://shapechange.net/get-started/config/input/#Descriptor_sources]) of the property is encoded in the `sh:description` predicate.
- With *rule-shacl-prop-shorder* the sequence number of the property is encoded in the `sh:order` predicate, as an RDF literal with datatype string.
  - The datatype is string, because in addition to sequence numbers being pure integers, ShapeChange supports sequence numbers that are strings composed of integers separated by dots (e.g., 10.1, 10.1.3).

NOTE	*rule-shacl-prop-shname* and *rule-shacl-prop-shdescription* would not be needed if a general mechanism for documenting model elements was available, for example using descriptor targets like in the ShapeChange ontology target.

8.4.4.3. Range

The value type of a UML property is encoded using Value Type Constraint Components, more specifically `sh:class` - if the value type is mapped to an RDFS/OWL class - or `sh:datatype` - if the value type is mapped to an RDFS/OWL datatype.

In the following situations, however, `sh:node` is used instead of a value type constraint, referring to the SHACL shape that was created for the value type of the property:

- the value type is a basic type
- the value type is a type discriminator union
- the value type is an enumeration that is encoded as a choice of literals

8.4.4.4. Multiplicity

The multiplicity of a UML property can be encoded using Cardinality Constraint Components.

- With *rule-shacl-prop-minCardinality*, if the minimum cardinality of the property is different to 0, encode it using the `sh:minCount` predicate.
- With *rule-shacl-prop-maxCardinality*, if the maximum cardinality of the property is different to unlimited, encode it using the `sh:maxCount` predicate.

Due to the nature of RDF, where an RDF graph contains a set of triples - and thus the same combination of subject, predicate, and object does not exist twice, at least from a logical perspective - SHACL only counts distinct values. Furthermore, from a conceptual point of view, triples contained in an RDF graph are not ordered. Therefore, non-unique and ordered multiplicity cannot be represented in SHACL based validation scenarios.

NOTE

One might be tempted to use an RDF list as a property value, in order to represent a property with non-unique and/or ordered values. As shown on https://www.topquadrant.com/constraints-on-rdflists-using-shacl/ it is possible to inspect the values encoded in such a list with SHACL. However, again the set of values is checked, therefore duplicates and ordering will be ignored when the actual property constraints are evaluated.

Furthermore, the solution for gathering all values of an RDF list using a SHACL property path does not work for RDF containers such as bags and sequences of arbitrary length. The reason is that membership to a certain container is RDF encoded using instances of the rdfs:ContainerMembershipProperty class, whose names depend on numbering: rdf:_1, rdf:_2, rdf:_3, etc., which cannot be represented for a container with unknown size using a SHACL property path.

In general, it is advisable to avoid using RDF lists and containers, because they are not (well) supported by OWL and SHACL. Applications that absolutely require value collections other than sets may still use them, or alternative approaches such as the Collections ontology [https://lov.linkeddata.es/dataset/lov/vocabs/coll]. In any case, the conversion rules documented in this chapter only support plain sets of values. Development of conversion rules for other types and implementations of value collections is future work.

Subproperty relationships and entailment (for further details, see rdfs:subPropertyOf Relationships) can also impact the number of values that a SHACL validator identifies for a certain RDF/OWL property. With RDFS entailment, the values given for a sub-property S will also be counted for the properties that S directly or indirectly is a sub-property of. For cases in which the maximum cardinality of a UML property is restricted - for example to 1 - that can lead to unexpected validation results. The two conversion rules for encoding minimum and maximum cardinality constraints have been defined to allow 1) not checking cardinality constraints at all and 2) checking only minimum cardinality.

8.4.4.5. rdfs:subPropertyOf Relationships

An RDFS schema or OWL ontology may define `rdfs:subPropertyOf` relationships for properties that belong to the schema/ontology.

According to RDF Schema 1.1 [https://www.w3.org/TR/rdf-schema/#ch_subpropertyof], with `P1 rdfs:subPropertyOf P2`, any RDF resource that has `P1 {some_value}` implicitly also has `P2 {some_value}`.

NOTE	The ShapeChange ontology target supports defining custom subPropertyOf relationships [http://docs.opengeospatial.org/per/16-020.html#_custom_subpropertyof_mappings] for an RDF/OWL property that represents a UML property.

When evaluating a SHACL property path defined by an `sh:path` predicate in a SHACL property shape, a SHACL processor does not take into account `rdfs:subPropertyOf` relationships. If an `sh:path` was defined as `sh:path P1`, but an RDF resource only had a triple with P2 as predicate, a SHACL processor would flag that resource as being invalid - because predicate P1 was not found for it. However, RDFS entailment can be used to infer the missing predicate. For further details on entailment in SHACL processing, see section Entailment.

Example:

```
exshacl:ResourceShape
    a sh:NodeShape ;
    sh:targetClass exschema:Resource ;
    sh:property [
        a sh:PropertyShape ;
        sh:path rdfs:label ;
        sh:minCount 1 ;
    ] .
```

Given this RDF schema and RDF data:

```
exschema:Resource a owl:Class .

skos:prefLabel
  rdf:type rdf:Property ;
  rdf:type owl:AnnotationProperty ;
  rdfs:subPropertyOf rdfs:label .

ex:A a exschema:Resource ;
  rdfs:label "Resource A" .

ex:B a exschema:Resource ;
  skos:prefLabel "Resource B" .

ex:C a exschema:Resource .
```

Then without RDFS entailment, the SHACL processor would mark `ex:B` and `ex:C` as invalid because they do not have an `rdfs:label`.

However, if RDFS entailment was defined for the shapes graph, using `sh:entailment http://www.w3.org/ns/entailment/RDFS`, then the triple `ex:B rdfs:label "Resource B"` would be inferred before SHACL validation is performed, and the SHACL processor would mark `ex:B` as valid.

NOTE	If the schemas relevant for the RDF data that shall be validated with SHACL contain `rdfs:subPropertyOf` relationships, and the SHACL shapes to be validated contain property paths which use properties that are subjects of such relationships, then it is recommended to require RDFS entailment for the SHACL shapes graph (using the `sh:entailment` predicate).

8.4.4.6. Attribute

UML attributes within an application schema typically have a simple type, a basic type, a datatype, an enumeration, or a code list as value.

NOTE	Value types that are types with identity (feature and object types) are used by association roles.

UML attributes can have an initial value, which - under *rule-shacl-prop-initialValue* - is encoded in the `sh:defaultValue` predicate.

A UML attribute with a basic or simple type as value type may also have restrictions defined via UML tagged values, much like it is done for basic types. With *rule-shacl-prop-constrainingFacets*, SHACL predicates to represent these restrictions are added to the property shape that represents the attribute. The RDFS/OWL datatype to which the value type of the attribute is mapped, or as which the value type is implemented (in the case of a basic type), defines which facets can be encoded. For further details, see the basic types section.

8.4.4.7. Association Role

The general conversion rules for UML properties apply for the conversion of association roles.

No additional specific rules are defined for the conversion of association roles.

8.4.4.8. Property Metadata Stereotype

The property stereotype <<propertyMetadata>>, developed in the OGC UGAS-2019 Pilot, is used to indicate that values of the property can be associated with some metadata. The UGAS-2019 Pilot produced a report [https://shapechange.net/wp-content/uploads/2019/12/UGAS19-D100_property_stereotypes.pdf] that describes the concept in detail. Section 2.3.3 of that report also documents potential encodings to represent the concept in linked data. One of them is RDF reification, which uses `rdf:Statement` as defined by the W3C RDFS 1.1 standard [https://www.w3.org/TR/rdf-schema/#ch_statement] to associate an RDF triple with metadata.

SHACL Core does not define any mechanism for validating RDF statements about certain triples that are identified using property shapes. In that context, validation could for example be a test that the RDF statement is a valid representation of the metadata type defined for the property.

The unofficial document DASH Reification Support for SHACL [http://datashapes.org/reification.html] defines the RDF property `dash:reifiableBy`, which - according to the document - "can be used to link a SHACL property shape with one or more node shapes. Any reified statement must conform to these node shapes." This means that `dash:reifiableBy` would support SHACL validation of RDF statements against certain shapes, and thus the encoding of the property metadata stereotype in

SHACL.

However, tests using TopBraidComposer showed that `dash:reifiableBy` is not well supported yet. For further details, see a stackoverflow question [https://stackoverflow.com/questions/62551189/shacl-validation-using-dashreifiableby] which has been answered by Holger Knublauch (the author of "DASH Reification Support for SHACL").

API and tool support for `dash:reifiableBy` may improve in the future. For the time being, we can conclude that no widespread solution for defining and executing SHACL validation of RDF statements against certain shapes exists. Therefore, no conversion rule for the property metadata stereotype has been defined.

8.4.5. Association Class

Since association classes cannot be represented in RDF, no SHACL conversion rules are defined for association classes.

Association classes should be transformed as defined by GML 3.3. This approach is also used for the JSON Schema and the ontology [http://docs.opengeospatial.org/per/16-020.html#rdf_cr_associationclass] encodings.

8.4.6. OCL Constraints

A subtask of analyzing the potential use of SHACL to validate linked data was to analyze the possibility of translating OCL expressions which cannot be translated to OWL. These OCL expressions are documented in the OGC Testbed-14: Application Schema-based Ontology Development Engineering Report [http://docs.opengeospatial.org/per/18-032r2.html], see table 2 [http://docs.opengeospatial.org/per/18-032r2.html#table_RDF_oclTranslation_none] in section 5.1 as well as Annex B.2 of that report.

The analysis in UGAS-2020 first identified if a translation of every OCL language construct that is supported by ShapeChange (see the tables in section 5.1 of the OGC Testbed-14: Application Schema-based Ontology Development Engineering Report [http://docs.opengeospatial.org/per/18-032r2.html]) can be achieved using SHACL. Three potential approaches have been investigated:

- Translation solely based on SHACL Core
- Translation using SHACL Core and SHACL Advanced Features
- Translation using SPARQL queries, embedded in SHACL constructs

The analysis revealed that none of these approaches supports a full translation of all OCL language constructs right now. Since a full automatic conversion of the OCL language constructs is not possible, an attempt was made to manually define SHACL based solutions for validating the NAS OCL constraints that cannot be translated with OWL. The results are documented in section SHACL Implementations for specific NAS OCL Constraints.

8.4.6.1. Translation solely based on SHACL Core

SHACL Core supports the same OCL language constructs that the translation with OWL does. In addition, SHACL supports property paths and thus the collect() operator and shorthand for collect

notation.

Still, the following OCL language constructs cannot be translated when using only SHACL Core: let variables and expressions, arithmetic operations (+,-,*,/), string operations concat() and substring(), the operation call allInstances(), and the iterator call isUnique().

NOTE	A benefit of this approach is that any SHACL processor is able to validate the translated constraints, because only SHACL Core language components are used in the translation.

8.4.6.2. Translation using SHACL Core and SHACL Advanced Features

This approach makes use of language components from SHACL Core and especially SHACL functions and SHACL node expressions, which are defined by SHACL Advanced Features.

NOTE	SHACL Advanced Features is not a W3C Recommendation yet (as of July 2020). It is work in progress, which is currently published as a W3C Draft Community Group Report [https://w3c.github.io/shacl/shacl-af/]. The features defined by that specification may change at any time in the future. Furthermore, implementation support may be limited (when compared with implementation support for SHACL Core and SHACL SPARQL).

SHACL functions support arithmetic operations and string operations such as substring() and concat(). Basically, SPARQL-based functions would be used. That way, the full range of SPARQL expressions would be available in this approach. Annex C shows how SPARQL-based functions can be implemented.

However, SHACL functions can only be used in SHACL expression constraints. These constraints use node expressions, which do not support variables (like translating the variable 'self' at an arbitrary location within the OCL expression, or translating let variables).

NOTE	The only exception regarding variables are SPARQL node expressions. Within their queries, variables can be used - including the variable *?this* to represent the focus node. The SHACL Implementations for specific NAS OCL Constraints make use of this feature.

The operation call allInstances() cannot be realized using a SPARQL-function, because of the way that `sh:SPARQLFunction` is defined: *"For SELECT queries, the function's return value is the binding of the (single) result variable of the first solution in the result set. Since all other bindings will be ignored, such SELECT queries should only return at most one solution."*

NOTE	That means that no SPARQL-function can be defined whose return value would be the set of all instances of a certain type. Within a SPARQL query, however, it is possible to express the operation call allInstances(). That is shown in the SHACL implementation of the specific NAS OCL constraint Related Datatype Use Required (at least one).

The current version of the SHACL Advanced Features defines several node expression types, which

are useful for converting OCL language constructs. For example, there are expressions to select nodes (*path expressions*), to count a set of values (*count expressions*), to represent if-then-else (*if expressions*), etc. However, a node expression to represent universal quantification, i.e., the OCL iterator call forAll(), is missing. That may be the case because the node expression types are heavily based on SPARQL 1.1 language features, and universal quantification is not available in SPARQL 1.1 either. In some situations, forAll() can be represented using count expressions, as in the following example.

NOTE The example uses SPARQL-based functions whose definitions are given in Annex C.

```
# inv: self.role->forAll(b|b.attB > 5)

exshacl:ForAll_role
  a sh:NodeShape ;
  sh:targetClass exschema:ClassA ;
  sh:expression [
    shaclex:equals (
      [ sh:count [ sh:path (exschema:role exschema:attB) ] ]
      [ sh:count [
        sh:nodes [ sh:path (exschema:role exschema:attB) ] ;
        sh:filterShape [
          a sh:NodeShape ;
          sh:minExclusive 5
        ]
      ]]
    )
  ] .

# inv: self->forAll(a|a.attA < 5)

exshacl:ForAll_att
  a sh:NodeShape ;
  sh:targetClass exschema:ClassA ;
  sh:expression [
    shaclex:equals (
      [ sh:count [ sh:path exschema:attA ] ]
      [ sh:count [
        sh:nodes [ sh:path exschema:attA ] ;
        sh:filterShape [
          a sh:NodeShape ;
          sh:maxExclusive 5
        ]
      ]]
    )
  ] .
```

Given this RDF schema and RDF data:

```
exschema:ClassA a owl:Class .
exschema:ClassB a owl:Class .

exschema:attA a owl:DataProperty ;
  rdfs:range xsd:integer ;
  rdfs:domain exschema:ClassA .

exschema:role a owl:ObjectProperty ;
  rdfs:range exschema:ClassB ;
  rdfs:domain exschema:ClassA .

exschema:attB a owl:DataProperty ;
  rdfs:range xsd:integer ;
  rdfs:domain exschema:ClassB .

ex:A_1
  a exschema:ClassA ;
  exschema:attA 1 ;
  exschema:attA 4 ;
  exschema:role ex:B_1 ;
  exschema:role ex:B_2 .

ex:B_1
  a exschema:ClassB ;
  exschema:attB 6 .

ex:B_2
  a exschema:ClassB ;
  exschema:attB 4 .

ex:A_2
  a exschema:ClassA ;
  exschema:attA 10 ;
  exschema:role ex:B_1 .

ex:A_3
  a exschema:ClassA .
```

Then the resources `ex:A_1` and `ex:B_1` are marked as invalid, because:

- `ex:A_1` has `exschema:role ex:B_1`, with `ex:B_2 exschema:attB 4` - that validates the condition that all `exschema:attB` of `exschema:role` must have a value that is greater than 5.
- `ex:A_2` has `exschema:attA 10` - which validates the condition that all `exschema:attA` must have a value that is less than 5.

8.4.6.3. Translation using SPARQL queries, embedded in SHACL constructs

In this approach, a single SPARQL 1.1 query would be used to encode a constraint.

NOTE

A SPARQL translation of OCL language constructs was investigated because SPARQL is a powerful RDF query language, which has been standardized by W3C and which has a number of implementations. SPARQL-based SHACL constraints and constraint components are part of SHACL SPARQL, a standardized extension of SHACL Core. SPARQL queries can also be used in certain kinds of node expressions (*SPARQL ASK and SPARQL SELECT expressions*), which are defined by SHACL Advanced Features.

A benefit of this approach is that SPARQL naturally supports variables, which is something that the other two approaches do not support. SHACL SPARQL pre-binds the focus node to the variable $this, so there is a direct translation of the OCL variable *self*, even if used in some location within an OCL expression other than the start of that expression.

A major issue with this approach is that SPARQL does not have a language construct to express universal quantification, which would be needed to translate the OCL iterator call forAll() - which is an important OCL construct to ensure that property values satisfy certain conditions.

In some cases, it is possible to express the intent of a universal quantification in SPARQL. For example, a chain of forAll() calls, with only the last defining some condition on its variable, and no use of variables from the previous forAll() calls, can be translated using a single property path, with FILTER condition on that variable. The OCL constraint `inv: self.prop1→forAll(v1|v1.prop2→forAll(v2|v2.prop3 < 5))` can be translated to the following SPARQL query:

```
SELECT *
WHERE {
 ?fn a exschema:ClassA .
 ?fn exschema:prop1/exschema:prop2/exschema:prop3 ?val .
 FILTER (?val >= 5)
}
```

Another example looks for values of the forAll() iterator variable that do not satisfy the condition defined by the iterator, and checks if any such value exists. The OCL constraint `inv: self→forAll(a|a.attA < 5)` can be translated to the following SPARQL query:

```
SELECT *
WHERE {
 ?fn a exschema:ClassA .
 OPTIONAL {
   ?fn exschema:attA ?a .
   FILTER (?a >= 5)
 }
 FILTER (bound(?a))
}
```

NOTE	When the query is used in a SHACL SPARQL constraint component or expression, `?fn a exschema:ClassA .` can be omitted, since the SHACL shape would identify the focus nodes to be checked by the query (e.g., using the target predicate `sh:targetClass exschema:ClassA`). Variable `?fn` would then be replaced by variable `$this`.

These examples only cover specific situations. There is no guarantee that a solution can be found for any kind of universal quantification that occurs in an OCL expression.

8.4.6.4. SHACL Implementations for specific NAS OCL Constraints

Annex B.2 of the OGC Testbed-14: Application Schema-based Ontology Development Engineering Report [http://docs.opengeospatial.org/per/18-032r2.html] documents seven OCL constraints that cannot be expressed with OWL. The following subsections document the results of the analysis on translating these constraints to SHACL-based validation instructions.

NOTE	The translations use SPARQL ASK queries, embedded in SHACL node expressions (which are defined in SHACL Advanced Features [https://w3c.github.io/shacl/shacl-af/#ask]). They have been tested with TopBraidComposer Maestro Edition 6.3.2.

8.4.6.4.1. Property Co-constraint (related entity, numeric comparison)

```
inv: (width.valueOrReason.value->notEmpty() and
runwayDirection.runway.width.valueOrReason.value->notEmpty()) implies
((width.valueOrReason.value.lowerValue >=
runwayDirection.runway.width.valueOrReason.value.lowerValue) or
(width.valueOrReason.value.upperValue >=
runwayDirection.runway.width.valueOrReason.value.upperValue))
```

- Class in which the example constraint is defined: Stopway
- Constraint description: The Width attribute interval value of a stopway must equal or exceed the width interval value of its associated runway.

The constraint can be expressed using the following SHACL shape:

```
exshacl:WidthConsRunwayWidth
  a sh:NodeShape ;
  sh:targetClass exschema:Stopway ;
  sh:message "The Width attribute interval value of a stopway must equal or exceed
the width interval value of its associated runway.";
  sh:expression [
    sh:prefixes [
      sh:declare [
        sh:prefix "rdf" ;
        sh:namespace "http://www.w3.org/1999/02/22-rdf-syntax-ns#"^^xsd:anyURI ;
      ] ;
      sh:declare [
```

```
        sh:prefix "rdfs" ;
        sh:namespace "http://www.w3.org/2000/01/rdf-schema#"^^xsd:anyURI ;
      ] ;
      sh:declare [
        sh:prefix "exschema" ;
        sh:namespace "http://example.org/shacl/schema/"^^xsd:anyURI ;
      ]
    ] ;
    sh:ask """
      ASK {
        OPTIONAL {
          ?this exschema:width/exschema:valueOrReason/exschema:value ?v1 .
          ?this
```

```
exschema:runwayDirection/exschema:runway/exschema:width/exschema:valueOrReason/exschem
a:value ?v2 .
        }
        FILTER (bound(?v1) && bound(?v2))
        ?v1 exschema:lowerValue ?v1low .
        ?v1 exschema:upperValue ?v1up .
        ?v2 exschema:lowerValue ?v2low .
        ?v2 exschema:upperValue ?v2up .
        FILTER(?v1low >= ?v2low || ?v1up >= ?v2up)
      }
      """
  ] .
```

8.4.6.4.2. Property Co-constraint (type conditional, numeric comparison)

```
inv: (self.oclIsTypeOf(Runway) and self.length.valueOrReason.value->notEmpty() and
self.width.valueOrReason.value->notEmpty()) implies
((self.length.valueOrReason.value.lowerValue >
self.width.valueOrReason.value.lowerValue) or
(self.length.valueOrReason.value.upperValue >
self.width.valueOrReason.value.upperValue))
```

- Class in which the example constraint is defined: AerodromeMoveArea
- Constraint description: The Length attribute interval value of a runway must exceed its width interval value.

The constraint can be expressed using the following SHACL shape:

```
exshacl:LengthConsWidthRunway
  a sh:NodeShape ;
  sh:targetClass exschema:AerodromeMoveArea ;
  sh:message "The Length attribute interval value of a runway must exceed its width interval value.";
  sh:expression [
    sh:prefixes [
      sh:declare [
        sh:prefix "rdf" ;
        sh:namespace "http://www.w3.org/1999/02/22-rdf-syntax-ns#"^^xsd:anyURI ;
      ] ;
      sh:declare [
        sh:prefix "rdfs" ;
        sh:namespace "http://www.w3.org/2000/01/rdf-schema#"^^xsd:anyURI ;
      ] ;
      sh:declare [
        sh:prefix "exschema" ;
        sh:namespace "http://example.org/shacl/schema/"^^xsd:anyURI ;
      ]
    ] ;
    sh:ask """
      ASK {
        OPTIONAL {
          ?this rdf:type/rdfs:subClassOf* exschema:Runway .
          ?this exschema:length/exschema:valueOrReason/exschema:value ?v1 .
          ?this exschema:width/exschema:valueOrReason/exschema:value ?v2 .
        }
        FILTER (bound(?v1) && bound(?v2))
        ?v1 exschema:lowerValue ?v1low .
        ?v1 exschema:upperValue ?v1up .
        ?v2 exschema:lowerValue ?v2low .
        ?v2 exschema:upperValue ?v2up .
        FILTER(?v1low > ?v2low || ?v1up > ?v2up)
      }
      """
  ] .
```

NOTE The constraint is defined on UML type *AerodromeMoveArea*. However, the constraint is really only applicable if *self* is of type *Runway*. Thus, the OCL constraint could be moved to type *Runway*, and the first pre-condition (`self.oclIsTypeOf(Runway)`) be omitted. Then the translation would be the same as that for Property Co-constraint (related entity, numeric comparison).

8.4.6.4.3. Property Valid Numeric Interval

```
inv: length.valueOrReason.value->notEmpty() implies
(length.valueOrReason.value.lowerValue <= length.valueOrReason.value.upperValue) and
((length.valueOrReason.value.lowerValue = length.valueOrReason.value.upperValue)
implies (length.valueOrReason.value.intervalClosureType =
IntervalClosureType::closedInterval)) and
((length.valueOrReason.value.intervalClosureType = IntervalClosureType::gtSemiInterval
or length.valueOrReason.value.intervalClosureType =
IntervalClosureType::gteSemiInterval) implies (length.valueOrReason.value.upperValue-
>isEmpty())) and ((length.valueOrReason.value.intervalClosureType =
IntervalClosureType::ltSemiInterval or length.valueOrReason.value.intervalClosureType
= IntervalClosureType::lteSemiInterval) implies
(length.valueOrReason.value.lowerValue->isEmpty()))
```

- Class in which the example constraint is defined: AccessZone
- Constraint description: The Length attribute value of the access zone must be a well-formed and numerically valid interval.

The constraint can be expressed using the following SHACL shape:

```
exshacl:LengthValidInterval
  a sh:NodeShape ;
  sh:targetClass exschema:AccessZone ;
  sh:message "The Length attribute value of the access zone must be a well-formed and numerically valid interval.";
  sh:expression [
    sh:prefixes [
      sh:declare [
        sh:prefix "rdf" ;
        sh:namespace "http://www.w3.org/1999/02/22-rdf-syntax-ns#"^^xsd:anyURI ;
      ] ;
      sh:declare [
        sh:prefix "rdfs" ;
        sh:namespace "http://www.w3.org/2000/01/rdf-schema#"^^xsd:anyURI ;
      ] ;
      sh:declare [
        sh:prefix "ex" ;
        sh:namespace "http://example.org/shacl/data#"^^xsd:anyURI ;
      ] ;
      sh:declare [
        sh:prefix "exschema" ;
        sh:namespace "http://example.org/shacl/schema/"^^xsd:anyURI ;
      ]
    ] ;
    sh:ask """
      ASK {
        OPTIONAL {
          ?this exschema:length/exschema:valueOrReason/exschema:value ?v1 .
          ?v1 exschema:lowerValue ?low .
          ?v1 exschema:upperValue ?up .
          ?v1 exschema:intervalClosureType ?ict .
        }
        FILTER ( !bound(?v1) || (
          ( bound(?low) && bound(?up) && ?low <= ?up ) &&
          ( !(?low = ?up) || (bound(?ict) && ?ict = ex:Code_closedInterval) ) &&
          ( !bound(?ict) || ( !(?ict = ex:Code_gtSemiInterval || ?ict = ex:Code_gteSemiInterval) || !bound(?up) ) ) &&
          ( !bound(?ict) || ( !(?ict = ex:Code_ltSemiInterval || ?ict = ex:Code_lteSemiInterval) || !bound(?low) ) )
        ) )
      }
      """
  ] .
```

8.4.6.4.4. Related Entity Property Numeric Range Restriction

```
inv: movementArea->notEmpty() implies movementArea-
>forAll(x|((x.highestElevation.valuesOrReason.values->forAll(e|((e.value >=
self.aerodromeElevation.valueOrReason.value - 60.96) and (e.value <=
self.aerodromeElevation.valueOrReason.value + 60.96))))))
```

- Class in which the example constraint is defined: Aerodrome
- Constraint description: The Highest Elevation(s) of all aerodrome movement areas at an aerodrome must not differ by more than 200 feet from the Aerodrome Elevation.

The constraint can be expressed using the following SHACL shape:

```
exshacl:AerodromeElevationConsMoveArea
  a sh:NodeShape ;
  sh:targetClass exschema:Aerodrome ;
  sh:message "The Highest Elevation(s) of all aerodrome movement areas at an aerodrome must not differ by more than 200 feet from the Aerodrome Elevation.";
  sh:expression [
    sh:prefixes [
      sh:declare [
        sh:prefix "rdf" ;
        sh:namespace "http://www.w3.org/1999/02/22-rdf-syntax-ns#"^^xsd:anyURI ;
      ] ;
      sh:declare [
        sh:prefix "rdfs" ;
        sh:namespace "http://www.w3.org/2000/01/rdf-schema#"^^xsd:anyURI ;
      ] ;
      sh:declare [
        sh:prefix "exschema" ;
        sh:namespace "http://example.org/shacl/schema/"^^xsd:anyURI ;
      ]
    ] ;
    sh:ask """
      ASK {
        OPTIONAL {
          ?this exschema:movementArea ?x .
        }
        FILTER ( !bound(?x) || (
          NOT EXISTS {
            ?x exschema:highestElevation/exschema:valuesOrReason/exschema:values ?e .
            ?e exschema:value ?v1 .
            ?this exschema:aerodromeElevation/exschema:valueOrReason/exschema:value ?v2 .
            FILTER ( !( (?v1 >= ?v2 - 60.96) && (?v1 <= ?v2 + 60.96) ) )
          }
        ))
      }
      """
  ] .
```

8.4.6.4.5. Special case #1

```
inv: self.oclIsTypeOf(TimePointInfo) /* Placeholder NULL Constraint */
```

- Class in which the example constraint is defined: TimePointInfo
- Constraint description: The Temporal Position attribute value of the time point information uses a Temporal Reference Frame that is specified using "http://api.nsgreg.nga.mil/codelist/CalendarSystem".

- Additional note on the constraint: The ShapeChange OCL parser is currently limited by the expressiveness of Schematron – which does not directly support pattern-based constraints; either a Java extension function with XPath 1.0 or the "matches" function in XPath 2.0 is required. An always-true OCL constraint is currently specified instead of the applicable pattern-based constraint.

At the moment, the OCL constraint given as example is a placeholder. The intended constraint can be expressed using a SHACL shape, given that the following assumptions are correct:

- The temporal position is defined as an RDFS/OWL class, where the temporal reference frame is defined via a property.
- The codelist http://api.nsgreg.nga.mil/codelist/CalendarSystem is encoded as an RDFS/OWL class, with code values represented as individuals of that class.

```
exshacl:TemporalReferenceFrame
  a sh:NodeShape ;
  sh:targetClass exschema:TimePointInfo ;
  sh:message """The Temporal Position attribute value of the time point information uses a Temporal Reference Frame that is specified using "http://api.nsgreg.nga.mil/codelist/CalendarSystem".""";
  sh:expression [
    sh:prefixes [
      sh:declare [
        sh:prefix "rdf" ;
        sh:namespace "http://www.w3.org/1999/02/22-rdf-syntax-ns#"^^xsd:anyURI ;
      ] ;
      sh:declare [
        sh:prefix "rdfs" ;
        sh:namespace "http://www.w3.org/2000/01/rdf-schema#"^^xsd:anyURI ;
      ] ;
      sh:declare [
        sh:prefix "exschema" ;
        sh:namespace "http://example.org/shacl/schema/"^^xsd:anyURI ;
      ]
    ] ;
    sh:ask """
      ASK {
        ?this exschema:temporalPosition/exschema:temporalReferenceFrame ?x .
        ?x rdf:type/rdfs:subClassOf* <http://api.nsgreg.nga.mil/codelist/CalendarSystem> .
      }
      """
  ] .
```

8.4.6.4.6. Special case #2

```
inv: self.oclIsTypeOf(TM_Period) /* Placeholder NULL Constraint */
```

- Class in which the example constraint is defined: TM_Period
- Constraint description: The temporal position of the beginning of the period must be less than (i.e., earlier than) the temporal position of the end of the period.
- Additional note on the constraint: ISO 19108 expresses this constraint loosely as { self.begin.position < self.end.position}. The comparison operator is, however, not directly applicable to values of the complex type TM_Position; instead, a set of discriminated comparisons are required between values of its various alternate datatypes: Date, Time, DateTime, and TM_TemporalPosition.

It is unclear if the intent - as explained in the constraint description - can be represented in SHACL. The RDFS/OWL definition of a TM_Period would need to be defined.

8.4.6.4.7. Related Datatype Use Required (at least one)

```
inv: HydroVertDimIntervalMeta.allInstances().soundingMetadata->exists(sm|sm=self) or
HydroVertDimMeta.allInstances().soundingMetadata->exists(sm|sm=self)
```

- Class in which the example constraint is defined: SoundingMetadata
- Constraint description: There exists at least one of the following associated datatypes: Hydrographic Vertical Dimension Interval with Metadata, or Hydrographic Vertical Dimension or Reason; with Metadata.

The constraint can be expressed using the following SHACL shape:

```
exshacl:SoundingMetadataUsed
  a sh:NodeShape ;
  sh:targetClass exschema:SoundingMetadata ;
  sh:message "There exists at least one of the following associated datatypes:
Hydrographic Vertical Dimension Interval with Metadata, or Hydrographic Vertical
Dimension or Reason; with Metadata.";
  sh:expression [
    sh:prefixes [
      sh:declare [
        sh:prefix "rdf" ;
        sh:namespace "http://www.w3.org/1999/02/22-rdf-syntax-ns#"^^xsd:anyURI ;
      ] ;
      sh:declare [
        sh:prefix "rdfs" ;
        sh:namespace "http://www.w3.org/2000/01/rdf-schema#"^^xsd:anyURI ;
      ] ;
      sh:declare [
        sh:prefix "exschema" ;
        sh:namespace "http://example.org/shacl/schema/"^^xsd:anyURI ;
      ]
    ] ;
    sh:ask """
      ASK {
        {
          ?x rdf:type/rdfs:subClassOf* exschema:HydroVertDimIntervalMeta .
          FILTER EXISTS {
            ?x exschema:soundingMetadata ?this .
          }
        } UNION {
          ?y rdf:type/rdfs:subClassOf* exschema:HydroVertDimMeta .
          FILTER EXISTS {
            ?y exschema:soundingMetadata ?this .
          }
        }
      }
      """
  ] .
```

8.5. Summary

Validation of Linked Data means checking the logical consistency of the data and checking that the data satisfies a set of structural constraints. Data consistency can be checked with so-called reasoners, e.g., an OWL reasoner, based upon a set of vocabularies (such as OWL ontologies). Structural constraints can be checked using the Shapes Constraint Language (SHACL).

This chapter provides an introduction of SHACL. It gives an overview of specifications related to SHACL, together with their standardization status. It also explains core SHACL language features, and how validation of linked data is performed using SHACL shapes.

The SHACL Conversion Rules section explains in detail how an application schema - such as the NAS - can be encoded as a set of SHACL shapes. The conversion rules are designed to support data structures resulting from an OWL encoding (based upon the UML to RDF/OWL/SKOS [http://docs.opengeospatial.org/per/16-020.html#section_rdf] conversion rules) but also from a JSON encoding (based upon the UML to JSON Schema conversion rules), assuming a JSON-LD based transformation of JSON to linked data. Conversion rules have been defined for almost all elements of an application schema. The only exceptions are Property Metadata Stereotype, and OCL Constraints. No (full) SHACL conversion could be found for these model elements. During the analysis of OCL constraints in UGAS-2020, an attempt was therefore made to find SHACL-based translations for the NAS OCL expressions that cannot be translated with OWL, as determined by an analysis conducted in OGC Testbed-14 [http://docs.opengeospatial.org/per/18-032r2.html#RDF_OCLConstraints]. It turned out that most of these OCL expressions can be expressed using SHACL in combination with SPARQL queries. Using OWL and SHACL translations, the knowledge encoded in NAS OCL constraints can therefore be brought into the linked data technology space, in a way that software can actually use it (for linked data validation and inferencing).

So far, only conversion rules to produce RDFS/OWL/SKOS vocabularies, existed. These rules have been used to generate the NSG Enterprise Ontology (NEO) from the NAS. The main purpose of such vocabularies is to support inferencing, i.e., the generation of new information/intelligence. They also support identifying inconsistencies in linked data, but that is only one part of linked data validation. The other part, i.e., checking structural constraints, can now also be achieved - using the SHACL Conversion Rules documented in this chapter. Applying these rules to the NAS, SHACL shapes could be produced for validating NEO data.

NOTE	Only the development of SHACL conversion rules was in scope of the UGAS-2020 Pilot, not the implementation of these rules. The implementation was postponed to a future activity.

Chapter 9. Generating OpenAPI definitions from an application schema in UML

9.1. Introduction

The OpenAPI 3.0 specification [https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.2.md] defines a standard, programming language-agnostic interface description for Web APIs, which allows both humans and computers to discover and understand the capabilities of a Web API.

An OpenAPI definition consists of multiple sections:

- info (API metadata);
- servers (base URLs of servers supporting the API);
- paths (resources and their HTTP methods);
- tags (used to group path entries);
- security (security requirements relevant for deployment); and
- components (path/query parameters, schemas, responses).

The schemas sub-section specifies request and response schemas. For data resources, these schemas reuse content schemas, for example, of features in an application schema. Schemas are specified using JSON Schema, either expressed in JSON or YAML ("YAML Ain't Markup Language").

The methodology for deriving technology-specific encodings of conceptual schemas in UML based on the General Feature Model that has been implemented in ShapeChange, therefore, has direct application to the generation of OpenAPI content schemas. Chapter UML to JSON Schema Encoding Rule specifies how conceptual schemas in UML are converted to JSON schemas.

A similar methodology can be used to generate documentation for Web APIs.

An example application of this method has been prototyped in 2018 using ShapeChange to derive configurations for ldproxy (OGC reference implementation for OGC API - Features - Part 1: Core) that implement a Web API based on early drafts of the OGC API Features standard and PostgreSQL/PostGIS databases (see https://shapechange.net/targets/ldproxy/). The non-content sections of the OpenAPI definition are determined by the OGC API specifications (e.g., paths, parameters) and deployment specific configurations (e.g., info, servers, security).

Building blocks implemented by an API instance that are based on the OGC API specifications are determined by the selection of conformance classes that a given API should support (exclusive of any consideration for tool-specific extensions).

The pre-defined building blocks from the OGC API specifications may be extended or changed, too, as described in OGC API - Features - Part 1: Core, sub-clause 5.5.3 [http://docs.opengeospatial.org/is/17-069r3/17-069r3.html#_references_to_openapi_components_in_normative_statements].

The scope of the task in the UGAS-2020 pilot was restricted to the minimal solution for ShapeChange-based generation of OpenAPI definitions for a Web API implementing OGC API

Features parts 1 and 2 for sections other than for content schemas.

9.2. Scenario

In order to illustrate the analysis and discuss options, the following scenario will be used.

An organization has a feature dataset according to the application schema shown in Figure 16 and wants to share the data through a Web API that conforms to the OGC API Features standards (parts 1 and 2). From part 1, the requirements classes "core", "geojson", "html" and "oas30" are needed.

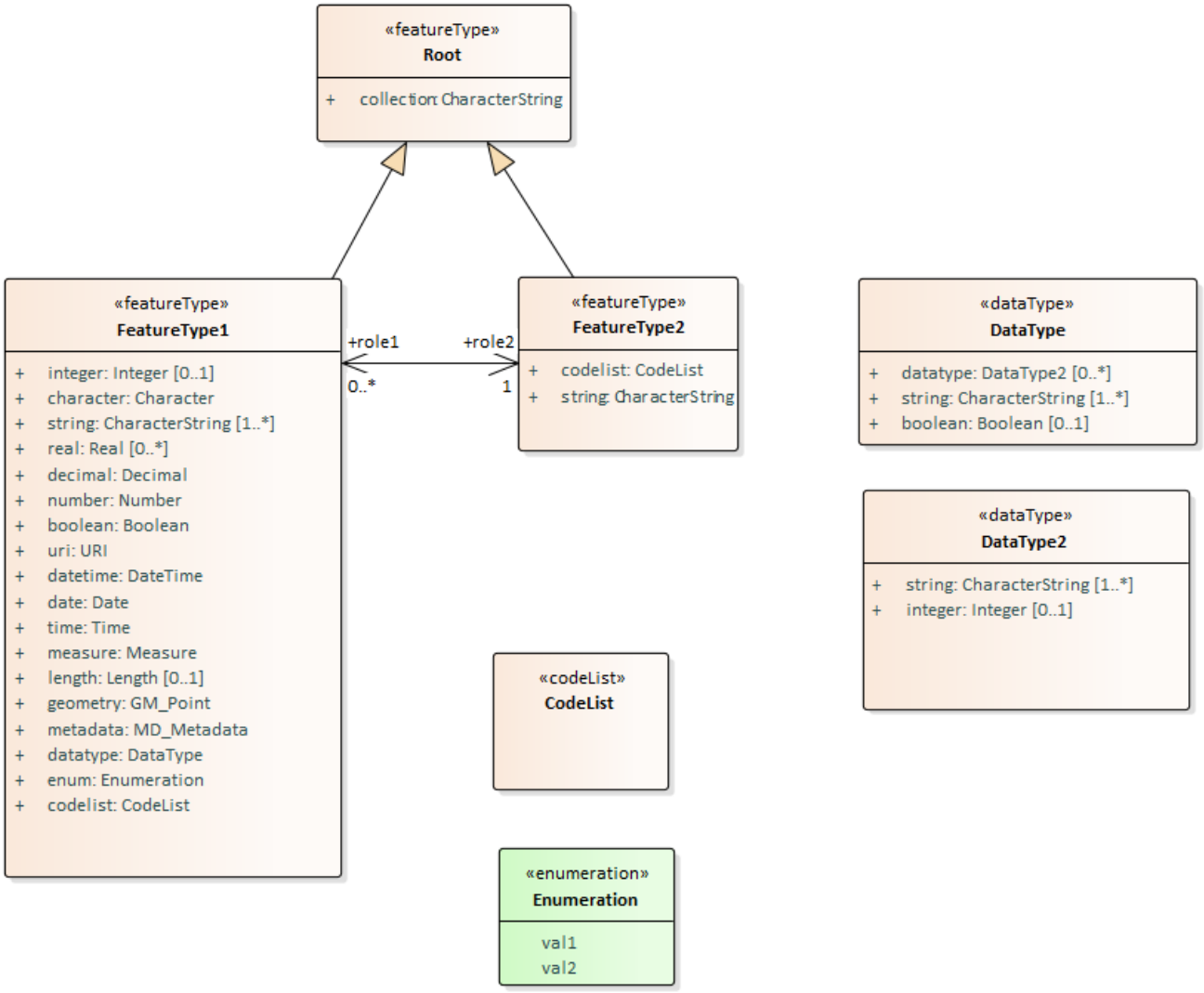

Figure 16. A simple application schema

NOTE The schema is one of the application schemas used in the ShapeChange unit tests.

The organization follows a design-first approach to its Web APIs.

NOTE	"Design-first" and "code-first" are opposite ends of the spectrum how to develop APIs. The OGC API standards development also follows a design-first approach (it is the design that is standardized and code is eventually developed implementing the design), but the design is informed and verified by running code from the beginning. There are numerous blog posts about the topic, for example, from Swagger [https://swagger.io/blog/api-design/design-first-or-code-first-api-development/], APIs you won't hate [https://apisyouwonthate.com/blog/api-design-first-vs-code-first], or Light [https://www.networknt.com/design/design-first/].

For sharing the feature dataset, the organization wants:

- an API that conforms to the relevant standards,
- customized to fit their needs, and
- managed consistently with the dataset.

Since the application schema of the feature dataset is managed in an application schema in UML and various implementation schemas (e.g., SQL DDL, XML, JSON) are already derived from that model in a model-driven workflow, the idea is to explore generating also an OpenAPI definition for the Web API as another implementation schema. A goal of this would be to limit (unnecessary) variations in the Web APIs of the organization for sharing spatial data.

This OpenAPI definition is then used to develop / configure / deploy the Web API. There are two general options, using the API definition as a blueprint.

1. Develop software for the Web API. Code generation from the OpenAPI definition may be used as a starting point.
2. Use an existing software tool that implements the OGC API Features standards and use the configuration tools of the software.

The approach that is chosen will depend on the specific needs of the organization.

The UGAS-2020 pilot has considered limited customization options that are already discussed in the standards.

- From the two approaches documented in OGC API Features Part 1 (use "{collectionId}" as a path parameter or use explicit paths for each collection), the second option is mainly relevant so that the JSON Schema of the application schema can be used in the OpenAPI definition.
- Support for an additional query parameter to override content negotiation headers. In this case, a parameter `f` with an enum value for the media type of the response content.
- Support for a query parameter to filter against a feature property [http://docs.opengeospatial.org/is/17-069r3/17-069r3.html#_parameters_for_filtering_on_feature_properties]. In this case, a parameter `string` is used for feature collection `FeatureType2`.
- All collections will support the coordinate reference systems CRS84 (OGC), 4326 and 3395 (both EPSG).

Each feature type will be published in the Web API as a Collection.

NOTE	A more comprehensive UML profile for modeling custom Web APIs from OGC API building blocks is in scope of OGC Testbed 16.

NOTE	The solution that was developed in the pilot is limited to supporting JSON schemas using the GeoJSON encoding rule of the JSON Schema target. Other JSON encodings for features are out-of-scope.

9.3. Analysis and design

9.3.1. Analyze the target OpenAPI definition

9.3.1.1. Starting with an OpenAPI template

This section documents the OpenAPI definition of the two APIs described in the Scenario, and how it can be constructed. The analysis was performed by deconstructing the target OpenAPI definition into building blocks for each conformance class and the additional options, and analyzing how the building blocks can be merged into a complete and consistent OpenAPI definition.

The JSON in Listing 75 consists of the minimal JSON structure every OpenAPI definition must have.

Listing 75. Minimal starting point of an OpenAPI definition

```
{
  "openapi": "3.0.2",
  "info": {
    "title": "a title of the API",
    "version": "a version identifier, e.g. 1.0.0"
  },
  "paths": {}
}
```

However, every deployed API should include some of the optional elements, so it is useful to already include them in the default template, to be used in the ShapeChange OpenAPI target. These are:

- a detailed description of the API;
- contact information;
- licensing information;
- information about the base URI(s) of the API; and
- tags to structure the resources, e.g., in an API documentation.

The result is a code block as shown in Listing 76. The texts are placeholders and need to be updated for each API.

NOTE	The ShapeChange OpenAPI target supports the use of custom templates, with additional information, pre-filled information (e.g., contact details), etc., so that organisations may adjust the processing to their needs.

Listing 76. Potential default template for an OpenAPI definition

```
{
  "openapi": "3.0.2",
  "info": {
    "title": "a title of the API",
    "version": "a version identifier, e.g. 1.0.0",
    "description": "a longer description what the API does, supports Markdown
markup",
    "contact": {
      "name": "name of the organisation",
      "email": "info@example.org",
      "url": "https://example.org/"
    },
    "license": {
      "name": "name of the license of the API, e.g. CC-BY 4.0 license",
      "url": "https://creativecommons.org/licenses/by/4.0/"
    }
  },
  "servers": [
    {
      "url": "https://data.example.org/"
    }
  ],
  "tags": [
    {
      "name": "Capabilities",
      "description": "essential characteristics of this API"
    },
    {
      "name": "Data",
      "description": "access to data (features)"
    }
  ],
  "paths": {}
}
```

9.3.1.2. Add support for conformance class "Core"

The next step is to add the building blocks for the conformance class "Core" of OGC API - Features - Part 1: Core.

Each set of connected building blocks is specified as an incomplete OpenAPI definition in JSON, a JSON overlay. Listing 77 provides an example. General texts have been added that can be kept as is.

When merging such a JSON document into the current OpenAPI definition, the rules specified in

JSON Merge Patch (RFC 7396) [https://tools.ietf.org/html/rfc7396] are applied. The only exception are query parameter arrays, where the current array value is not replaced, but the items in the array are appended to the current list of values. This exception is not yet relevant in this merge, but will be necessary in a later merge step.

Listing 77. Building blocks for Core

```
{
  "paths": {
    "/": {
      "get": {
        "tags": [
          "Capabilities"
        ],
        "summary": "landing page",
        "description": "The landing page provides links to the API definition, the conformance declaration and to the feature collections of this dataset.",
        "operationId": "getLandingPage",
        "responses": {
          "200": {
            "$ref": "#/components/responses/LandingPage"
          },
          "500": {
            "$ref": "#/components/responses/ServerError"
          }
        }
      }
    },
    "/conformance": {
      "get": {
        "tags": [
          "Capabilities"
        ],
        "summary": "information about specifications that this API conforms to",
        "description": "A list of all conformance classes specified in a standard that the API conforms to.",
        "operationId": "getConformanceDeclaration",
        "responses": {
          "200": {
            "$ref": "#/components/responses/ConformanceDeclaration"
          },
          "500": {
            "$ref": "#/components/responses/ServerError"
          }
        }
      }
    },
    "/collections": {
      "get": {
        "tags": [
          "Capabilities"
```

```
        ],
        "summary": "the feature collections",
        "description": "Fetch the feature collections in the dataset.",
        "operationId": "getCollections",
        "responses": {
          "200": {
            "$ref": "#/components/responses/Collections"
          },
          "500": {
            "$ref": "#/components/responses/ServerError"
          }
        }
      }
    },
    "/collections/{collectionId}": {
      "get": {
        "tags": [
          "Capabilities"
        ],
        "summary": "the feature collection '{collectionId}'",
        "description": "Fetch the feature collection '{collectionId}'.",
        "operationId": "getCollection_{collectionId}",
        "parameters": [
          {
            "$ref": "#/components/parameters/collectionId"
          }
        ],
        "responses": {
          "200": {
            "$ref": "#/components/responses/Collection"
          },
          "404": {
            "$ref": "#/components/responses/NotFound"
          },
          "500": {
            "$ref": "#/components/responses/ServerError"
          }
        }
      }
    },
    "/collections/{collectionId}/items": {
      "get": {
        "tags": [
          "Data"
        ],
        "summary": "fetch features in the feature collection '{collectionId}'",
        "description": "Fetch features in the feature collection '{collectionId}'.
The features included in the response are determined by the server based on the
query parameters of the request. To support access to larger collections without
overloading the client, the API supports paged access with links to the next page,
if more features are selected that the page size. The `bbox` and `datetime`
```

```
    parameter can be used to select only a subset of the features in the collection
(the features that are in the bounding box or date-time interval). The `bbox`
parameter matches all features in the collection that are not associated with a
location, too. The `datetime` parameter matches all features in the collection that
are not associated with a time stamp or interval, too. The `limit` parameter may be
used to control the maximum number of the selected features that should be returned in
the response, the page size. Each page may include information about the number of
selected and returned features (`numberMatched` and `numberReturned`) as well as a
link to support paging (link relation type `next`).",
 90         "operationId": "getFeatures_{collectionId}",
 91         "parameters": [
 92           {
 93             "$ref": "#/components/parameters/collectionId"
 94           },
 95           {
 96             "$ref": "#/components/parameters/limit"
 97           },
 98           {
 99             "$ref": "#/components/parameters/bbox"
100           },
101           {
102             "$ref": "#/components/parameters/datetime"
103           }
104         ],
105         "responses": {
106           "200": {
107             "$ref": "#/components/responses/Features"
108           },
109           "400": {
110             "$ref": "#/components/responses/InvalidParameter"
111           },
112           "404": {
113             "$ref": "#/components/responses/NotFound"
114           },
115           "500": {
116             "$ref": "#/components/responses/ServerError"
117           }
118         }
119       }
120     },
121     "/collections/{collectionId}/items/{featureId}": {
122       "get": {
123         "tags": [
124           "Data"
125         ],
126         "summary": "fetch a single feature in the feature collection
'{collectionId}'",
127         "description": "Fetch the feature with id `featureId`.",
128         "operationId": "getFeature_{collectionId}",
129         "parameters": [
130           {
```

```
            "$ref": "#/components/parameters/collectionId"
          },
          {
            "$ref": "#/components/parameters/featureId"
          }
        ],
        "responses": {
          "200": {
            "$ref": "#/components/responses/Feature"
          },
          "404": {
            "$ref": "#/components/responses/NotFound"
          },
          "500": {
            "$ref": "#/components/responses/ServerError"
          }
        }
      }
    }
  },
  "components": {
    "parameters": {
      "collectionId": {
        "name": "collectionId",
        "in": "path",
        "description": "local identifier of a collection",
        "required": true,
        "schema": {
          "type": "string"
        }
      },
      "featureId": {
        "name": "featureId",
        "in": "path",
        "description": "local identifier of a feature",
        "required": true,
        "schema": {
          "type": "string"
        }
      },
      "bbox": {
        "name": "bbox",
        "in": "query",
        "description": "Only features that have a geometry that intersects the
    bounding box are selected.\nThe bounding box is provided as four or six numbers,
    depending on whether the coordinate reference system includes a vertical axis
    (height or depth):\n\n* Lower left corner, coordinate axis 1\n* Lower left corner,
    coordinate axis 2\n* Minimum value, coordinate axis 3 (optional)\n* Upper right
    corner, coordinate axis 1\n* Upper right corner, coordinate axis 2\n* Maximum
    value, coordinate axis 3 (optional)\n\nThe coordinate reference system of the
    values is WGS 84
```

```
    longitude/latitude\n(http://www.opengis.net/def/crs/OGC/1.3/CRS84) unless a
different coordinate reference system is specified in the parameter `bbox-crs`.\n\nFor
WGS 84 longitude/latitude the values are in most cases the sequence of\nminimum
longitude, minimum latitude, maximum longitude and maximum latitude. However, in cases
where the box spans the antimeridian the first value (west-most box edge) is larger
than the third value (east-most box edge).\n\nIf the vertical axis is included, the
third and the sixth number are the bottom and the top of the 3-dimensional bounding
box.",
175         "required": false,
176         "style": "form",
177         "explode": false,
178         "schema": {
179           "minItems": 4,
180           "maxItems": 6,
181           "type": "array",
182           "items": {
183             "type": "number"
184           }
185         }
186       },
187       "datetime": {
188         "name": "datetime",
189         "in": "query",
190         "description": "Either a date-time or an interval, open or closed. Date
and time expressions adhere to RFC 3339. Open intervals are expressed using double-
dots. Examples:\n\n* A date-time: \"2018-02-12T23:20:50Z\"\n* A closed interval:
\"2018-02-12T00:00:00Z/2018-03-18T12:31:12Z\"\n* Open intervals: \"2018-02-
12T00:00:00Z/..\" or \"../2018-03-18T12:31:12Z\"\n\nOnly features that have a temporal
property that intersects the value of `datetime` are selected.",
191         "required": false,
192         "style": "form",
193         "explode": false,
194         "schema": {
195           "type": "string"
196         }
197       },
198       "limit": {
199         "name": "limit",
200         "in": "query",
201         "description": "The optional limit parameter limits the number of items
that are presented in the response document. Only items are counted that are on the
first level of the collection in the response document. Nested objects contained
within the explicitly requested items are not be counted.",
202         "required": false,
203         "style": "form",
204         "explode": false,
205         "schema": {
206           "minimum": 1,
207           "maximum": 10000,
208           "type": "integer",
209           "default": 10
```

```
        }
      }
    },
    "responses": {
      "LandingPage": {
        "description": "The landing page provides links to the API definition (link relation types `service-desc` and `service-doc`), the Conformance declaration (path `/conformance`, link relation type `conformance`), and to other resources."
      },
      "ConformanceDeclaration": {
        "description": "The URIs of all conformance classes supported by the API."
      },
      "Collections": {
        "description": "The feature collections shared by this API."
      },
      "Collection": {
        "description": "A feature collection."
      },
      "Features": {
        "description": "The response is a document consisting of features in the collection. The features included in the response are determined by the server based on the query parameters of the request."
      },
      "Feature": {
        "description": "The feature with id `{featureId}` in the feature collection with id `{collectionId}`"
      },
      "InvalidParameter": {
        "description": "A query parameter has an invalid value."
      },
      "NotFound": {
        "description": "The requested URI was not found."
      },
      "ServerError": {
        "description": "A server error occurred."
      }
    }
  }
}
```

Note that this overlay and the next ones implement the first approach for representing feature collections in the OpenAPI definition [http://docs.opengeospatial.org/is/17-069r3/17-069r3.html#two_appraoches_oas]. The second approach is analyzed and described in a later step.

NOTE In the UGAS-2020 pilot, a single JSON overlay document was used per conformance class. In a more structured approach, the overlay document could be modularized and the individual building blocks (responses, schemas, parameters, resources/paths, etc.) could be identified and managed separately.

For referencing building blocks from approved OGC API standards, it would be beneficial, if the components would not only be available as YAML files in the OGC schema repository [http://schemas.opengis.net/ogcapi/features/part1/1.0/openapi/], but also as JSON files.

9.3.1.3. In "Core", add support for the encoding conformance classes

The "Core" building blocks define the resources and their operations, but they do not define the content of the responses. This is added using the building blocks for the GeoJSON and HTML conformance classes, see Listing 78 and Listing 79. The HTML building blocks are simple because the HTML structure is not expressed; the payload is always a string.

Listing 78. Building blocks for GeoJSON

```
{
  "components": {
    "schemas": {
      "Collection" : {
        "required" : [ "id", "links" ],
        "type" : "object",
        "properties" : {
          "id" : {
            "type" : "string",
            "description" : "identifier of the collection used, for example, in
URIs"
          },
          "title" : {
            "type" : "string",
            "description" : "human readable title of the collection"
          },
          "description" : {
            "type" : "string",
            "description" : "a description of the features in the collection"
          },
          "links" : {
            "type" : "array",
            "items" : {
              "$ref" : "#/components/schemas/Link"
            }
          },
          "extent" : {
            "$ref" : "#/components/schemas/Extent"
          },
          "itemType" : {
            "type" : "string",
            "description" : "indicator about the type of the items in the
collection (the default value is 'feature')."
            "default" : "feature"
          },
          "crs" : {
            "type" : "array",
            "description" : "the list of coordinate reference systems supported by
```

```
    the service",
37              "items" : {
38                "type" : "string"
39              },
40              "default" : [
41                "http://www.opengis.net/def/crs/OGC/1.3/CRS84"
42              ]
43            }
44          }
45        },
46        "Collections" : {
47          "required" : [ "collections", "links" ],
48          "type" : "object",
49          "properties" : {
50            "links" : {
51              "type" : "array",
52              "items" : {
53                "$ref" : "#/components/schemas/Link"
54              }
55            },
56            "collections" : {
57              "type" : "array",
58              "items" : {
59                "$ref" : "#/components/schemas/Collection"
60              }
61            }
62          }
63        },
64        "ConformanceDeclaration" : {
65          "required" : [ "conformsTo" ],
66          "type" : "object",
67          "properties" : {
68            "conformsTo" : {
69              "type" : "array",
70              "items" : {
71                "type" : "string"
72              }
73            }
74          }
75        },
76        "Exception" : {
77          "required" : [ "code" ],
78          "type" : "object",
79          "properties" : {
80            "code" : {
81              "type" : "string",
82              "description" : "HTTP status code of the error (4xx or 5xx)"
83            },
84            "description" : {
85              "type" : "string",
86              "description" : "description of the error"
```

```
          }
        }
      },
      "Extent" : {
        "type" : "object",
        "properties" : {
          "spatial" : {
            "type" : "object",
            "properties" : {
              "bbox" : {
                "minItems" : 1,
                "type" : "array",
                "description" : "One or more bounding boxes that describe the spatial extent of the dataset.\nIn the Core only a single bounding box is supported. Extensions may support\nadditional areas. If multiple areas are provided, the union of the bounding\nboxes describes the spatial extent.",
                "items" : {
                  "maxItems" : 6,
                  "minItems" : 4,
                  "type" : "array",
                  "description" : "Each bounding box is provided as four or six numbers, depending on\nwhether the coordinate reference system includes a vertical axis\n(height or depth):\n\n* Lower left corner, coordinate axis 1\n* Lower left corner, coordinate axis 2\n* Minimum value, coordinate axis 3 (optional)\n* Upper right corner, coordinate axis 1\n* Upper right corner, coordinate axis 2\n* Maximum value, coordinate axis 3 (optional)\n\nThe coordinate reference system of the values is WGS 84 longitude/latitude\n(http://www.opengis.net/def/crs/OGC/1.3/CRS84) unless a different coordinate\nreference system is specified in `crs`.\n\nFor WGS 84 longitude/latitude the values are in most cases the sequence of\nminimum longitude, minimum latitude, maximum longitude and maximum latitude.\nHowever, in cases where the box spans the antimeridian the first value\n(west-most box edge) is larger than the third value (east-most box edge).\n\nIf the vertical axis is included, the third and the sixth number are\nthe bottom and the top of the 3-dimensional bounding box.\n\nIf a feature has multiple spatial geometry properties, it is the decision of the\nserver whether only a single spatial geometry property is used to determine\nthe extent or all relevant geometries.",
                  "example" : [ -180, -90, 180, 90 ],
                  "items" : {
                    "type" : "number"
                  }
                }
              },
              "crs" : {
                "type" : "string",
                "description" : "Coordinate reference system of the coordinates in the spatial extent\n(property `bbox`). The default reference system is WGS 84 longitude/latitude.",
                "default" : "http://www.opengis.net/def/crs/OGC/1.3/CRS84"
              }
            },
```

```
            "description" : "The spatial extent of the features in the
collection."
          },
          "temporal" : {
            "type" : "object",
            "properties" : {
              "interval" : {
                "minItems" : 1,
                "type" : "array",
                "description" : "One or more time intervals that describe the
temporal extent of the dataset. The value `null` is supported and indicates an
open time intervall.\nIn the Core only a single time interval is supported.
Extensions may support multiple intervals. If multiple intervals are provided, the
union of the intervals describes the temporal extent.",
                "items" : {
                  "maxItems" : 2,
                  "minItems" : 2,
                  "type" : "array",
                  "description" : "Begin and end times of the time interval. The
timestamps are in the coordinate reference system specified in `trs`. By default
this is the Gregorian calendar.",
                  "items" : {
                    "type" : "string",
                    "format" : "date-time",
                    "nullable" : true
                  }
                }
              },
              "trs" : {
                "type" : "string",
                "description" : "Coordinate reference system of the coordinates in
the temporal extent\n(property `interval`). The default reference system is the
Gregorian calendar.",
                "default" : "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian"
              }
            },
            "description" : "The temporal extent of the features in the
collection."
          }
        },
        "description" : "The extent of the features in the collection. In the Core
only spatial and temporal\nextents are specified. Extensions may add additional
members to represent other\nextents, for example, thermal or pressure ranges."
      },
      "Features" : {
        "required" : [ "features", "type" ],
        "type" : "object",
        "properties" : {
          "type" : {
            "type" : "string",
            "enum" : [ "FeatureCollection" ]
```

```
        },
        "features" : {
          "type" : "array",
          "items" : {
            "$ref" : "#/components/schemas/Feature"
          }
        },
        "links" : {
          "type" : "array",
          "items" : {
            "$ref" : "#/components/schemas/Link"
          }
        },
        "timeStamp" : {
          "$ref" : "#/components/schemas/TimeStamp"
        },
        "numberMatched" : {
          "$ref" : "#/components/schemas/NumberMatched"
        },
        "numberReturned" : {
          "$ref" : "#/components/schemas/NumberReturned"
        }
      }
    },
    "Feature" : {
      "required" : [ "geometry", "properties", "type" ],
      "type" : "object",
      "properties" : {
        "type" : {
          "type" : "string",
          "enum" : [ "Feature" ]
        },
        "geometry" : {
          "$ref" : "#/components/schemas/Geometry"
        },
        "properties" : {
          "type" : "object",
          "nullable" : true
        },
        "id" : {
          "oneOf" : [ {
            "type" : "string"
          }, {
            "type" : "integer"
          } ]
        },
        "links" : {
          "type" : "array",
          "items" : {
            "$ref" : "#/components/schemas/Link"
          }
```

```
          }
        }
      },
      "Geometry" : {
        "oneOf" : [ {
          "$ref" : "#/components/schemas/Point"
        }, {
          "$ref" : "#/components/schemas/MultiPoint"
        }, {
          "$ref" : "#/components/schemas/LineString"
        }, {
          "$ref" : "#/components/schemas/MultiLineString"
        }, {
          "$ref" : "#/components/schemas/Polygon"
        }, {
          "$ref" : "#/components/schemas/MultiPolygon"
        }, {
          "$ref" : "#/components/schemas/GeometryCollection"
        } ],
        "nullable": true
      },
      "GeometryCollection" : {
        "required" : [ "geometries", "type" ],
        "type" : "object",
        "properties" : {
          "type" : {
            "type" : "string",
            "enum" : [ "GeometryCollection" ]
          },
          "geometries" : {
            "type" : "array",
            "items" : {
              "$ref" : "#/components/schemas/Geometry"
            }
          }
        }
      },
      "LandingPage" : {
        "required" : [ "links" ],
        "type" : "object",
        "properties" : {
          "title" : {
            "type" : "string"
          },
          "description" : {
            "type" : "string"
          },
          "links" : {
            "type" : "array",
            "items" : {
              "$ref" : "#/components/schemas/Link"
```

```
          }
        }
      }
    },
    "LineString" : {
      "required" : [ "coordinates", "type" ],
      "type" : "object",
      "properties" : {
        "type" : {
          "type" : "string",
          "enum" : [ "LineString" ]
        },
        "coordinates" : {
          "minItems" : 2,
          "type" : "array",
          "items" : {
            "minItems" : 2,
            "type" : "array",
            "items" : {
              "type" : "number"
            }
          }
        }
      }
    },
    "Link" : {
      "required" : [ "href" ],
      "type" : "object",
      "properties" : {
        "href" : {
          "type" : "string"
        },
        "rel" : {
          "type" : "string"
        },
        "type" : {
          "type" : "string"
        },
        "hreflang" : {
          "type" : "string"
        },
        "title" : {
          "type" : "string"
        },
        "length" : {
          "type" : "integer"
        }
      }
    },
    "MultiLineString" : {
      "required" : [ "coordinates", "type" ],
```

```
      "type" : "object",
      "properties" : {
        "type" : {
          "type" : "string",
          "enum" : [ "MultiLineString" ]
        },
        "coordinates" : {
          "type" : "array",
          "items" : {
            "minItems" : 2,
            "type" : "array",
            "items" : {
              "minItems" : 2,
              "type" : "array",
              "items" : {
                "type" : "number"
              }
            }
          }
        }
      }
    },
    "MultiPoint" : {
      "required" : [ "coordinates", "type" ],
      "type" : "object",
      "properties" : {
        "type" : {
          "type" : "string",
          "enum" : [ "MultiPoint" ]
        },
        "coordinates" : {
          "type" : "array",
          "items" : {
            "minItems" : 2,
            "type" : "array",
            "items" : {
              "type" : "number"
            }
          }
        }
      }
    },
    "MultiPolygon" : {
      "required" : [ "coordinates", "type" ],
      "type" : "object",
      "properties" : {
        "type" : {
          "type" : "string",
          "enum" : [ "MultiPolygon" ]
        },
        "coordinates" : {
```

```
            "type" : "array",
            "items" : {
              "type" : "array",
              "items" : {
                "minItems" : 4,
                "type" : "array",
                "items" : {
                  "minItems" : 2,
                  "type" : "array",
                  "items" : {
                    "type" : "number"
                  }
                }
              }
            }
          }
        }
      },
      "NumberMatched" : {
        "minimum" : 0,
        "type" : "integer",
        "description" : "The number of features of the feature type that match the selection\nparameters like `bbox`."
      },
      "NumberReturned" : {
        "minimum" : 0,
        "type" : "integer",
        "description" : "The number of features in the feature collection.\n\nA server may omit this information in a response, if the information about the number of features is not known or difficult to compute.\n\nIf the value is provided, the value is identical to the number of items in the 'features' array."
      },
      "Point" : {
        "required" : [ "coordinates", "type" ],
        "type" : "object",
        "properties" : {
          "type" : {
            "type" : "string",
            "enum" : [ "Point" ]
          },
          "coordinates" : {
            "minItems" : 2,
            "type" : "array",
            "items" : {
              "type" : "number"
            }
          }
        }
      },
      "Polygon" : {
        "required" : [ "coordinates", "type" ],
```

```
        "type" : "object",
        "properties" : {
          "type" : {
            "type" : "string",
            "enum" : [ "Polygon" ]
          },
          "coordinates" : {
            "type" : "array",
            "items" : {
              "minItems" : 4,
              "type" : "array",
              "items" : {
                "minItems" : 2,
                "type" : "array",
                "items" : {
                  "type" : "number"
                }
              }
            }
          }
        }
      },
      "TimeStamp" : {
        "type" : "string",
        "description" : "This property indicates the time and date when the
response was generated.",
        "format" : "date-time"
      }
    },
    "responses": {
      "LandingPage": {
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/LandingPage"
            }
          }
        }
      },
      "ConformanceDeclaration": {
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/ConformanceDeclaration"
            }
          }
        }
      },
      "Collections": {
        "content": {
          "application/json": {
```

```
            "schema": {
              "$ref": "#/components/schemas/Collections"
            }
          }
        }
      },
      "Collection": {
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Collection"
            }
          }
        }
      },
      "Features": {
        "content": {
          "application/geo+json": {
            "schema": {
              "$ref": "#/components/schemas/Features"
            }
          }
        }
      },
      "Feature": {
        "content": {
          "application/geo+json": {
            "schema": {
              "$ref": "#/components/schemas/Feature"
            }
          }
        }
      },
      "InvalidParameter": {
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Exception"
            }
          }
        }
      },
      "NotFound": {
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Exception"
            }
          }
        }
      },
```

```
      "ServerError": {
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Exception"
            }
          }
        }
      }
    }
  }
}
```

Listing 79. Building blocks for HTML

```
{
  "components": {
    "schemas": {
      "htmlPage": {
        "type": "string"
      }
    },
    "responses": {
      "LandingPage": {
        "content": {
          "text/html": {
            "schema": {
              "$ref": "#/components/schemas/htmlPage"
            }
          }
        }
      },
      "ConformanceDeclaration": {
        "content": {
          "text/html": {
            "schema": {
              "$ref": "#/components/schemas/htmlPage"
            }
          }
        }
      },
      "Collections": {
        "content": {
          "text/html": {
            "schema": {
              "$ref": "#/components/schemas/htmlPage"
            }
          }
        }
      },
      "Collection": {
```

```
        "content": {
          "text/html": {
            "schema": {
              "$ref": "#/components/schemas/htmlPage"
            }
          }
        }
      },
      "Features": {
        "content": {
          "text/html": {
            "schema": {
              "$ref": "#/components/schemas/htmlPage"
            }
          }
        }
      },
      "Feature": {
        "content": {
          "text/html": {
            "schema": {
              "$ref": "#/components/schemas/htmlPage"
            }
          }
        }
      },
      "InvalidParameter": {
        "content": {
          "text/html": {
            "schema": {
              "$ref": "#/components/schemas/htmlPage"
            }
          }
        }
      },
      "NotFound": {
        "content": {
          "text/html": {
            "schema": {
              "$ref": "#/components/schemas/htmlPage"
            }
          }
        }
      },
      "ServerError": {
        "content": {
          "text/html": {
            "schema": {
              "$ref": "#/components/schemas/htmlPage"
            }
          }
        }
```

```
        }
      }
    }
  }
}
```

9.3.1.4. Conformance class "OpenAPI 3.0 Specification"

There is nothing to add to the OpenAPI definition for the OpenAPI 3.0 conformance class. The steps described in this analysis are constructed so that the resulting OpenAPI definition is conformant.

9.3.1.5. Add support for the conformance class "Coordinate Reference System by Reference"

The final conformance class to add is Coordinate Reference Systems by Reference. This adds additional query parameters and extends the Collection schema, but adds no new resource.

In this step and all other cases where query parameters are added, the special rule for merging arrays, which was mentioned in the beginning of the section, is necessary so that the additional query parameters are added and do not replace the previously defined parameters.

Listing 80. Building blocks for Coordinate Reference Systems

```
{
  "paths": {
    "/collections/{collectionId}/items": {
      "get": {
        "parameters": [
          {
            "$ref": "#/components/parameters/crs"
          },
          {
            "$ref": "#/components/parameters/bbox-crs"
          }
        ]
      }
    },
    "/collections/{collectionId}/items/{featureId}": {
      "get": {
        "parameters": [
          {
            "$ref": "#/components/parameters/crs"
          }
        ],
        "responses": {
          "400": {
            "$ref": "#/components/responses/InvalidParameter"
          }
        }
      }
    }
  },
```

```
  "components": {
    "schemas": {
      "Collections" : {
        "properties" : {
          "crs" : {
            "type" : "array",
            "items": {
              "type": "string"
            },
            "description" : "a list of CRS identifiers that are supported for more that one feature collection offered by the service"
          }
        }
      },
      "Collection" : {
        "properties" : {
          "storageCrs" : {
            "type" : "string",
            "description" : "the CRS identifier, from the list of supported CRS identifiers, that may be used to retrieve features from a collection without the need to apply a CRS transformation"
          }
        }
      }
    },
    "parameters": {
      "crs": {
        "name": "crs",
        "in": "query",
        "description": "The coordinate reference system of the response geometries. Default is WGS84 longitude/latitude (http://www.opengis.net/def/crs/OGC/1.3/CRS84).",
        "required": false,
        "style": "form",
        "explode": false,
        "schema": {
          "type": "string",
          "default": "http://www.opengis.net/def/crs/OGC/1.3/CRS84"
        }
      },
      "bbox-crs": {
        "name": "bbox-crs",
        "in": "query",
        "description": "The coordinate reference system of the bbox parameter. Default is WGS84 longitude/latitude (http://www.opengis.net/def/crs/OGC/1.3/CRS84).",
        "required": false,
        "style": "form",
        "explode": false,
        "schema": {
          "type": "string",
```

```
          "default": "http://www.opengis.net/def/crs/OGC/1.3/CRS84"
        }
      }
    }
  }
}
```

In the scenario, the API should support the following coordinate reference systems in all collections:

- http://www.opengis.net/def/crs/OGC/1.3/CRS84
- http://www.opengis.net/def/crs/EPSG/0/4326
- http://www.opengis.net/def/crs/EPSG/0/3395

This information is added via the building blocks shown in Listing 81.

Listing 81. Building blocks for constraining the coordinate reference systems that may be requested

```
{
  "components": {
    "parameters": {
      "crs": {
        "schema": {
          "enum": [
            "http://www.opengis.net/def/crs/OGC/1.3/CRS84",
            "http://www.opengis.net/def/crs/EPSG/0/4326",
            "http://www.opengis.net/def/crs/EPSG/0/3395"
          ]
        }
      },
      "bbox-crs": {
        "schema": {
          "enum": [
            "http://www.opengis.net/def/crs/OGC/1.3/CRS84",
            "http://www.opengis.net/def/crs/EPSG/0/4326",
            "http://www.opengis.net/def/crs/EPSG/0/3395"
          ]
        }
      }
    }
  }
}
```

NOTE

The building block shown in Listing 81 is automatically constructed by ShapeChange, using the CRS identifiers that are given in the @param XML attribute of the CRS conformance class element, which is contained in the ShapeChange configuration of the OpenAPI target.

9.3.1.6. Add support for additional query parameters before feature processing

Before feature specific changes are applied to the OpenAPI definition that has been built so far, overlays for adding further query parameters can be merged.

With the overlay in Listing 82, query parameter `f` - which is not specified in a conformance class - is added to the OpenAPI definition.

Listing 82. Building blocks for the `f` parameter (overriding the "Accept" header)

```
{
  "paths": {
    "/": {
      "get": {
        "parameters": [
          {
            "$ref": "#/components/parameters/f"
          }
        ],
        "responses": {
          "400": {
            "$ref": "#/components/responses/InvalidParameter"
          }
        }
      }
    },
    "/conformance": {
      "get": {
        "parameters": [
          {
            "$ref": "#/components/parameters/f"
          }
        ],
        "responses": {
          "400": {
            "$ref": "#/components/responses/InvalidParameter"
          }
        }
      }
    },
    "/collections": {
      "get": {
        "parameters": [
          {
            "$ref": "#/components/parameters/f"
          }
        ],
        "responses": {
          "400": {
            "$ref": "#/components/responses/InvalidParameter"
          }
```

```
        }
      }
    },
    "/collections/{collectionId}": {
      "get": {
        "parameters": [
          {
            "$ref": "#/components/parameters/f"
          }
        ],
        "responses": {
          "400": {
            "$ref": "#/components/responses/InvalidParameter"
          }
        }
      }
    },
    "/collections/{collectionId}/items": {
      "get": {
        "parameters": [
          {
            "$ref": "#/components/parameters/f"
          }
        ]
      }
    },
    "/collections/{collectionId}/items/{featureId}": {
      "get": {
        "parameters": [
          {
            "$ref": "#/components/parameters/f"
          }
        ],
        "responses": {
          "400": {
            "$ref": "#/components/responses/InvalidParameter"
          }
        }
      }
    }
  },
  "components": {
    "parameters": {
      "f": {
        "name": "f",
        "in": "query",
        "description": "The format of the response. If no value is provided, the
standard http rules apply, i.e., the accept header will be used to determine the
format. Allowed values are 'json' and 'html'.",
        "required": false,
        "style": "form",
```

```
        "explode": false,
        "schema": {
          "type": "string",
          "enum": [
            "json",
            "html"
          ]
        }
      }
    }
  }
}
```

NOTE	Essentially, the overlay to add the query parameter `f` is defined in the ShapeChange configuration of the OpenAPI target, in a `<QueryParameter>` element with @phase equal to `pre-feature-identification`. ShapeChange will automatically merge all query parameter overlays defined for that phase, before executing the steps described in the next section.

9.3.1.7. Feature type specific modifications

OGC API Features Part 1 [http://docs.opengeospatial.org/is/17-069r3/17-069r3.html#two_appraoches_oas] documents two general approaches for representing feature collections in the OpenAPI definition. The process for building an OpenAPI definition - with the processing steps taken so far described in the previous sections - now reaches a fork with two branches, one for each of the approaches.

- Following the instructions given in section Constrain the collection identifier values, the OpenAPI definition would use “{collectionId}” as a path parameter.
- Section Constrain the response schema for each feature collection., on the other hand, would create an OpenAPI definition with explicit feature collection paths.

As described in the Scenario section, the benefit of the second approach is that JSON Schema references can be given for each feature collection, which is why this approach has been implemented in UGAS-2020.

For both of the approaches, the feature collections that shall actually be published via the service described by the OpenAPI definition need to be identified. That is described in the first subsection.

9.3.1.7.1. Identifying the feature collections

The OpenAPI definition of an OGC API Features compliant service needs to identify the feature types for which the service has data.

Three different rules for identifying the feature types are available.

- *rule-openapi-cls-instantiable-feature-types*: Each instantiable feature type is selected, i.e., abstract feature types are skipped.
- *rule-openapi-cls-top-level-feature-types*: Each top-level feature type is selected, i.e., feature types with a supertype in the same application schema are skipped.

- *rule-openapi-all-explicit-collections*: The feature types are explicitly identified using the ShapeChange OpenAPI target parameter *collections*. This supports a tailored approach for more complex use cases, and also enables the specification of APIs supporting only a profile of the application schema.

9.3.1.7.2. Constrain the collection identifier values

In the first approach for representing feature collections in the OpenAPI definition [http://docs.opengeospatial.org/is/17-069r3/17-069r3.html#two_appraoches_oas] using the `collectionId` path parameter, the only information that is specific to the application schema that can be expressed in the API definition is to constrain the values of the `collectionId` parameter, to the names of the relevant feature types. For the Scenario, that would result in an overlay as shown in Listing 83.

Listing 83. Building blocks for constraining the collections that may be requested

```
{
  "components": {
    "parameters": {
      "collectionId": {
        "schema": {
          "enum": [
            "FeatureType1",
            "FeatureType2"
          ]
        }
      }
    }
  }
}
```

NOTE	As explained in the Scenario section, this approach has not been implemented in UGAS-2020, because it does not support inclusion of references to feature type specific JSON Schemas.

9.3.1.7.3. Constrain the response schema for each feature collection.

In the second approach for representing feature collections in the OpenAPI definition [http://docs.opengeospatial.org/is/17-069r3/17-069r3.html#two_appraoches_oas] using seperate Collection resources for each feature collection, the following additional steps are required.

1. For each feature type in the application schema:
   - Create a copy of the path "/collections/{collectionId}", replace all occurrences of "{collectionId}" with the feature type identifier, and remove the query parameter `"{$ref": "#/components/parameters/collectionId}"`.
   - Create a copy of the path "/collections/{collectionId}/items", change the "200"-response to `{ "$ref": "#/components/responses/Features_{collectionId}" }`, replace all occurrences of "{collectionId}" with the feature type identifier, and remove the query parameter `"{$ref": "#/components/parameters/collectionId}"`.

- Create a copy of the path "/collections/{collectionId}/items/{featureId}", change the "200"-response to `{ "$ref": "#/components/responses/Feature_{collectionId}" }`, replace all occurrences of "{collectionId}" with the feature type identifier, and remove the query parameter `"{$ref": "#/components/parameters/collectionId}"`.
- Create a copy of the response "Features" ("/components/responses/Features") and rename it to "Features_{collectionId}", where "{collectionId}" is replaced with the feature type identifier.
  - If the GeoJSON conformance class is applicable:
    - Change the schema of the "application/geo+json" response to `{ "$ref": "#/components/schemas/Features_{collectionId}" }` after replacing "{collectionId}" with the feature type identifier.
    - Create a copy of the schema "Features" ("/components/schemas/Features") and rename it to "Features_{collectionId}", where "{collectionId}" is replaced with the feature type identifier. Change value of "items" in the "features" property to `{ "$ref": "#/components/schemas/Feature_{collectionId}" }` after replacing "{collectionId}" with the feature type identifier.
  - No change necessary for the HTML conformance class.
- Create a copy of the response "Feature" ("/components/responses/Feature") and rename it to "Feature_{collectionId}", where "{collectionId}" is replaced with the feature type identifier.
  - If the GeoJSON conformance class is applicable:
    - Change the schema of the "application/geo+json" response to `{ "$ref": "#/components/schemas/Feature_{collectionId}" }` after replacing "{collectionId}" with the feature type identifier.
    - Create a new schema "Feature_{collectionId}", where "{collectionId}" is replaced with the feature type identifier, referencing the schema created by the JSON Schema target for the feature type. The schema reference is constructed as follows: `{jsonSchemasBaseLocation} + {jsonDirectory} + {jsonSchemasPathSeparator} + {jsonSchemaFileName} + {definitionsReference} + {featureTypeName}`, where:
      - *jsonSchemasBaseLocation* - Is configured via the ShapeChange OpenAPI target parameter with the same name (i.e., "jsonSchemasBaseLocation"), and identifies the base directory of the single location where all JSON Schemas for the feature types of the OpenAPI definition are stored.
      - *jsonDirectory* - Is as defined in the JSON Schema chapter, section Schema Identifier.
      - *jsonSchemasPathSeparator* - Is `\`, if the *jsonSchemasBaseLocation* contains a `\`, otherwise it is `/`.
      - *jsonSchemaFileName* - Defines the name of the JSON Schema file that contains the definition of the feature type. The file name is given by the value of tag "jsonDocument" of the package that owns the feature type, or the nearest ancestor package that defines such a value. However, if the application schema package is reached in the sequence of ancestor packages, and the application schema package does not define a value for the tag, then the name of the application schema is used, normalized by replacing all forward slashes and

spaces with an underscore, and appending ".json".

- *definitionsReference* - Is a fragment identifier with a JSON Pointer to reference the definitions section within the JSON Schema file. If the value of the ShapeChange OpenAPI target parameter *jsonSchemaVersion* is "2019-09", then the reference will be "#/$defs/", otherwise it will be "#/definitions/".

NOTE	The ShapeChange OpenAPI target parameter *jsonSchemaVersion* has the same values as in the ShapeChange JSON Schema target (see section JSON Schema Version). The parameter must reflect the version of the JSON Schemas created for the feature types. For an OpenAPI 3.0 definition, the *jsonSchemaVersion* must be set to "OpenApi30".

- *featureTypeName* - Is the name of the feature type.

- Again, no change necessary for the HTML conformance class.

2. Remove elements from the OpenAPI definition that are no longer used, merging the json document given in Listing 84.

NOTE	ShapeChange automatically creates this JSON object and merges it internally. The document does not need to be configured.

This removes:

- the paths with a `collectionId` path parameter from the definition;
- the `collectionId` parameter;
- the "Features" and "Feature" responses;
- the "Features" and "Feature" schema.

Listing 84. Building blocks to remove the generic resources and components

```
{
  "paths": {
    "/collections/{collectionId}": null,
    "/collections/{collectionId}/items": null,
    "/collections/{collectionId}/items/{featureId}": null
  },
  "components": {
    "parameters": {
      "collectionId": null
    },
    "schemas": {
      "Features": null,
      "Feature": null
    },
    "responses": {
      "Features": null,
      "Feature": null
    }
  }
}
```

NOTE	In the steps above it is said that the JSON schemas of the feature types that are generated by the ShapeChange JSON Schema target are referenced from the OpenAPI definition. Another option could also be to embed the schema in the OpenAPI definition.

WARNING	The JSON schemas for the feature types must be generated using the JSON Schema variant of the OpenAPI version. This is discussed in JSON Schema variants.

9.3.1.8. Add support for additional query parameters during finalization of the OpenAPI definition

With feature specific changes complete, the OpenAPI definition is ready for use. However, ShapeChange supports adding further query parameters to the OpenAPI definition while finalizing it. An additional query parameter is defined in the ShapeChange configuration of the OpenAPI target, in a `<QueryParameter>` element with @phase equal to `finalization`. That element also contains an overlay, which will be merged into the OpenAPI definition.

For example:

In the approach where the OpenAPI definition reflects the application schema of the dataset, additional query parameters can be added that act as filters [http://docs.opengeospatial.org/is/17-069r3/17-069r3.html#_parameters_for_filtering_on_feature_properties]. In the scenario, a parameter is added to filter the feature collection "FeatureType2" based on feature property "string". This is done using the overlay in Listing 85.

Listing 85. Building blocks for filtering features based on a property

```
{
  "paths": {
    "/collections/FeatureType2/items": {
      "get": {
        "parameters": [
          {
            "$ref": "#/components/parameters/string"
          }
        ]
      }
    }
  },
  "components": {
    "parameters": {
      "string": {
        "name": "string",
        "in": "query",
        "description": "Only return features where the property of the same name has the provided value. Default = return all features.",
        "required": false,
        "schema": {
          "type": "string"
        },
        "style": "form",
        "explode": false
      }
    }
  }
}
```

9.3.2. Design of a minimal OpenAPI target in ShapeChange

The design had to address two aspects.

- Which information should be added to the UML model and how?
- Which information should be specified in the ShapeChange configuration?

9.3.2.1. Additional information in the UML model

The term "UML model" is used in the chapter as a synonym for an Enterprise Architect project, e.g., an EAP file. The EAP file contains the application schema. Since the OpenAPI target needs to process the application schema to identify the list of feature collections (see section Identifying the feature collections) any additional information has to be added in the same UML model.

In general, additional information could be added in two ways:

- as new packages for the API definition(s), and/or

- via additional stereotypes and/or tagged values in the application schema.

In the UGAS-2020 pilot, no additional information has been added to the UML model in order to avoid constraints on the work in OGC Testbed 16, which has been based on the results of this pilot. However, the following adds some considerations.

- Representing the building blocks and conformance classes of the OGC API standards plus support typical extension patterns (like adding additional query parameters) likely requires the definition of a UML profile for OGC API standards. Existing standards, draft specification and custom extensions would then be modelled as UML packages in accordance with that UML profile.
- Such a UML profile probably should not be a general UML profile for Web APIs / HTTP (there are a number of attempts for this, e.g., an MDG for Enterprise Architect [http://www.aprocessgroup.com/products/sparx-enterprise-architect/premier-api-modeling-solution/]), but instead be a focused profile for the OGC API building blocks and custom extensions. The reason is that a key goal is simplified API management and minimizing the variations across all APIs in an organisation, not a new capability to model any Web API.
- It may also be useful to include some aspects in the application schema itself. An example could be a tagged value in feature type classes for the purpose of Identifying the feature collections or in an attribute to identify the property as queryable property.

9.3.2.2. Dependency on the JSON Schema target

The step Constrain the response schema for each feature collection. has a dependency on JSON schemas for the feature types generated by the JSON Schema target.

The dependency is minimal since the current design only needs to know the relative location where the JSON schema documents are generated and how they are structured. The OpenAPI target as designed in this pilot does not require access to the JSON schema documents itself. I.e., the order in which the targets are executed is not relevant and the JSON Schema target may be executed after the OpenAPI target.

NOTE

This is useful since currently ShapeChange does not have a mechanism to express additional dependencies to control the sequence in which targets are executed. The current ShapeChange processing model is as follows:

- an input model is loaded;
- it can be transformed in a number of ways, thus ultimately creating a set of models;
- targets can then be applied to selected models.

Processing occurs in a tree-like fashion.

- ShapeChange identifies the targets and transformations that shall be applied to the input model or a model that results from a transformation.
- For a given model, the associated targets will be executed first, and then the associated transformations.

- If more than one transformation is applied, then a copy of the model is created for use by each associated transformation.
- The input model is kept in memory until all associated transformations have been processed. Only then can the input model be released. ShapeChange tries to keep a low memory profile. Therefore it tries to minimize the number of models that are kept in memory.

As a consequence there are a number of considerations with respect to introducing dependencies between targets.

- A dependency from target A on target B may require a transformation to be executed multiple times, instead of only once - unless the result of a transformation is always stored, either in memory (which can be difficult for large models and the memory limitations of a 32-bit Java process) or in temporary files. The latter would require SCXML files to be written to and re-read from a temporary directory.
- A target configuration may have multiple inputs. A dependency from target A on target B may thus need to be qualified with a subset of the inputs from both targets. This would increase the complexity of creating a ShapeChange configuration.
- Dependencies between targets may create a dependency chain or even a tree. ShapeChange would have to detect a dependency loop, and prevent execution if one was detected, because it could never be satisfied.

Implementing target dependencies would be possible, but would require substantial effort, and would result in a significantly more complex processing model. Since it is not necessary for the OpenAPI target design, no changes to the processing model are foreseen. Even if a future extension of the OpenAPI target would require access to the JSON schema documents (e.g., to embed the schema definitions inline), this could also be addressed by two separate ShapeChange executions, the first generating the JSON schemas, and the second the OpenAPI definition.

9.3.2.3. ShapeChange configuration

Since no information will be added for the minimal OpenAPI target, all required input needs to be part of the ShapeChange configuration of the target. If a complete design is available in the future, e.g., from OGC Testbed 16, some of these input parameters could be removed.

A requirement for the configuration is to support the workflow and the configuration options described in the section Analyze the target OpenAPI definition. These are:

- a reference to a base OpenAPI template (local file or URI), with the default being the one shown in Listing 76;
- information needed to construct references to feature type specific JSON Schemas:
  - the base directory of the single location where all JSON Schemas for the feature types of the OpenAPI definition are stored;

    - the version of the JSON Schemas created for the feature types (for an OpenAPI 3.0 definition, the JSON Schemas should be OpenAPI 3.0 compatible);
- a set of well-known OGC API conformance classes, identified by their URIs;

NOTE	Only the core conformance class is required. The others are optional

    - Options for all conformance classes: a link to a JSON overlay document. Default values:
        - Core: Listing 77
        - GeoJSON: Listing 78
        - HTML: Listing 79
        - CRS: Listing 80
    - Options for Core:
        - Conversion rules to select the general strategy for determining the list of feature types, including rule specific parameters (e.g., a parameter to specify an explicit list of feature types)
    - Options for CRS:
        - A parameter with a (whitespace-separated) list of CRS URIs supported by the API
- A list of additional query parameters to be added (requires a link to a JSON overlay document). Note that in the UGAS-2020 pilot, a single JSON overlay document was used per query parameter. In a more structured approach, the overlay document could just include the parameter definition and specify the applicable combinations of the resource path, the HTTP method and any additional exceptions separately.

A sample ShapeChange configuration:

Listing 86. Sample ShapeChange configuration for the OpenAPI target

```
<Target class="de.interactive_instruments.ShapeChange.Target.OpenApi.OpenApiDefinition" mode="enabled" inputs="model">
 <advancedProcessConfigurations>
  <OpenApiConfigItems>
   <conformanceClasses>
    <ConformanceClass uri="http://www.opengis.net/spec/ogcapi-features-1/1.0/conf/core" overlay="https://shapechange.net/resources/openapi/overlays/features-1-10-core.json"/>
    <ConformanceClass uri="http://www.opengis.net/spec/ogcapi-features-1/1.0/conf/geojson" overlay="https://shapechange.net/resources/openapi/overlays/features-1-10-geojson.json"/>
    <ConformanceClass uri="http://www.opengis.net/spec/ogcapi-features-1/1.0/conf/html" overlay="https://shapechange.net/resources/openapi/overlays/features-1-10-html.json"/>
    <ConformanceClass uri="http://www.opengis.net/spec/ogcapi-features-2/1.0/conf/crs" overlay="https://shapechange.net/resources/openapi/overlays/features-2-10-crs.json" param="http://www.opengis.net/def/crs/OGC/1.3/CRS84 http://www.opengis.net/def/crs/EPSG/0/4326 http://www.opengis.net/def/crs/EPSG/0/3395"/>
   </conformanceClasses>
   <queryParameters>
    <QueryParameter name="f" overlay="config/f.json" appliesToPhase="pre-feature-identification"/>
    <QueryParameter name="string" overlay="config/string.json" appliesToPhase="finalization"/>
   </queryParameters>
  </OpenApiConfigItems>
 </advancedProcessConfigurations>
 <targetParameter name="outputDirectory" value="results/openapi"/>
 <targetParameter name="outputFilename" value="openapi.json"/>
 <targetParameter name="baseTemplate" value="https://shapechange.net/resources/openapi/overlays/default-template.json"/>
 <targetParameter name="jsonSchemasBaseLocation" value="https://example.org/schemas/json"/>
 <targetParameter name="jsonSchemaVersion" value="openapi30"/>
 <targetParameter name="collections" value="FeatureType1, FeatureType2"/>
 <targetParameter name="defaultEncodingRule" value="openapiEncodingRule"/>
 <rules>
  <EncodingRule name="openapiEncodingRule">
   <rule name="rule-openapi-all-explicit-collections"/>
  </EncodingRule>
 </rules>
</Target>
```

Customisation options beyond this minimal, extendable OpenAPI target were out-of-scope for this task. Examples include: Additional resource types, additional HTTP methods, more complex parameter (path or query), API metadata, deployment options, etc.

9.3.3. JSON Schema variants

9.3.3.1. General remarks

The OpenAPI 3.0 Schema object uses "an extended subset of JSON Schema Specification Wright Draft 00" [https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.3.md#data-types]. JSON Schema Specification Wright Draft 00 is also known as JSON Schema draft 05 [https://json-schema.org/specification-links.html#draft-5] and differs from draft 04 only in the textual descriptions.

It is important to note that differences between schemas in OpenAPI and JSON Schema should no longer be an issue with OpenAPI 3.1, which is expected to support the current JSON Schema Draft 2019-09 (current at the time of writing). A pull request to support JSON Schema Draft 2019-09 has already been merged (link [https://github.com/OAI/OpenAPI-Specification/pull/1977]). Any additional ShapeChange code to support the OpenAPI 3.0 variant are, therefore, fixes that are only relevant for a few months. Any code change that makes the code harder to maintain was thus considered out-of-scope.

To identify incompatibilities with JSON schemas created using the new ShapeChange JSON Schema target the JSON schemas created by the ShapeChange JSON Schema target unit tests [https://github.com/ShapeChange/ShapeChange/tree/json/src/test/resources/json/basic/reference/json_schemas/constraintconverter] were analyzed by using them in Swagger Editor [https://editor.swagger.io/], an online editor supporting OpenAPI 3.0.

A number of issues were identified and support for these issues were implemented in the JSON Schema target.

9.3.3.2. Issue: anchors

The JSON Schema target supports anchors to implement Location Independent Schema Identifiers. This is not supported in OpenAPI and the conversion rule *rule-json-cls-name-as-anchor* must not be used.

This has two effects:

- JSON members `$anchor` (draft 2019-09) or `$id` (draft 07) are suppressed;
- References in `$ref` do not use anchor identifiers for the fragment, but JSON pointers. Example: `https://example.org/baseuri/ts3/ts3/test3.json#/definitions/TS3_FT2` instead of `https://example.org/baseuri/ts3/ts3/test3.json#TS3_FT2`.

9.3.3.3. Issue: no type arrays

The value of `type` in OpenAPI 3.0 schemas is restricted to a single type, arrays like `"type": [ "string", "integer" ]` are not supported. However, `oneOf` is supported, so this can be written as `"oneOf": [ {"type": "string"}, {"type": "integer"} ]`.

9.3.3.4. Issue: `nullable` instead of a type `null`

OpenAPI 3.0 does not support a type `null`. Instead an extension element `nullable` was added. The semantics of `nullable` were not well-specified, which has caused issues. As part of the migration to using JSON Schema Draft 2019-09 in OpenAPI 3.1, the semantics of nullable have been clarified [https://github.com/OAI/OpenAPI-Specification/blob/master/proposals/003_Clarify-Nullable.md].

A consequence is that only simple cases can be fully converted to an OpenAPI schema. In the unit tests there were two patterns that can be converted.

1. A combination of a basic type and `null`, e.g., `{ "type": ["string","null"] }`. This can be mapped to `nullable`, the result is `{ "type": "string", "nullable": true }`.
2. A combination of an array type and `null`, i.e., `{ "oneOf": [ {"type": "null"}, {"type": "array", ⋯} ] }`. Again, this can be mapped to `nullable`, the result is `{ "type": "array", "nullable": true, ⋯ }`.

All other cases cannot be supported due to the semantic issues with `nullable` vs. an explicit type `null`. In those cases, the JSON Schema target reports an error and ignores the `null` option.

An example: `{ "oneOf": [{"type": "null"}, {"$ref": "X"}, {"$ref": "Y"}] }`. The main problem is that it is not possible to make the referenced types `X` or `Y` nullable.

9.3.3.5. Selecting the OpenAPI 3.0 JSON Schema variant

The OpenAPI 3.0 variant is selected by setting "OpenApi30" as the value of the ShapeChange JSON Schema target configuration parameter *jsonSchemaVersion*. See JSON Schema Version.

9.4. Results

Based upon the results of the analysis, and following the design developed in UGAS-2020, a new ShapeChange OpenAPI target has been implemented, which supports the creation of an OpenAPI definition with explicit feature collection paths.

Using ShapeChange, an OpenAPI definition as well as a JSON Schema has been derived from the application schema used in the Scenario. The results are contained in Annex A.

Annex A: OpenAPI and JSON Schema Files

A.1. Results from the OpenAPI Scenario

The OpenAPI definition in Listing 87 as well as the JSON Schema in Listing 88 have been produced using the new ShapeChange targets developed in UGAS-2020, for the application schema used in the OpenAPI scenario.

Listing 87. OpenAPI definition produced for the OpenAPI scenario

```
{
 "openapi": "3.0.2",
 "info": {
  "title": "a title of the API",
  "version": "a version identifier, e.g. 1.0.0",
  "description": "a longer description what the API does, supports Markdown markup",
  "contact": {
   "name": "name of the organisation",
   "email": "info@example.org",
   "url": "https://example.org/"
  },
  "license": {
   "name": "name of the license of the API, e.g. CC-BY 4.0 license",
   "url": "https://creativecommons.org/licenses/by/4.0/"
  }
 },
 "servers": [
  {"url": "https://data.example.org/"}
 ],
 "tags": [
  {
   "name": "Capabilities",
   "description": "essential characteristics of this API"
  },
  {
   "name": "Data",
   "description": "access to data (features)"
  }
 ],
 "paths": {
  "/": {
   "get": {
    "tags": ["Capabilities"],
    "summary": "landing page",
    "description": "The landing page provides links to the API definition, the conformance declaration and to the feature collections of this dataset.",
    "operationId": "getLandingPage",
    "responses": {
```

```
      "200": {"$ref": "#/components/responses/LandingPage"},
      "500": {"$ref": "#/components/responses/ServerError"},
      "400": {"$ref": "#/components/responses/InvalidParameter"}
     },
     "parameters": [
      {"$ref": "#/components/parameters/f"}
     ]
    }
   },
   "/conformance": {
    "get": {
     "tags": ["Capabilities"],
     "summary": "information about specifications that this API conforms to",
     "description": "A list of all conformance classes specified in a standard that the API conforms to.",
     "operationId": "getConformanceDeclaration",
     "responses": {
      "200": {"$ref": "#/components/responses/ConformanceDeclaration"},
      "500": {"$ref": "#/components/responses/ServerError"},
      "400": {"$ref": "#/components/responses/InvalidParameter"}
     },
     "parameters": [
      {"$ref": "#/components/parameters/f"}
     ]
    }
   },
   "/collections": {
    "get": {
     "tags": ["Capabilities"],
     "summary": "the feature collections",
     "description": "Fetch the feature collections in the dataset.",
     "operationId": "getCollections",
     "responses": {
      "200": {"$ref": "#/components/responses/Collections"},
      "500": {"$ref": "#/components/responses/ServerError"},
      "400": {"$ref": "#/components/responses/InvalidParameter"}
     },
     "parameters": [
      {"$ref": "#/components/parameters/f"}
     ]
    }
   },
   "/collections/FeatureType1": {
    "get": {
     "tags": ["Capabilities"],
     "summary": "the feature collection 'FeatureType1'",
     "description": "Fetch the feature collection 'FeatureType1'.",
     "operationId": "getCollection_FeatureType1",
     "parameters": [
      {"$ref": "#/components/parameters/f"}
     ],
```

```
    "responses": {
     "200": {"$ref": "#/components/responses/Collection"},
     "404": {"$ref": "#/components/responses/NotFound"},
     "500": {"$ref": "#/components/responses/ServerError"},
     "400": {"$ref": "#/components/responses/InvalidParameter"}
    }
   }
  },
  "/collections/FeatureType1/items": {
   "get": {
    "tags": ["Data"],
    "summary": "fetch features in the feature collection 'FeatureType1'",
    "description": "Fetch features in the feature collection 'FeatureType1'. The features included in the response are determined by the server based on the query parameters of the request. To support access to larger collections without overloading the client, the API supports paged access with links to the next page, if more features are selected that the page size. The `bbox` and `datetime` parameter can be used to select only a subset of the features in the collection (the features that are in the bounding box or date-time interval). The `bbox` parameter matches all features in the collection that are not associated with a location, too. The `datetime` parameter matches all features in the collection that are not associated with a time stamp or interval, too. The `limit` parameter may be used to control the maximum number of the selected features that should be returned in the response, the page size. Each page may include information about the number of selected and returned features (`numberMatched` and `numberReturned`) as well as a link to support paging (link relation type `next`).",
    "operationId": "getFeatures_FeatureType1",
    "parameters": [
     {"$ref": "#/components/parameters/limit"},
     {"$ref": "#/components/parameters/bbox"},
     {"$ref": "#/components/parameters/datetime"},
     {"$ref": "#/components/parameters/crs"},
     {"$ref": "#/components/parameters/bbox-crs"},
     {"$ref": "#/components/parameters/f"}
    ],
    "responses": {
     "200": {"$ref": "#/components/responses/Features_FeatureType1"},
     "400": {"$ref": "#/components/responses/InvalidParameter"},
     "404": {"$ref": "#/components/responses/NotFound"},
     "500": {"$ref": "#/components/responses/ServerError"}
    }
   }
  },
  "/collections/FeatureType1/items/{featureId}": {
   "get": {
    "tags": ["Data"],
    "summary": "fetch a single feature in the feature collection 'FeatureType1'",
    "description": "Fetch the feature with id `featureId`.",
    "operationId": "getFeature_FeatureType1",
    "parameters": [
```

```
      {"$ref": "#/components/parameters/featureId"},
      {"$ref": "#/components/parameters/crs"},
      {"$ref": "#/components/parameters/f"}
     ],
     "responses": {
      "200": {"$ref": "#/components/responses/Feature_FeatureType1"},
      "404": {"$ref": "#/components/responses/NotFound"},
      "500": {"$ref": "#/components/responses/ServerError"},
      "400": {"$ref": "#/components/responses/InvalidParameter"}
     }
    }
   },
   "/collections/FeatureType2": {
    "get": {
     "tags": ["Capabilities"],
     "summary": "the feature collection 'FeatureType2'",
     "description": "Fetch the feature collection 'FeatureType2'.",
     "operationId": "getCollection_FeatureType2",
     "parameters": [
      {"$ref": "#/components/parameters/f"}
     ],
     "responses": {
      "200": {"$ref": "#/components/responses/Collection"},
      "404": {"$ref": "#/components/responses/NotFound"},
      "500": {"$ref": "#/components/responses/ServerError"},
      "400": {"$ref": "#/components/responses/InvalidParameter"}
     }
    }
   },
   "/collections/FeatureType2/items": {
    "get": {
     "tags": ["Data"],
     "summary": "fetch features in the feature collection 'FeatureType2'",
     "description": "Fetch features in the feature collection 'FeatureType2'. The features included in the response are determined by the server based on the query parameters of the request. To support access to larger collections without overloading the client, the API supports paged access with links to the next page, if more features are selected that the page size. The `bbox` and `datetime` parameter can be used to select only a subset of the features in the collection (the features that are in the bounding box or date-time interval). The `bbox` parameter matches all features in the collection that are not associated with a location, too. The `datetime` parameter matches all features in the collection that are not associated with a time stamp or interval, too. The `limit` parameter may be used to control the maximum number of the selected features that should be returned in the response, the page size. Each page may include information about the number of selected and returned features (`numberMatched` and `numberReturned`) as well as a link to support paging (link relation type `next`).",
     "operationId": "getFeatures_FeatureType2",
     "parameters": [
      {"$ref": "#/components/parameters/limit"},
```

```
      {"$ref": "#/components/parameters/bbox"},
      {"$ref": "#/components/parameters/datetime"},
      {"$ref": "#/components/parameters/crs"},
      {"$ref": "#/components/parameters/bbox-crs"},
      {"$ref": "#/components/parameters/f"},
      {"$ref": "#/components/parameters/string"}
     ],
     "responses": {
      "200": {"$ref": "#/components/responses/Features_FeatureType2"},
      "400": {"$ref": "#/components/responses/InvalidParameter"},
      "404": {"$ref": "#/components/responses/NotFound"},
      "500": {"$ref": "#/components/responses/ServerError"}
     }
    }
   },
   "/collections/FeatureType2/items/{featureId}": {
    "get": {
     "tags": ["Data"],
     "summary": "fetch a single feature in the feature collection 'FeatureType2'",
     "description": "Fetch the feature with id `featureId`.",
     "operationId": "getFeature_FeatureType2",
     "parameters": [
      {"$ref": "#/components/parameters/featureId"},
      {"$ref": "#/components/parameters/crs"},
      {"$ref": "#/components/parameters/f"}
     ],
     "responses": {
      "200": {"$ref": "#/components/responses/Feature_FeatureType2"},
      "404": {"$ref": "#/components/responses/NotFound"},
      "500": {"$ref": "#/components/responses/ServerError"},
      "400": {"$ref": "#/components/responses/InvalidParameter"}
     }
    }
   }
  },
  "components": {
   "parameters": {
    "featureId": {
     "name": "featureId",
     "in": "path",
     "description": "local identifier of a feature",
     "required": true,
     "schema": {"type": "string"}
    },
    "bbox": {
     "name": "bbox",
     "in": "query",
     "description": "Only features that have a geometry that intersects the
bounding box are selected.\nThe bounding box is provided as four or six numbers,
depending on whether the coordinate reference system includes a vertical axis
(height or depth):\n\n* Lower left corner, coordinate axis 1\n* Lower left corner,
```

```
    coordinate axis 2\n* Minimum value, coordinate axis 3 (optional)\n* Upper right
corner, coordinate axis 1\n* Upper right corner, coordinate axis 2\n* Maximum value,
coordinate axis 3 (optional)\n\nThe coordinate reference system of the values is WGS
84 longitude/latitude\n(http://www.opengis.net/def/crs/OGC/1.3/CRS84) unless a
different coordinate reference system is specified in the parameter `bbox-crs`.\n\nFor
WGS 84 longitude/latitude the values are in most cases the sequence of\nminimum
longitude, minimum latitude, maximum longitude and maximum latitude. However, in cases
where the box spans the antimeridian the first value (west-most box edge) is larger
than the third value (east-most box edge).\n\nIf the vertical axis is included, the
third and the sixth number are the bottom and the top of the 3-dimensional bounding
box.",
210     "required": false,
211     "style": "form",
212     "explode": false,
213     "schema": {
214      "minItems": 4,
215      "maxItems": 6,
216      "type": "array",
217      "items": {"type": "number"}
218     }
219    },
220    "datetime": {
221     "name": "datetime",
222     "in": "query",
223     "description": "Either a date-time or an interval, open or closed. Date and
time expressions adhere to RFC 3339. Open intervals are expressed using double-dots.
Examples:\n\n* A date-time: \"2018-02-12T23:20:50Z\"\n* A closed interval: \"2018-02-
12T00:00:00Z/2018-03-18T12:31:12Z\"\n* Open intervals: \"2018-02-12T00:00:00Z/..\" or
\"../2018-03-18T12:31:12Z\"\n\nOnly features that have a temporal property that
intersects the value of `datetime` are selected.",
224     "required": false,
225     "style": "form",
226     "explode": false,
227     "schema": {"type": "string"}
228    },
229    "limit": {
230     "name": "limit",
231     "in": "query",
232     "description": "The optional limit parameter limits the number of items that
are presented in the response document. Only items are counted that are on the first
level of the collection in the response document. Nested objects contained within the
explicitly requested items are not be counted.",
233     "required": false,
234     "style": "form",
235     "explode": false,
236     "schema": {
237      "minimum": 1,
238      "maximum": 10000,
239      "type": "integer",
240      "default": 10
241     }
```

```
    },
    "crs": {
      "name": "crs",
      "in": "query",
      "description": "The coordinate reference system of the response geometries. Default is WGS84 longitude/latitude (http://www.opengis.net/def/crs/OGC/1.3/CRS84).",
      "required": false,
      "style": "form",
      "explode": false,
      "schema": {
        "type": "string",
        "default": "http://www.opengis.net/def/crs/OGC/1.3/CRS84",
        "enum": [
          "http://www.opengis.net/def/crs/OGC/1.3/CRS84",
          "http://www.opengis.net/def/crs/EPSG/0/4326",
          "http://www.opengis.net/def/crs/EPSG/0/3395"
        ]
      }
    },
    "bbox-crs": {
      "name": "bbox-crs",
      "in": "query",
      "description": "The coordinate reference system of the bbox parameter. Default is WGS84 longitude/latitude (http://www.opengis.net/def/crs/OGC/1.3/CRS84).",
      "required": false,
      "style": "form",
      "explode": false,
      "schema": {
        "type": "string",
        "default": "http://www.opengis.net/def/crs/OGC/1.3/CRS84",
        "enum": [
          "http://www.opengis.net/def/crs/OGC/1.3/CRS84",
          "http://www.opengis.net/def/crs/EPSG/0/4326",
          "http://www.opengis.net/def/crs/EPSG/0/3395"
        ]
      }
    },
    "f": {
      "name": "f",
      "in": "query",
      "description": "The format of the response. If no value is provided, the standard http rules apply, i.e., the accept header will be used to determine the format. Allowed values are 'json' and 'html'.",
      "required": false,
      "style": "form",
      "explode": false,
      "schema": {
        "type": "string",
        "enum": [
          "json",
```

```
        "html"
       ]
      }
     },
     "string": {
      "name": "string",
      "in": "query",
      "description": "Only return features where the property of the same name has the provided value. Default = return all features.",
      "required": false,
      "schema": {"type": "string"},
      "style": "form",
      "explode": false
     }
    },
    "responses": {
     "LandingPage": {
      "description": "The landing page provides links to the API definition (link relation types `service-desc` and `service-doc`), the Conformance declaration (path `/conformance`, link relation type `conformance`), and to other resources.",
      "content": {
       "application/json": {
        "schema": {"$ref": "#/components/schemas/LandingPage"}
       },
       "text/html": {
        "schema": {"$ref": "#/components/schemas/htmlPage"}
       }
      }
     },
     "ConformanceDeclaration": {
      "description": "The URIs of all conformance classes supported by the API.",
      "content": {
       "application/json": {
        "schema": {"$ref": "#/components/schemas/ConformanceDeclaration"}
       },
       "text/html": {
        "schema": {"$ref": "#/components/schemas/htmlPage"}
       }
      }
     },
     "Collections": {
      "description": "The feature collections shared by this API.",
      "content": {
       "application/json": {
        "schema": {"$ref": "#/components/schemas/Collections"}
       },
       "text/html": {
        "schema": {"$ref": "#/components/schemas/htmlPage"}
       }
      }
     },
```

```
    "Collection": {
     "description": "A feature collection.",
     "content": {
      "application/json": {
       "schema": {"$ref": "#/components/schemas/Collection"}
      },
      "text/html": {
       "schema": {"$ref": "#/components/schemas/htmlPage"}
      }
     }
    },
    "InvalidParameter": {
     "description": "A query parameter has an invalid value.",
     "content": {
      "application/json": {
       "schema": {"$ref": "#/components/schemas/Exception"}
      },
      "text/html": {
       "schema": {"$ref": "#/components/schemas/htmlPage"}
      }
     }
    },
    "NotFound": {
     "description": "The requested URI was not found.",
     "content": {
      "application/json": {
       "schema": {"$ref": "#/components/schemas/Exception"}
      },
      "text/html": {
       "schema": {"$ref": "#/components/schemas/htmlPage"}
      }
     }
    },
    "ServerError": {
     "description": "A server error occurred.",
     "content": {
      "application/json": {
       "schema": {"$ref": "#/components/schemas/Exception"}
      },
      "text/html": {
       "schema": {"$ref": "#/components/schemas/htmlPage"}
      }
     }
    },
    "Features_FeatureType1": {
     "description": "The response is a document consisting of features in the collection. The features included in the response are determined by the server based on the query parameters of the request.",
     "content": {
      "application/geo+json": {
       "schema": {"$ref": "#/components/schemas/Features_FeatureType1"}
```

```
      },
      "text/html": {
       "schema": {"$ref": "#/components/schemas/htmlPage"}
      }
     }
    },
    "Feature_FeatureType1": {
     "description": "The feature with id `{featureId}` in the feature collection with id `FeatureType1`",
     "content": {
      "application/geo+json": {
       "schema": {"$ref": "#/components/schemas/Feature_FeatureType1"}
      },
      "text/html": {
       "schema": {"$ref": "#/components/schemas/htmlPage"}
      }
     }
    },
    "Features_FeatureType2": {
     "description": "The response is a document consisting of features in the collection. The features included in the response are determined by the server based on the query parameters of the request.",
     "content": {
      "application/geo+json": {
       "schema": {"$ref": "#/components/schemas/Features_FeatureType2"}
      },
      "text/html": {
       "schema": {"$ref": "#/components/schemas/htmlPage"}
      }
     }
    },
    "Feature_FeatureType2": {
     "description": "The feature with id `{featureId}` in the feature collection with id `FeatureType2`",
     "content": {
      "application/geo+json": {
       "schema": {"$ref": "#/components/schemas/Feature_FeatureType2"}
      },
      "text/html": {
       "schema": {"$ref": "#/components/schemas/htmlPage"}
      }
     }
    }
   },
   "schemas": {
    "Collection": {
     "required": [
      "id",
      "links"
     ],
     "type": "object",
```

```
    "properties": {
     "id": {
      "type": "string",
      "description": "identifier of the collection used, for example, in URIs"
     },
     "title": {
      "type": "string",
      "description": "human readable title of the collection"
     },
     "description": {
      "type": "string",
      "description": "a description of the features in the collection"
     },
     "links": {
      "type": "array",
      "items": {"$ref": "#/components/schemas/Link"}
     },
     "extent": {"$ref": "#/components/schemas/Extent"},
     "itemType": {
      "type": "string",
      "description": "indicator about the type of the items in the collection (the default value is 'feature')."
      "default": "feature"
     },
     "crs": {
      "type": "array",
      "description": "the list of coordinate reference systems supported by the service",
      "items": {"type": "string"},
      "default": ["http://www.opengis.net/def/crs/OGC/1.3/CRS84"]
     },
     "storageCrs": {
      "type": "string",
      "description": "the CRS identifier, from the list of supported CRS identifiers, that may be used to retrieve features from a collection without the need to apply a CRS transformation"
     }
    }
   },
   "Collections": {
    "required": [
     "collections",
     "links"
    ],
    "type": "object",
    "properties": {
     "links": {
      "type": "array",
      "items": {"$ref": "#/components/schemas/Link"}
     },
     "collections": {
```

```
      "type": "array",
      "items": {"$ref": "#/components/schemas/Collection"}
     },
     "crs": {
      "type": "array",
      "items": {"type": "string"},
      "description": "a list of CRS identifiers that are supported for more that one feature collection offered by the service"
     }
    }
   },
   "ConformanceDeclaration": {
    "required": ["conformsTo"],
    "type": "object",
    "properties": {
     "conformsTo": {
      "type": "array",
      "items": {"type": "string"}
     }
    }
   },
   "Exception": {
    "required": ["code"],
    "type": "object",
    "properties": {
     "code": {
      "type": "string",
      "description": "HTTP status code of the error (4xx or 5xx)"
     },
     "description": {
      "type": "string",
      "description": "description of the error"
     }
    }
   },
   "Extent": {
    "type": "object",
    "properties": {
     "spatial": {
      "type": "object",
      "properties": {
       "bbox": {
        "minItems": 1,
        "type": "array",
        "description": "One or more bounding boxes that describe the spatial extent of the dataset.\nIn the Core only a single bounding box is supported. Extensions may support\nadditional areas. If multiple areas are provided, the union of the bounding\nboxes describes the spatial extent.",
        "items": {
         "maxItems": 6,
         "minItems": 4,
```

```
526           "type": "array",
527           "description": "Each bounding box is provided as four or six numbers,
    depending on\nwhether the coordinate reference system includes a vertical
    axis\n(height or depth):\n\n* Lower left corner, coordinate axis 1\n* Lower left
    corner, coordinate axis 2\n* Minimum value, coordinate axis 3 (optional)\n* Upper
    right corner, coordinate axis 1\n* Upper right corner, coordinate axis 2\n*
    Maximum value, coordinate axis 3 (optional)\n\nThe coordinate reference system of
    the values is WGS 84
    longitude/latitude\n(http://www.opengis.net/def/crs/OGC/1.3/CRS84) unless a
    different coordinate\nreference system is specified in `crs`.\n\nFor WGS 84
    longitude/latitude the values are in most cases the sequence of\nminimum
    longitude, minimum latitude, maximum longitude and maximum latitude.\nHowever, in
    cases where the box spans the antimeridian the first value\n(west-most box edge)
    is larger than the third value (east-most box edge).\n\nIf the vertical axis is
    included, the third and the sixth number are\nthe bottom and the top of the 3-
    dimensional bounding box.\n\nIf a feature has multiple spatial geometry
    properties, it is the decision of the\nserver whether only a single spatial
    geometry property is used to determine\nthe extent or all relevant geometries.",
528           "example": [
529            -180,
530            -90,
531            180,
532            90
533           ],
534           "items": {"type": "number"}
535          }
536         },
537         "crs": {
538          "type": "string",
539          "description": "Coordinate reference system of the coordinates in the
    spatial extent\n(property `bbox`). The default reference system is WGS 84
    longitude/latitude.",
540          "default": "http://www.opengis.net/def/crs/OGC/1.3/CRS84"
541         }
542        },
543        "description": "The spatial extent of the features in the collection."
544       },
545       "temporal": {
546        "type": "object",
547        "properties": {
548         "interval": {
549          "minItems": 1,
550          "type": "array",
551          "description": "One or more time intervals that describe the temporal
    extent of the dataset. The value `null` is supported and indicates an open time
    intervall.\nIn the Core only a single time interval is supported. Extensions may
    support multiple intervals. If multiple intervals are provided, the union of the
    intervals describes the temporal extent.",
552          "items": {
553           "maxItems": 2,
554           "minItems": 2,
```

```
555          "type": "array",
556          "description": "Begin and end times of the time interval. The timestamps
     are in the coordinate reference system specified in `trs`. By default this is the
     Gregorian calendar.",
557          "items": {
558           "type": "string",
559           "format": "date-time",
560           "nullable": true
561          }
562         }
563        },
564        "trs": {
565         "type": "string",
566         "description": "Coordinate reference system of the coordinates in the
     temporal extent\n(property `interval`). The default reference system is the
     Gregorian calendar.",
567          "default": "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian"
568        }
569       },
570       "description": "The temporal extent of the features in the collection."
571      }
572     },
573     "description": "The extent of the features in the collection. In the Core only
     spatial and temporal\nextents are specified. Extensions may add additional members
     to represent other\nextents, for example, thermal or pressure ranges."
574    },
575    "Geometry": {
576     "oneOf": [
577      {"$ref": "#/components/schemas/Point"},
578      {"$ref": "#/components/schemas/MultiPoint"},
579      {"$ref": "#/components/schemas/LineString"},
580      {"$ref": "#/components/schemas/MultiLineString"},
581      {"$ref": "#/components/schemas/Polygon"},
582      {"$ref": "#/components/schemas/MultiPolygon"},
583      {"$ref": "#/components/schemas/GeometryCollection"}
584     ],
585     "nullable": true
586    },
587    "GeometryCollection": {
588     "required": [
589      "geometries",
590      "type"
591     ],
592     "type": "object",
593     "properties": {
594      "type": {
595       "type": "string",
596       "enum": ["GeometryCollection"]
597      },
598      "geometries": {
599       "type": "array",
```

```
      "items": {"$ref": "#/components/schemas/Geometry"}
     }
    }
   },
   "LandingPage": {
    "required": ["links"],
    "type": "object",
    "properties": {
     "title": {"type": "string"},
     "description": {"type": "string"},
     "links": {
      "type": "array",
      "items": {"$ref": "#/components/schemas/Link"}
     }
    }
   },
   "LineString": {
    "required": [
     "coordinates",
     "type"
    ],
    "type": "object",
    "properties": {
     "type": {
      "type": "string",
      "enum": ["LineString"]
     },
     "coordinates": {
      "minItems": 2,
      "type": "array",
      "items": {
       "minItems": 2,
       "type": "array",
       "items": {"type": "number"}
      }
     }
    }
   },
   "Link": {
    "required": ["href"],
    "type": "object",
    "properties": {
     "href": {"type": "string"},
     "rel": {"type": "string"},
     "type": {"type": "string"},
     "hreflang": {"type": "string"},
     "title": {"type": "string"},
     "length": {"type": "integer"}
    }
   },
   "MultiLineString": {
```

```
    "required": [
     "coordinates",
     "type"
    ],
    "type": "object",
    "properties": {
     "type": {
      "type": "string",
      "enum": ["MultiLineString"]
     },
     "coordinates": {
      "type": "array",
      "items": {
       "minItems": 2,
       "type": "array",
       "items": {
        "minItems": 2,
        "type": "array",
        "items": {"type": "number"}
       }
      }
     }
    }
   },
   "MultiPoint": {
    "required": [
     "coordinates",
     "type"
    ],
    "type": "object",
    "properties": {
     "type": {
      "type": "string",
      "enum": ["MultiPoint"]
     },
     "coordinates": {
      "type": "array",
      "items": {
       "minItems": 2,
       "type": "array",
       "items": {"type": "number"}
      }
     }
    }
   },
   "MultiPolygon": {
    "required": [
     "coordinates",
     "type"
    ],
    "type": "object",
```

```
    "properties": {
     "type": {
      "type": "string",
      "enum": ["MultiPolygon"]
     },
     "coordinates": {
      "type": "array",
      "items": {
       "type": "array",
       "items": {
        "minItems": 4,
        "type": "array",
        "items": {
         "minItems": 2,
         "type": "array",
         "items": {"type": "number"}
        }
       }
      }
     }
    },
    "NumberMatched": {
     "minimum": 0,
     "type": "integer",
     "description": "The number of features of the feature type that match the selection\nparameters like `bbox`."
    },
    "NumberReturned": {
     "minimum": 0,
     "type": "integer",
     "description": "The number of features in the feature collection.\n\nA server may omit this information in a response, if the information about the number of features is not known or difficult to compute.\n\nIf the value is provided, the value is identical to the number of items in the 'features' array."
    },
    "Point": {
     "required": [
      "coordinates",
      "type"
     ],
     "type": "object",
     "properties": {
      "type": {
       "type": "string",
       "enum": ["Point"]
      },
      "coordinates": {
       "minItems": 2,
       "type": "array",
       "items": {"type": "number"}
```

```
      }
     }
    },
    "Polygon": {
     "required": [
      "coordinates",
      "type"
     ],
     "type": "object",
     "properties": {
      "type": {
       "type": "string",
       "enum": ["Polygon"]
      },
      "coordinates": {
       "type": "array",
       "items": {
        "minItems": 4,
        "type": "array",
        "items": {
         "minItems": 2,
         "type": "array",
         "items": {"type": "number"}
        }
       }
      }
     }
    },
    "TimeStamp": {
     "type": "string",
     "description": "This property indicates the time and date when the response was generated.",
     "format": "date-time"
    },
    "htmlPage": {"type": "string"},
    "Features_FeatureType1": {
     "required": [
      "features",
      "type"
     ],
     "type": "object",
     "properties": {
      "type": {
       "type": "string",
       "enum": ["FeatureCollection"]
      },
      "features": {
       "type": "array",
       "items": {"$ref": "#/components/schemas/Feature_FeatureType1"}
      },
      "links": {
```

```
      "type": "array",
      "items": {"$ref": "#/components/schemas/Link"}
     },
     "timeStamp": {"$ref": "#/components/schemas/TimeStamp"},
     "numberMatched": {"$ref": "#/components/schemas/NumberMatched"},
     "numberReturned": {"$ref": "#/components/schemas/NumberReturned"}
    }
   },
   "Feature_FeatureType1": {"$ref":
```

```
"https://example.org/schemas/json/applications/app_schema.json#/definitions/FeatureTyp
e1"},
808    "Features_FeatureType2": {
809     "required": [
810      "features",
811      "type"
812     ],
813     "type": "object",
814     "properties": {
815      "type": {
816       "type": "string",
817       "enum": ["FeatureCollection"]
818      },
819      "features": {
820       "type": "array",
821       "items": {"$ref": "#/components/schemas/Feature_FeatureType2"}
822      },
823      "links": {
824       "type": "array",
825       "items": {"$ref": "#/components/schemas/Link"}
826      },
827      "timeStamp": {"$ref": "#/components/schemas/TimeStamp"},
828      "numberMatched": {"$ref": "#/components/schemas/NumberMatched"},
829      "numberReturned": {"$ref": "#/components/schemas/NumberReturned"}
830     }
831    },
832    "Feature_FeatureType2": {"$ref":
"https://example.org/schemas/json/applications/app_schema.json#/definitions/FeatureTyp
e2"}
833   }
834  }
835 }
```

Listing 88. JSON Schema produced for the OpenAPI scenario

```
1 {
2  "$id": "https://example.org/schemas/json/applications/app_schema.json",
3  "definitions": {
4   "CodeList": {"type": "string"},
5   "DataType": {
6    "type": "object",
7    "properties": {
8     "datatype": {
9      "type": "array",
10     "items": {"$ref":
```

```
"https://example.org/schemas/json/applications/app_schema.json#/definitions/DataType2"
},
 11       "uniqueItems": true
 12      },
 13      "string": {
 14       "type": "array",
 15       "minItems": 1,
 16       "items": {"type": "string"},
 17       "uniqueItems": true
 18      },
 19      "boolean": {"type": "boolean"}
 20     },
 21     "required": ["string"]
 22    },
 23    "DataType2": {
 24     "type": "object",
 25     "properties": {
 26      "string": {
 27       "type": "array",
 28       "minItems": 1,
 29       "items": {"type": "string"},
 30       "uniqueItems": true
 31      },
 32      "integer": {"type": "integer"}
 33     },
 34     "required": ["string"]
 35    },
 36    "Enumeration": {
 37     "type": "string",
 38     "enum": [
 39      "val1",
 40      "val2"
 41     ]
 42    },
 43    "FeatureType1": {
 44     "allOf": [
 45      {"$ref": "https://geojson.org/schema/Feature.json"},
 46      {"$ref":
"https://example.org/schemas/json/applications/app_schema.json#/definitions/Root"},
 47      {
 48       "type": "object",
 49       "properties": {
 50        "geometry": {"$ref": "https://geojson.org/schema/Point.json"},
 51        "properties": {
 52         "type": "object",
 53         "properties": {
 54          "integer": {"type": "integer"},
 55          "character": {"type": "string"},
 56          "string": {
 57           "type": "array",
```

```
          "minItems": 1,
          "items": {"type": "string"},
          "uniqueItems": true
        },
        "real": {
          "type": "array",
          "items": {"type": "number"},
          "uniqueItems": true
        },
        "decimal": {"type": "number"},
        "number": {"type": "number"},
        "boolean": {"type": "boolean"},
        "uri": {
          "type": "string",
          "format": "uri"
        },
        "datetime": {
          "type": "string",
          "format": "date-time"
        },
        "date": {
          "type": "string",
          "format": "date"
        },
        "time": {
          "type": "string",
          "format": "time"
        },
        "measure": {"type": "number"},
        "length": {"type": "number"},
        "metadata": {"$ref":
"https://example.org/external/schema/definitions.json#MD_Metadata"},
        "datatype": {"$ref":
```

```
"https://example.org/schemas/json/applications/app_schema.json#/definitions/DataType"}
,
 90         "enum": {"$ref":
"https://example.org/schemas/json/applications/app_schema.json#/definitions/Enumeratio
n"},
 91         "codelist": {"$ref":
"https://example.org/schemas/json/applications/app_schema.json#/definitions/CodeList"}
,
 92         "role2": {"$ref": "https://example.org/jsonschema/byreference.json"}
 93        },
 94        "required": [
 95         "boolean",
 96         "character",
 97         "codelist",
 98         "datatype",
 99         "date",
100         "datetime",
101         "decimal",
102         "enum",
103         "measure",
104         "metadata",
105         "number",
106         "role2",
107         "string",
108         "time",
109         "uri"
110        ]
111       }
112      },
113      "required": ["properties"]
114     }
115    ]
116   },
117   "FeatureType2": {
118    "allOf": [
119     {"$ref": "https://geojson.org/schema/Feature.json"},
120     {"$ref":
"https://example.org/schemas/json/applications/app_schema.json#/definitions/Root"},
121     {
122      "type": "object",
123      "properties": {
124       "properties": {
125        "type": "object",
126        "properties": {
127         "codelist": {"$ref":
"https://example.org/schemas/json/applications/app_schema.json#/definitions/CodeList"}
,
128         "string": {"type": "string"},
129         "role1": {
130          "type": "array",
```

```
          "items": {"$ref": "https://example.org/jsonschema/byreference.json"},
          "uniqueItems": true
         }
        },
        "required": [
         "codelist",
         "string"
        ]
       }
      },
      "required": ["properties"]
     }
    ]
  },
  "NilUnion": {
   "type": "object",
   "properties": {
    "value": {"$ref":
```

```
"https://example.org/schemas/json/applications/app_schema.json#/definitions/DataType2"
},
149     "reason": {"type": "string"}
150    },
151    "additionalProperties": false,
152    "minProperties": 1,
153    "maxProperties": 1
154   },
155   "Root": {
156    "type": "object",
157    "properties": {
158     "collection": {}
159    },
160    "required": ["collection"]
161   },
162   "Union": {
163    "type": "object",
164    "properties": {
165     "option1": {"$ref":
"https://example.org/schemas/json/applications/app_schema.json#/definitions/Enumeratio
n"},
166     "option2": {"type": "integer"},
167     "option3": {
168      "type": "array",
169      "items": {"type": "string"},
170      "uniqueItems": true
171     }
172    },
173    "additionalProperties": false,
174    "minProperties": 1,
175    "maxProperties": 1
176   }
177  }
178 }
```

Annex B: ShapeChange Configurations for Derivation of JSON Schemas from the NAS Conceptual Model

This Annex contains ShapeChange configurations with which schemas defined in the NAS conceptual model can be encoded as JSON Schemas:

- the NSG Application Schema
- ISO 19100-series Schemas
- the U.S. Intelligence Community Metadata Schema
- the NAS SWE Common Implementation Schema

In addition, UGAS-2020 investigated the derivation of a JSON Schema for SWE Common 2.0. The results of that investigation are documented in section SWE Common 2.0 JSON Schema.

All of these JSON Schema encodings create an additional JSON member, with which the *type* of the JSON encoded object can be identified (for details on the according conversion rule, see Type Identification). Such a type member is useful for a conversion from JSON to RDF (using JSON-LD), and is also useful (actually, necessary) to support the conversion of restrictions of a property value type, defined using OCL constraint (for further details, see the Value Type section in the UML to JSON Schema Encoding Rule chapter).

Furthermore, the JSON Schema encodings support both inline and by reference encoding of property values. The main reason is that XML Schemas of ISO 19100-series standards typically support both inline and by reference value encoding.

The JSON Schema encodings use mappings for types from ISO 19103, ISO 19107, ISO 19108, and ISO 19109 as defined in Listing 89. The only exception is the SWE Common 2.0 JSON encoding, which requires specific mappings. Most of the mappings from Listing 89 are defined by the Features Core Profile of Key Community Conceptual Schemas.

Listing 89. Common map entries used by JSON Schema encodings

```
<?xml version="1.0" encoding="UTF-8"?>
<mapEntries xmlns="http://www.interactive-instruments.de/ShapeChange/Configuration/1.1" xmlns:xi="http://www.w3.org/2001/XInclude">

 <!-- ISO/TS 19103:2015 -->
 <MapEntry type="Boolean" rule="*" targetType="boolean" param=""/>
 <MapEntry type="Character" rule="*" targetType="string" param="keywords{minLength=1;maxLength=1}"/>
 <MapEntry type="CharacterString" rule="*" targetType="string" param=""/>
 <MapEntry type="Date" rule="*" targetType="string" param="keywords{format=date}"/>
 <MapEntry type="DateTime" rule="*" targetType="string" param="keywords{format=date-time}"/>
 <MapEntry type="Duration" rule="*" targetType="string"
```

```
   param="keywords{format=duration}"/>
  <MapEntry type="Time" rule="*" targetType="string" param="keywords{format=time}"/>
  <MapEntry type="Decimal" rule="*" targetType="number" param=""/>
  <MapEntry type="Integer" rule="*" targetType="integer" param=""/>
  <MapEntry type="Number" rule="*" targetType="number" param=""/>
  <MapEntry type="Real" rule="*" targetType="number" param=""/>
  <MapEntry type="URI" rule="*" targetType="string" param="keywords{format=uri}"/>
  <MapEntry type="URL" rule="*" targetType="string" param="keywords{format=uri}"/>
  <MapEntry type="URN" rule="*" targetType="string" param="keywords{format=uri}"/>
  <MapEntry type="Measure" rule="*"
targetType="http://www.opengis.net/to/be/determined/Measure.json" param=""/>
  <MapEntry type="Length" rule="*"
targetType="http://www.opengis.net/to/be/determined/Measure.json" param=""/>
  <MapEntry type="Any" rule="*"
targetType="http://www.opengis.net/to/be/determined/Any.json" param=""/>
  <MapEntry type="Record" rule="*"
targetType="http://www.opengis.net/to/be/determined/Record.json" param=""/>
  <MapEntry type="RecordType" rule="*"
targetType="http://www.opengis.net/to/be/determined/RecordType.json" param=""/>


  <MapEntry type="GenericName" rule="*" targetType="string" param=""/>
  <MapEntry type="LocalName" rule="*" targetType="string" param=""/>
  <MapEntry type="MemberName" rule="*" targetType="string" param=""/>
  <MapEntry type="ScopedName" rule="*" targetType="string"
param="keywords{format=uri}"/>

  <!-- ISO 19107:2003, ISO 19107:2019 -->
  <MapEntry type="GM_Curve" rule="*"
targetType="http://www.opengis.net/to/be/determined/LineString.json" param=
"geometry"/>
  <MapEntry type="GM_MultiCurve" rule="*"
targetType="http://www.opengis.net/to/be/determined/MultiLineString.json"
param="geometry"/>
  <MapEntry type="GM_MultiPoint" rule="*"
targetType="http://www.opengis.net/to/be/determined/MultiPoint.json" param=
"geometry"/>
  <MapEntry type="GM_MultiSurface" rule="*"
targetType="http://www.opengis.net/to/be/determined/MultiPolygon.json"
param="geometry"/>
  <MapEntry type="GM_Point" rule="*"
targetType="http://www.opengis.net/to/be/determined/Point.json" param="geometry"/>
  <MapEntry type="DirectPosition" rule="*"
targetType="http://www.opengis.net/to/be/determined/Point.json" param="geometry"/>
  <MapEntry type="GM_Surface" rule="*"
targetType="http://www.opengis.net/to/be/determined/Polygon.json" param="geometry"/>
  <MapEntry type="GM_Solid" rule="*"
targetType="http://www.opengis.net/to/be/determined/Polyhedron.json" param=
"geometry"/>
  <MapEntry type="GM_MultiSolid" rule="*"
targetType="http://www.opengis.net/to/be/determined/MultiPolyhedron.json"
```

```
param="geometry"/>
41  <MapEntry type="GM_Object" rule="*"
targetType="http://www.opengis.net/to/be/determined/Geometry.json" param="geometry"/>
42
43  <MapEntry type="GeometryLineString" rule="*"
targetType="http://www.opengis.net/to/be/determined/LineString.json" param=
"geometry"/>
44  <MapEntry type="GeometryCurve" rule="*"
targetType="http://www.opengis.net/to/be/determined/LineString.json" param=
"geometry"/>
45  <MapEntry type="GeometryCurveRep" rule="*"
targetType="http://www.opengis.net/to/be/determined/LineString.json" param=
"geometry"/>
46  <MapEntry type="GeometryPoint" rule="*"
targetType="http://www.opengis.net/to/be/determined/Point.json" param="geometry"/>
47  <MapEntry type="GeometryPosition" rule="*"
targetType="http://www.opengis.net/to/be/determined/Point.json" param="geometry"/>
48  <MapEntry type="GeometrySurfaceRep" rule="*"
targetType="http://www.opengis.net/to/be/determined/Polygon.json" param="geometry"/>
49  <MapEntry type="GeometrySurface" rule="*"
targetType="http://www.opengis.net/to/be/determined/Polygon.json" param="geometry"/>
50  <MapEntry type="GeometryPolygon" rule="*"
targetType="http://www.opengis.net/to/be/determined/Polygon.json" param="geometry"/>
51  <MapEntry type="SimplePolygon" rule="*"
targetType="http://www.opengis.net/to/be/determined/Polygon.json" param="geometry"/>
52  <MapEntry type="GeometryCircle" rule="*"
targetType="http://www.opengis.net/to/be/determined/Polygon.json" param="geometry"/>
53  <MapEntry type="GeometryEnvelope" rule="*"
targetType="http://www.opengis.net/to/be/determined/Polygon.json" param="geometry"/>
54  <MapEntry type="SimpleRectangle" rule="*"
targetType="http://www.opengis.net/to/be/determined/Polygon.json" param="geometry"/>
55
56  <!-- ISO 19108:2002 -->
57  <!-- without special values (now) -->
58  <!--<MapEntry type="TM_Instant" rule="*"
targetType="http://www.opengis.net/to/be/determined/InstantRfc3339.json" param=""/>
59  <MapEntry type="TM_TimePeriod" rule="*"
targetType="http://www.opengis.net/to/be/determined/SimpleIntervalRFC3339.json"
param=""/>-->
60  <!-- with special values (now) -->
61  <MapEntry type="TM_Instant" rule="*"
targetType="http://www.opengis.net/to/be/determined/InstantGregorian.json" param=""/>
62  <MapEntry type="TM_TimePeriod" rule="*"
targetType="http://www.opengis.net/to/be/determined/SimpleIntervalGregorian.json"
param=""/>
63  <MapEntry type="TimeInstant" rule="*"
targetType="http://www.opengis.net/to/be/determined/InstantGregorian.json" param=""/>
64  <MapEntry type="TimePeriod" rule="*"
targetType="http://www.opengis.net/to/be/determined/SimpleIntervalGregorian.json"
param=""/>
65
```

```
66  <!-- ISO 19109:2015 -->
67  <MapEntry type="AnyFeature" rule="*"
    targetType="http://www.opengis.net/to/be/determined/Feature.json" param=""/>
68
69  <!-- Extensible Markup Language (XML) Schema - defined in the NAS X-3 UML model
    -->
70  <MapEntry type="NonColonizedName" rule="*" targetType="string" param=""/>
71
72  <!-- ISO 19136-1 Geography Markup Language (GML) - Part 1: Fundamentals - defined
    in the NAS X-3 UML model -->
73  <!-- subpackage: Geography Markup Language (GML) Temporal Reference System -->
74  <MapEntry type="TimeGeometryPrimitive" rule="*"
    targetType="http://www.opengis.net/to/be/determined/TimeGeometryPrimitive.json"
    param=""/>
75  <!-- subpackage: Geography Markup Language (GML) Coordinate Reference System -->
76  <MapEntry type="VerticalCoordRefSystem" rule="*"
    targetType="http://www.opengis.net/to/be/determined/VerticalCoordRefSystem.json"
    param=""/>
77  <!-- subpackage: Geography Markup Language (GML) Reference System -->
78  <MapEntry type="CoordinateReferenceSystem" rule="*"
    targetType="http://www.opengis.net/to/be/determined/CoordinateReferenceSystem.json"
    param=""/>
79  <!-- subpackage: Geography Markup Language (GML) Geometry 0-dimensional and 1-
    dimensional Basic Type -->
80  <MapEntry type="GeometryObjectRepresentation" rule="*"
    targetType="http://www.opengis.net/to/be/determined/Geometry.json" param=
    "geometry"/>
81
82 </mapEntries>
```

B.1. NSG Application Schema

The ShapeChange configuration from Listing 90 derives a JSON Schema from the NSG Application Schema (NAS). The ShapeChange workflow contains a number of transformation steps, which:

- Transform properties with stereotype <<propertyMetadata>> and stereotype <<voidable>> to additional *..._metadata* and *..._nilReason* properties. (for further details, see Transforming Stereotype <<propertyMetadata>> and Generating NilReason Properties for Nillable Properties).
- Transform OCL constraints which restrict the allowed values of property *place* for a given feature type (for further details, see Constraints).
- Transform association classes as recommended in section Association Class.

Listing 91 shows the resulting JSON Schema definition for NAS feature type *River*.

Listing 90. ShapeChange configuration for deriving a JSON Schema for the NAS

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <ShapeChangeConfiguration xmlns="http://www.interactive-
  instruments.de/ShapeChange/Configuration/1.1" xmlns:sc="http://www.interactive-
```

```
    instruments.de/ShapeChange/Configuration/1.1"
xmlns:xi="http://www.w3.org/2001/XInclude"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.interactive-
instruments.de/ShapeChange/Configuration/1.1
file:/C:/NAS/UGAS20/resources/schema/ShapeChangeConfiguration.xsd">
  3  <input id="INPUT">
  4   <parameter name="inputModelType" value="SCXML"/>
  5   <parameter name="inputFile" value="C:/NAS/UGAS20/full_NAS/X-3/NAS_X-
3_SCXML.xml"/>
  6   <parameter name="appSchemaNameRegex" value="NSG Application Schema"/>
  7   <parameter name="publicOnly" value="true"/>
  8   <parameter name="checkingConstraints" value="enabled"/>
  9   <parameter name="codeAbsenceInModelAllowed" value="true"/>
 10   <parameter name="sortedSchemaOutput" value="true"/>
 11   <xi:include href="C:/NAS/UGAS20/GCSR/StandardAliases-GCSR.xml"/>
 12   <packages>
 13    <PackageInfo packageName="NSG Application Schema" ns="http://example.com/nas"
nsabr="nas" xsdDocument="nas.xsd" version="X-3"/>
 14    <PackageInfo packageName="ISO 19103 Conceptual schema language"
ns="http://example.com/iso/19103" nsabr="iso19103" xsdDocument="iso19103.xsd"
version="1.0"/>
 15    <PackageInfo packageName="ISO 19109 Rules for application schema"
ns="http://example.com/iso/19109" nsabr="iso19109" xsdDocument="iso19109.xsd"
version="1.0"/>
 16    <PackageInfo packageName="ISO 19112 Spatial Reference System Using Geographic
Identifier" ns="http://example.com/iso/19112" nsabr="iso19112"
xsdDocument="iso19112.xsd" version="1.0"/>
 17    <PackageInfo packageName="ISO 19115-1 Metadata - Fundamentals"
ns="http://example.com/iso/19115-1" nsabr="iso19115-1" xsdDocument="iso19115-1.xsd"
version="1.0"/>
 18    <PackageInfo packageName="ISO 19115-2 Metadata - Extensions for acquisition and
processing" ns="http://example.com/iso/19115-2" nsabr="iso19115-2"
xsdDocument="iso19115-2.xsd" version="1.0"/>
 19    <PackageInfo packageName="ISO 19119 Services" ns="http://example.com/iso/19119"
nsabr="iso19119" xsdDocument="iso19119.xsd" version="1.0"/>
 20    <PackageInfo packageName="ISO 19123 Schema for coverage geometry and functions"
ns="http://example.com/iso/19123" nsabr="iso19123" xsdDocument="iso19123.xsd"
version="1.0"/>
 21    <PackageInfo packageName="ISO 19157 Data Quality"
ns="http://example.com/iso/19157" nsabr="iso19157" xsdDocument="iso19157.xsd"
version="1.0"/>
 22    <PackageInfo packageName="ISO 19136-1 Geography Markup Language (GML) - Part 1:
Fundamentals" ns="http://example.com/iso/19136-1" nsabr="iso19136-1"
xsdDocument="iso19136-1.xsd" version="1.0"/>
 23    <PackageInfo packageName="ISO 19136-2 Geography Markup Language (GML) - Part 2:
Extended schemas and encoding rules" ns="http://example.com/iso/19136-2"
nsabr="iso19136-2" xsdDocument="iso19136-2.xsd" version="1.0"/>
 24
 25    <PackageInfo packageName="International Association of Oil and Gas Producers"
ns="urn:x-ogp:spec:schema-xsd:EPSG:2.1:dataset" nsabr="iogp" xsdDocument="iogp.xsd"
```

```
version="2.1"/>
 26    <PackageInfo packageName="Extensible Markup Language (XML)"
ns="http://example.com/xml" nsabr="xml" xsdDocument="xml.xsd" version="1.0"/>
 27    <PackageInfo packageName="Sensor Model Language"
ns="http://example.com/ogc/sml" nsabr="sml" xsdDocument="ogcSml.xsd" version="1.0"/>
 28    <PackageInfo packageName="Intelligence Community Metadata" ns="urn:us:gov:ic"
nsabr="icm" xsdDocument="icm.xsd" version="2.0"/>
 29   </packages>
 30  </input>
 31  <log>
 32   <parameter name="reportLevel" value="INFO"/>
 33   <parameter name="logFile" value="C:/NAS/UGAS20/NAS/results/log.xml"/>
 34  </log>
 35  <transformers>
 36   <Transformer
 37
class="de.interactive_instruments.ShapeChange.Transformation.TypeConversion.TypeConver
ter"
 38    input="INPUT" id="TRF_TYPE_CONVERSION" mode="enabled">
 39    <!-- NOTE: Create _metadata and _nilReason properties in the conceptual model,
derived from <<propertyMetadata>> and <<voidable>> properties. -->
 40    <rules>
 41     <ProcessRuleSet name="convert">
 42      <rule name="rule-trf-propertyMetadata-stereotype-to-metadata-property" />
 43      <rule name="rule-trf-nilReason-property-for-nillable-property" />
 44     </ProcessRuleSet>
 45    </rules>
 46   </Transformer>
 47   <Transformer
 48
class="de.interactive_instruments.ShapeChange.Transformation.Constraints.ConstraintCon
verter"
 49    id="TRF_GEOMETRY_RESTRICTION_TO_VALUETYPEOPTIONS_TAGGEDVALUE"
input="TRF_TYPE_CONVERSION" mode="enabled">
 50    <parameters>
 51     <ProcessParameter name="valueTypeRepresentationTypes"
 52      value="PlaceSpecification{PointPositionSpecification,
CurvePositionSpecification, SurfacePositionSpecification, LocationSpecification}"/>
 53     <ProcessParameter name="valueTypeRepresentationConstraintRegex"
 54      value=".*Place Representations Disallowed.*"/>
 55    </parameters>
 56    <rules>
 57     <ProcessRuleSet name="trf">
 58      <rule name="rule-trf-cls-constraints-valueTypeRestrictionToTV-exclusion"/>
 59     </ProcessRuleSet>
 60    </rules>
 61   </Transformer>
 62   <Transformer
class="de.interactive_instruments.ShapeChange.Transformation.Flattening.Flattener"
 63    id="TRF_FLATTEN_CONSTRAINTS"
input="TRF_GEOMETRY_RESTRICTION_TO_VALUETYPEOPTIONS_TAGGEDVALUE"
```

```
 mode="enabled">
 <rules>
  <ProcessRuleSet name="trf">
   <rule name="rule-trf-all-flatten-constraints"/>
  </ProcessRuleSet>
 </rules>
</Transformer>
<Transformer input="TRF_FLATTEN_CONSTRAINTS" id="TRF_LAST"
```

```
class="de.interactive_instruments.ShapeChange.Transformation.Flattening.AssociationCla
ssMapper"/>
 72  </transformers>
 73  <targets>
 74   <Target
class="de.interactive_instruments.ShapeChange.Target.JSON.JsonSchemaTarget"
mode="enabled" inputs="TRF_LAST">
 75    <targetParameter name="outputDirectory"
value="C:/NAS/UGAS20/NAS/results/json_schema"/>
 76    <targetParameter name="sortedOutput" value="true"/>
 77    <targetParameter name="jsonSchemaVersion" value="2019-09"/>
 78    <targetParameter name="jsonBaseUri" value="http://example.org"/>
 79    <targetParameter name="entityTypeName" value="type"/>
 80    <targetParameter name="inlineOrByReferenceDefault" value=
"inlineOrByReference"/>
 81    <targetParameter name="writeMapEntries" value="true"/>
 82    <targetParameter name="defaultEncodingRule" value="nasJsonSchemaRule"/>
 83    <rules>
 84     <EncodingRule name="nasJsonSchemaRule">
 85      <rule name="rule-json-prop-derivedAsReadOnly"/>
 86      <rule name="rule-json-prop-readOnly"/>
 87      <rule name="rule-json-prop-voidable"/>
 88      <rule name="rule-json-prop-initialValueAsDefault"/>
 89      <rule name="rule-json-cls-union-propertyCount"/>
 90      <rule name="rule-json-cls-name-as-entityType"/>
 91      <rule name="rule-json-cls-basictype"/>
 92      <rule name="rule-json-cls-valueTypeOptions"/>
 93     </EncodingRule>
 94    </rules>
 95    <xi:include href="C:\NAS\UGAS20\GCSR\StandardMapEntries_JSON.xml"/>
 96    <xi:include
href="C:\NAS\UGAS20\ISO_Schemas\results\json_schema\INPUT\ISO_19100_series_Standards_m
apEntries.xml"/>
 97    <xi:include
href="C:\NAS\UGAS20\ICM\results\json_schema\INPUT\Intelligence_Community_Metadata_mapE
ntries.xml"/>
 98   </Target>
 99  </targets>
100 </ShapeChangeConfiguration>
```

Listing 91. Example of the NAS JSON Schema - feature type River

```
1 {
2   "$schema": "https://json-schema.org/draft/2019-09/schema",
3   "$id": "http://example.org/nas/NSG_Application_Schema.json",
4   "$defs": {
5     ...
6     "River": {
7       "allOf": [
8         {
```

```
          "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/FeatureEntity"
        },
        {
          "type": "object",
          "properties": {
            "anabranch": {
              "type": [
                "boolean",
                "null"
              ]
            },
            "anabranch_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "anabranch_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidValueReason"
            },
            "area": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MeasureNonNegative"
                }
              ]
            },
            "area_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
```

```
            "area_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidNumValueReason"
            },
            "averageWaterDepth": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MeasureNonNegIntv"
                }
              ]
            },
            "averageWaterDepth_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "averageWaterDepth_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidNumValueReason"
            },
            "bottomMaterialType": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "type": "array",
                  "items": {
                    "$ref":
```

```
"http://example.org/nas/NSG_Application_Schema.json#/$defs/RiverBottomMaterialCodeList
"
 91                   },
 92                   "uniqueItems": true
 93                 }
 94               ]
 95             },
 96             "bottomMaterialType_metadata": {
 97               "oneOf": [
 98                 {
 99                   "type": "string",
100                   "format": "uri"
101                 },
102                 {
103                   "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
104                 }
105               ]
106             },
107             "bottomMaterialType_nilReason": {
108               "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidValueReason"
109             },
110             "conspicuousAirCategory": {
111               "oneOf": [
112                 {
113                   "type": "null"
114                 },
115                 {
116                   "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/RiverConspicuousAirCategory
Type"
117                 }
118               ]
119             },
120             "conspicuousAirCategory_metadata": {
121               "oneOf": [
122                 {
123                   "type": "string",
124                   "format": "uri"
125                 },
126                 {
127                   "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
128                 }
129               ]
130             },
131             "conspicuousAirCategory_nilReason": {
132               "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidValueReason"
```

```
133             },
134             "conspicuousGroundCategory": {
135               "oneOf": [
136                 {
137                   "type": "null"
138                 },
139                 {
140                   "$ref":
```

```
"http://example.org/nas/NSG_Application_Schema.json#/$defs/ConspicuousGroundCategoryTy
pe"
 141                 }
 142               ]
 143             },
 144             "conspicuousGroundCategory_metadata": {
 145               "oneOf": [
 146                 {
 147                   "type": "string",
 148                   "format": "uri"
 149                 },
 150                 {
 151                   "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
 152                 }
 153               ]
 154             },
 155             "conspicuousGroundCategory_nilReason": {
 156               "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidValueReason"
 157             },
 158             "controllingAuthority": {
 159               "oneOf": [
 160                 {
 161                   "type": "null"
 162                 },
 163                 {
 164                   "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/RiverControllingAuthorityCo
deList"
 165                 }
 166               ]
 167             },
 168             "controllingAuthority_metadata": {
 169               "oneOf": [
 170                 {
 171                   "type": "string",
 172                   "format": "uri"
 173                 },
 174                 {
 175                   "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
 176                 }
 177               ]
 178             },
 179             "controllingAuthority_nilReason": {
 180               "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidValueReason"
 181             },
 182             "coveredDrain": {
```

```
              "type": [
                "boolean",
                "null"
              ]
            },
            "coveredDrain_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "coveredDrain_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidAttributeReason"
            },
            "coveredDrainLength": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MeasureNonNegIntv"
                }
              ]
            },
            "coveredDrainLength_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "coveredDrainLength_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidNumValueReason"
            },
            "deepDepthBelowSurfLevel": {
              "oneOf": [
                {
```

```
                "type": "null"
              },
              {
                "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MeasureNonNegative"
              }
            ]
          },
          "deepDepthBelowSurfLevel_metadata": {
            "oneOf": [
              {
                "type": "string",
                "format": "uri"
              },
              {
                "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
              }
            ]
          },
          "deepDepthBelowSurfLevel_nilReason": {
            "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidNumValueReason"
          },
          "directivity": {
            "oneOf": [
              {
                "type": "null"
              },
              {
                "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/RiverDirectivityType"
              }
            ]
          },
          "directivity_metadata": {
            "oneOf": [
              {
                "type": "string",
                "format": "uri"
              },
              {
                "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
              }
            ]
          },
          "directivity_nilReason": {
            "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidValueReason"
          },
```

```
            "floodlit": {
              "type": [
                "boolean",
                "null"
              ]
            },
            "floodlit_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "floodlit_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidAttributeReason"
            },
            "highestElevation": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "type": "array",
                  "items": {
                    "$ref":
```

```
"http://example.org/nas/NSG_Application_Schema.json#/$defs/ElevationDatumAccuracy"
303                   },
304                   "uniqueItems": true
305                 }
306               ]
307             },
308             "highestElevation_metadata": {
309               "oneOf": [
310                 {
311                   "type": "string",
312                   "format": "uri"
313                 },
314                 {
315                   "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
316                 }
317               ]
318             },
319             "highestElevation_nilReason": {
320               "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidNumValueReason"
321             },
322             "highWaterMonthInterval": {
323               "oneOf": [
324                 {
325                   "type": "null"
326                 },
327                 {
328                   "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MonthIntervalStrucText"
329                 }
330               ]
331             },
332             "highWaterMonthInterval_metadata": {
333               "oneOf": [
334                 {
335                   "type": "string",
336                   "format": "uri"
337                 },
338                 {
339                   "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
340                 }
341               ]
342             },
343             "highWaterMonthInterval_nilReason": {
344               "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidAttributeReason"
345             },
346             "leastDepthBelowSurfLevel": {
```

```
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MeasureNonNegative"
                }
              ]
            },
            "leastDepthBelowSurfLevel_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "leastDepthBelowSurfLevel_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidNumValueReason"
            },
            "length": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MeasureNonNegIntv"
                }
              ]
            },
            "length_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "length_nilReason": {
              "$ref":
```

```
    "http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidNumValueReason"
393             },
394             "lowWaterMonthInterval": {
395               "oneOf": [
396                 {
397                   "type": "null"
398                 },
399                 {
400                   "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MonthIntervalStrucText"
401                 }
402               ]
403             },
404             "lowWaterMonthInterval_metadata": {
405               "oneOf": [
406                 {
407                   "type": "string",
408                   "format": "uri"
409                 },
410                 {
411                   "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
412                 }
413               ]
414             },
415             "lowWaterMonthInterval_nilReason": {
416               "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidAttributeReason"
417             },
418             "maintained": {
419               "type": [
420                 "boolean",
421                 "null"
422               ]
423             },
424             "maintained_metadata": {
425               "oneOf": [
426                 {
427                   "type": "string",
428                   "format": "uri"
429                 },
430                 {
431                   "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
432                 }
433               ]
434             },
435             "maintained_nilReason": {
436               "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidAttributeReason"
437             },
```

```
438             "navigabilityInformation": {
439               "oneOf": [
440                 {
441                   "type": "null"
442                 },
443                 {
444                   "$ref":
```

```
"http://example.org/nas/NSG_Application_Schema.json#/$defs/RiverNavigabilityInformatio
nCodeList"
                  }
                ]
              },
              "navigabilityInformation_metadata": {
                "oneOf": [
                  {
                    "type": "string",
                    "format": "uri"
                  },
                  {
                    "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                  }
                ]
              },
              "navigabilityInformation_nilReason": {
                "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidValueReason"
              },
              "navigationLandmark": {
                "type": [
                  "boolean",
                  "null"
                ]
              },
              "navigationLandmark_metadata": {
                "oneOf": [
                  {
                    "type": "string",
                    "format": "uri"
                  },
                  {
                    "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                  }
                ]
              },
              "navigationLandmark_nilReason": {
                "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidAttributeReason"
              },
              "operatingCycle": {
                "oneOf": [
                  {
                    "type": "null"
                  },
                  {
                    "$ref":
```

```
"http://example.org/nas/NSG_Application_Schema.json#/$defs/RiverOperatingCycleCodeList
"
 489                 }
 490               ]
 491             },
 492             "operatingCycle_metadata": {
 493               "oneOf": [
 494                 {
 495                   "type": "string",
 496                   "format": "uri"
 497                 },
 498                 {
 499                   "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
 500                 }
 501               ]
 502             },
 503             "operatingCycle_nilReason": {
 504               "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidValueReason"
 505             },
 506             "operatingRestriction": {
 507               "oneOf": [
 508                 {
 509                   "type": "null"
 510                 },
 511                 {
 512                   "type": "array",
 513                   "items": {
 514                     "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/RiverOperatingRestrictionCo
deList"
 515                   },
 516                   "uniqueItems": true
 517                 }
 518               ]
 519             },
 520             "operatingRestriction_metadata": {
 521               "oneOf": [
 522                 {
 523                   "type": "string",
 524                   "format": "uri"
 525                 },
 526                 {
 527                   "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
 528                 }
 529               ]
 530             },
 531             "operatingRestriction_nilReason": {
 532               "$ref":
```

```
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidValueReason"
            },
            "partialFeatureIndicator": {
              "type": [
                "boolean",
                "null"
              ]
            },
            "partialFeatureIndicator_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "partialFeatureIndicator_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidAttributeReason"
            },
            "pedestrianFordable": {
              "type": [
                "boolean",
                "null"
              ]
            },
            "pedestrianFordable_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "pedestrianFordable_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidAttributeReason"
            },
            "periodRestrictMonth": {
              "oneOf": [
                {
                  "type": "null"
                },
```

```
579                 {
580                   "$ref":
```

```
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MonthIntervalStrucText"
                    }
                  ]
                },
                "periodRestrictMonth_metadata": {
                  "oneOf": [
                    {
                      "type": "string",
                      "format": "uri"
                    },
                    {
                      "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                    }
                  ]
                },
                "periodRestrictMonth_nilReason": {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidAttributeReason"
                },
                "portAccess": {
                  "type": [
                    "boolean",
                    "null"
                  ]
                },
                "portAccess_metadata": {
                  "oneOf": [
                    {
                      "type": "string",
                      "format": "uri"
                    },
                    {
                      "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                    }
                  ]
                },
                "portAccess_nilReason": {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidAttributeReason"
                },
                "predominantAvWaterVel": {
                  "oneOf": [
                    {
                      "type": "null"
                    },
                    {
                      "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MeasureInterval"
```

```
                }
              ]
            },
            "predominantAvWaterVel_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "predominantAvWaterVel_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidNumValueReason"
            },
            "predominantGapWidth": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MeasureNonNegIntv"
                }
              ]
            },
            "predominantGapWidth_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "predominantGapWidth_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidNumValueReason"
            },
            "predominantMaxWaterDepth": {
              "oneOf": [
                {
                  "type": "null"
                },
```

```
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MeasureNonNegIntv"
                }
              ]
            },
            "predominantMaxWaterDepth_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "predominantMaxWaterDepth_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidNumValueReason"
            },
            "predominantMaxWaterVel": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MeasureInterval"
                }
              ]
            },
            "predominantMaxWaterVel_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "predominantMaxWaterVel_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidNumValueReason"
            },
            "predominantMinWaterDepth": {
              "oneOf": [
```

```
716                 {
717                   "type": "null"
718                 },
719                 {
720                   "$ref":
    "http://example.org/nas/NSG_Application_Schema.json#/$defs/MeasureNonNegIntv"
721                 }
722               ]
723             },
724             "predominantMinWaterDepth_metadata": {
725               "oneOf": [
726                 {
727                   "type": "string",
728                   "format": "uri"
729                 },
730                 {
731                   "$ref":
    "http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
732                 }
733               ]
734             },
735             "predominantMinWaterDepth_nilReason": {
736               "$ref":
    "http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidNumValueReason"
737             },
738             "predominantMinWaterVel": {
739               "oneOf": [
740                 {
741                   "type": "null"
742                 },
743                 {
744                   "$ref":
    "http://example.org/nas/NSG_Application_Schema.json#/$defs/MeasureInterval"
745                 }
746               ]
747             },
748             "predominantMinWaterVel_metadata": {
749               "oneOf": [
750                 {
751                   "type": "string",
752                   "format": "uri"
753                 },
754                 {
755                   "$ref":
    "http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
756                 }
757               ]
758             },
759             "predominantMinWaterVel_nilReason": {
760               "$ref":
    "http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidNumValueReason"
```

```
            },
            "predominantWaterDepth": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MeasureNonNegIntv"
                }
              ]
            },
            "predominantWaterDepth_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "predominantWaterDepth_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidNumValueReason"
            },
            "safeHorizontalClearance": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MeasureNonNegative"
                }
              ]
            },
            "safeHorizontalClearance_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
```

```
            "safeHorizontalClearance_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidNumValueReason"
            },
            "seasonallyFrozen": {
              "type": [
                "boolean",
                "null"
              ]
            },
            "seasonallyFrozen_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "seasonallyFrozen_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidAttributeReason"
            },
            "shorelineDelineated": {
              "type": [
                "boolean",
                "null"
              ]
            },
            "shorelineDelineated_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "shorelineDelineated_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidAttributeReason"
            },
            "specifiedComplianceType": {
              "oneOf": [
                {
```

```
853                 "type": "null"
854               },
855               {
856                 "$ref":
```

```
"http://example.org/nas/NSG_Application_Schema.json#/$defs/RiverSpecifiedComplianceCodeList"
                }
              ]
            },
            "specifiedComplianceType_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "specifiedComplianceType_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidValueReason"
            },
            "terrainGapWidth": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MeasureNonNegative"
                }
              ]
            },
            "terrainGapWidth_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "terrainGapWidth_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidNumValueReason"
            },
            "tideInfluenced": {
              "type": [
```

```
                "boolean",
                "null"
              ]
            },
            "tideInfluenced_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "tideInfluenced_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidAttributeReason"
            },
            "verticalRelativeLocation": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "$ref":
```

```
"http://example.org/nas/NSG_Application_Schema.json#/$defs/RiverVerticalRelativeLocati
onType"
 925                 }
 926               ]
 927             },
 928             "verticalRelativeLocation_metadata": {
 929               "oneOf": [
 930                 {
 931                   "type": "string",
 932                   "format": "uri"
 933                 },
 934                 {
 935                   "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
 936                 }
 937               ]
 938             },
 939             "verticalRelativeLocation_nilReason": {
 940               "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidValueReason"
 941             },
 942             "watercourseChannelType": {
 943               "oneOf": [
 944                 {
 945                   "type": "null"
 946                 },
 947                 {
 948                   "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/RiverWatercourseChannelCode
List"
 949                 }
 950               ]
 951             },
 952             "watercourseChannelType_metadata": {
 953               "oneOf": [
 954                 {
 955                   "type": "string",
 956                   "format": "uri"
 957                 },
 958                 {
 959                   "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
 960                 }
 961               ]
 962             },
 963             "watercourseChannelType_nilReason": {
 964               "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidValueReason"
 965             },
 966             "watercourseMorphology": {
```

```
967               "oneOf": [
968                 {
969                   "type": "null"
970                 },
971                 {
972                   "$ref":
```

```
"http://example.org/nas/NSG_Application_Schema.json#/$defs/RiverWatercourseMorphologyCodeList"
                }
              ]
            },
            "watercourseMorphology_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "watercourseMorphology_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidValueReason"
            },
            "width": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MeasureNonNegIntv"
                }
              ]
            },
            "width_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "width_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidNumValueReason"
            },
            "boundingWaterbodyBank": {
              "oneOf": [
```

```
                {
                  "type": "null"
                },
                {
                  "type": "array",
                  "items": {
                    "oneOf": [
                      {
                        "type": "string",
                        "format": "uri"
                      },
                      {
                        "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/InlandWaterbodyBank"
                      }
                    ]
                  },
                  "uniqueItems": true
                }
              ]
            },
            "boundingWaterbodyBank_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "boundingWaterbodyBank_nilReason": {
              "$ref":
```

```
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidRelationshipReason"
            },
            "portion": {
              "type": "array",
              "items": {
                "oneOf": [
                  {
                    "type": "string",
                    "format": "uri"
                  },
                  {
                    "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/River"
                  }
                ]
              },
              "uniqueItems": true
            },
            "relatedHydroVertPosInfo": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/HydroVertPositioningInfo"
                }
              ]
            },
            "relatedHydroVertPosInfo_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "relatedHydroVertPosInfo_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidRelationshipReason"
            },
            "relatedMarNavLandmarkInfo": {
```

```
1094                "oneOf": [
1095                  {
1096                    "type": "null"
1097                  },
1098                  {
1099                    "type": "string",
1100                    "format": "uri"
1101                  },
1102                  {
1103                    "$ref":
```

```
"http://example.org/nas/NSG_Application_Schema.json#/$defs/MaritimeNavLandmarkInfo"
                }
              ]
            },
            "relatedMarNavLandmarkInfo_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "relatedMarNavLandmarkInfo_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidRelationshipReason"
            },
            "relatedWaterResourceInfo": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/WaterResourceInfo"
                }
              ]
            },
            "relatedWaterResourceInfo_metadata": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/PropertyValueMeta"
                }
              ]
            },
            "relatedWaterResourceInfo_nilReason": {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/VoidRelationshipReason"
```

```
            },
            "whole": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/River"
                }
              ]
            },
            "place": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "type": "array",
                  "items": {
                    "oneOf": [
                      {
                        "type": "string",
                        "format": "uri"
                      },
                      {
                        "allOf": [
                          {
                            "$ref":
```

```
"http://example.org/nas/NSG_Application_Schema.json#/$defs/FeaturePlaceRelationship"
                            },
                            {
                              "type": "object",
                              "properties": {
                                "place": {
                                  "oneOf": [
                                    {
                                      "type": "string",
                                      "format": "uri"
                                    },
                                    {
                                      "if": {
                                        "properties": {
                                          "type": {
                                            "const": "CurvePositionSpecification"
                                          }
                                        }
                                      },
                                      "then": {
                                        "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/CurvePositionSpecification"
                                      },
                                      "else": {
                                        "if": {
                                          "properties": {
                                            "type": {
                                              "const": "LocationSpecification"
                                            }
                                          }
                                        },
                                        "then": {
                                          "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/LocationSpecification"
                                        },
                                        "else": {
                                          "if": {
                                            "properties": {
                                              "type": {
                                                "const":
"SurfacePositionSpecification"
                                              }
                                            }
                                          },
                                          "then": {
                                            "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/SurfacePositionSpecificatio
n"
                                          },
                                          "else": false
```

```
                                    }
                                  }
                                }
                              ]
                            }
                          }
                        }
                      ]
                    }
                  ]
                }
              }
            ]
          }
        }
      ]
    },
    ...
  }
}
```

B.2. ISO 19100-series Schemas

The ShapeChange configuration from Listing 92 derives a JSON Schema from a subset of the ISO schemas contained in the NAS X-3 conceptual model. The subset contains the following schemas:

- ISO 19112 Spatial Reference System Using Geographic Identifier
- ISO 19115-1 Metadata - Fundamentals
- ISO 19115-2 Metadata - Extensions for acquisition and processing
- ISO 19157 Data Quality

NOTE	ISO 19112 has been included because the NAS depends on it (SI_LocationInstance is supertype of NAS GeoNameCollection).

Apparently, the ISO schemas contained in the NAS X-3 model represent modified versions of the original ISO schemas, because they depend on the following NAS types:

- NSG Application Schema::Foundation::Resource Metadata::IANACharset
- NSG Application Schema::Foundation::Resource Metadata::ResourceConstraints
- NSG Application Schema::Foundation::General Datatype::IdentifierNamespaceCodeList
- NSG Application Schema::Foundation::General Resource Model::EntityCollection
- NSG Application Schema::Foundation::General Resource Model::RecordSet

The ShapeChange configuration therefore includes a reference to a map entries file that contains the JSON Schema mappings for these NAS types.

Listing 93 shows the resulting JSON Schema definition for type *MD_Metadata* (from ISO 19115-1).

Listing 92. ShapeChange configuration for deriving a JSON Schema for ISO schemas contained in the NAS

```
<?xml version="1.0" encoding="UTF-8"?>
<ShapeChangeConfiguration
 xmlns="http://www.interactive-instruments.de/ShapeChange/Configuration/1.1"
 xmlns:sc="http://www.interactive-instruments.de/ShapeChange/Configuration/1.1"
 xmlns:xi="http://www.w3.org/2001/XInclude"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.interactive-
instruments.de/ShapeChange/Configuration/1.1
file:/C:/NAS/UGAS20/resources/schema/ShapeChangeConfiguration.xsd">
 <input>
  <parameter name="inputModelType" value="SCXML"/>
  <parameter name="inputFile" value="C:/NAS/UGAS20/full_NAS/X-3/NAS_X-
3_SCXML.xml"/>
  <parameter name="appSchemaNameRegex" value="ISO 19100.*"/>
  <parameter name="excludedPackages"
   value="ISO 19103 Conceptual schema language,ISO 19109 Rules for application
schema,ISO 19119 Services,ISO 19123 Schema for coverage geometry and functions,ISO
19136-1 Geography Markup Language (GML) - Part 1: Fundamentals,ISO 19136-2
Geography Markup Language (GML) - Part 2: Extended schemas and encoding rules"/>
  <!-- We must not exclude "ISO 19112 Spatial Reference System Using Geographic
Identifier" because the NAS depends on it (SI_LocationInstance is supertype of NAS
GeoNameCollection). -->
  <!-- NOTE: NAS X-3 does not define any classes in the packages of ISO 19107, ISO
19108, and ISO 19111. Therefore, there is no need to exclude these packages. -->
  <parameter name="publicOnly" value="true"/>
  <parameter name="checkingConstraints" value="disabled"/>
  <parameter name="sortedSchemaOutput" value="true"/>
  <xi:include href="C:/NAS/UGAS20/GCSR/StandardAliases-GCSR.xml"/>
  <packages>
   <PackageInfo packageName="ISO 19100-series Standards"
ns="https://example.org/iso" nsabr="iso"/>
  </packages>
 </input>
 <log>
  <parameter name="reportLevel" value="INFO"/>
  <parameter name="logFile" value="C:/NAS/UGAS20/ISO_Schemas/results/log.xml"/>
 </log>
 <targets>
  <Target
class="de.interactive_instruments.ShapeChange.Target.JSON.JsonSchemaTarget"
mode="enabled">
   <targetParameter name="outputDirectory"
value="C:/NAS/UGAS20/ISO_Schemas/results/json_schema"/>
   <targetParameter name="sortedOutput" value="true"/>
   <targetParameter name="jsonSchemaVersion" value="2019-09"/>
   <targetParameter name="jsonBaseUri" value="http://example.org"/>
   <targetParameter name="entityTypeName" value="type"/>
```

```
34    <targetParameter name="inlineOrByReferenceDefault" value="inlineOrByReference"/>
35    <targetParameter name="writeMapEntries" value="true"/>
36    <targetParameter name="defaultEncodingRule" value="isoJsonSchemaRule"/>
37    <rules>
38     <EncodingRule name="isoJsonSchemaRule">
39      <rule name="rule-json-prop-derivedAsReadOnly"/>
40      <rule name="rule-json-prop-readOnly"/>
41      <rule name="rule-json-prop-voidable"/>
42      <rule name="rule-json-prop-initialValueAsDefault"/>
43      <!-- NOTE: ISO 19115 and ISO 19157 do not use type discriminator unions.
    Therefore, we use the property count based union encoding. -->
44      <rule name="rule-json-cls-union-propertyCount"/>
45      <rule name="rule-json-cls-name-as-entityType"/>
46     </EncodingRule>
47    </rules>
48    <xi:include href="C:/NAS/UGAS20/GCSR/StandardMapEntries_JSON.xml"/>
49    <xi:include
50
```

```
href="C:\NAS\UGAS20\NAS\results\json_schema\TRF_LAST\NSG_Application_Schema_mapEntries
.xml"/>
  </Target>
 </targets>
</ShapeChangeConfiguration>
```

Listing 93. Example of ISO JSON Schema - type MD_Metadata - as defined in the NAS X-3 UML model

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://example.org/iso/ISO_19100-series_Standards.json",
  "$defs": {
    ...
    "MD_Metadata": {
      "type": "object",
      "properties": {
        "type": {
          "type": "string"
        },
        "alternativeMetadataReference": {
          "type": "array",
          "items": {
            "oneOf": [
              {
                "type": "string",
                "format": "uri"
              },
              {
                "$ref": "http://example.org/iso/ISO_19100-
series_Standards.json#/$defs/CI_Citation"
              }
            ]
          },
          "uniqueItems": true
        },
        "defaultLocale": {
          "$ref": "http://example.org/iso/ISO_19100-
series_Standards.json#/$defs/PT_Locale"
        },
        "metaContact": {
          "type": "array",
          "minItems": 1,
          "items": {
            "oneOf": [
              {
                "type": "string",
                "format": "uri"
              },
              {
                "$ref": "http://example.org/iso/ISO_19100-
```

```
    series_Standards.json#/$defs/CI_Responsibility"
41                }
42              ]
43            },
44            "uniqueItems": true
45          },
46          "metadataIdentifier": {
47            "$ref": "http://example.org/iso/ISO_19100-
series_Standards.json#/$defs/MD_Identifier"
48          },
49          "metadataLinkage": {
50            "type": "array",
51            "items": {
52              "$ref": "http://example.org/iso/ISO_19100-
series_Standards.json#/$defs/CI_OnlineResource"
53            },
54            "uniqueItems": true
55          },
56          "metadataProfile": {
57            "type": "array",
58            "items": {
59              "oneOf": [
60                {
61                  "type": "string",
62                  "format": "uri"
63                },
64                {
65                  "$ref": "http://example.org/iso/ISO_19100-
series_Standards.json#/$defs/CI_Citation"
66                }
67              ]
68            },
69            "uniqueItems": true
70          },
71          "metadataStandard": {
72            "type": "array",
73            "items": {
74              "oneOf": [
75                {
76                  "type": "string",
77                  "format": "uri"
78                },
79                {
80                  "$ref": "http://example.org/iso/ISO_19100-
series_Standards.json#/$defs/CI_Citation"
81                }
82              ]
83            },
84            "uniqueItems": true
85          },
86          "otherLocale": {
```

```
          "type": "array",
          "items": {
            "$ref": "http://example.org/iso/ISO_19100-series_Standards.json#/$defs/PT_Locale"
          },
          "uniqueItems": true
        },
        "parentMetadata": {
          "oneOf": [
            {
              "type": "string",
              "format": "uri"
            },
            {
              "$ref": "http://example.org/iso/ISO_19100-series_Standards.json#/$defs/CI_Citation"
            }
          ]
        },
        "resourceMetadataDateTime": {
          "type": "array",
          "minItems": 1,
          "items": {
            "$ref": "http://example.org/iso/ISO_19100-series_Standards.json#/$defs/CI_Date"
          },
          "uniqueItems": true
        },
        "applicationSchemaInfo": {
          "type": "array",
          "items": {
            "oneOf": [
              {
                "type": "string",
                "format": "uri"
              },
              {
                "$ref": "http://example.org/iso/ISO_19100-series_Standards.json#/$defs/MD_ApplicationSchemaInfo"
              }
            ]
          },
          "uniqueItems": true
        },
        "contentInfo": {
          "type": "array",
          "items": {
            "oneOf": [
              {
                "type": "string",
                "format": "uri"
```

```
            },
            {
              "$ref": "http://example.org/iso/ISO_19100-
series_Standards.json#/$defs/MD_ContentInformation"
            }
          ]
        },
        "uniqueItems": true
      },
      "dataQualityInfo": {
        "type": "array",
        "items": {
          "oneOf": [
            {
              "type": "string",
              "format": "uri"
            },
            {
              "$ref": "http://example.org/iso/ISO_19100-
series_Standards.json#/$defs/DQ_DataQuality"
            }
          ]
        },
        "uniqueItems": true
      },
      "describesEntityCollection": {
        "type": "array",
        "items": {
          "oneOf": [
            {
              "type": "string",
              "format": "uri"
            },
            {
              "$ref":
"http://example.org/nas/NSG_Application_Schema.json#/$defs/EntityCollection"
            }
          ]
        },
        "uniqueItems": true
      },
      "describesRecordSet": {
        "type": "array",
        "items": {
          "oneOf": [
            {
              "type": "string",
              "format": "uri"
            },
            {
              "$ref":
```

```
    "http://example.org/nas/NSG_Application_Schema.json#/$defs/RecordSet"
                }
              ]
            },
            "uniqueItems": true
          },
          "distributionInfo": {
            "type": "array",
            "items": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref": "http://example.org/iso/ISO_19100-
series_Standards.json#/$defs/MD_Distribution"
                }
              ]
            },
            "uniqueItems": true
          },
          "identificationInfo": {
            "type": "array",
            "minItems": 1,
            "items": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref": "http://example.org/iso/ISO_19100-
series_Standards.json#/$defs/MD_Identification"
                }
              ]
            },
            "uniqueItems": true
          },
          "metadataConstraints": {
            "type": "array",
            "items": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "uri"
                },
                {
                  "$ref": "http://example.org/iso/ISO_19100-
series_Standards.json#/$defs/MD_Constraints"
                }
```

```
            ]
          },
          "uniqueItems": true
        },
        "metadataMaintenance": {
          "oneOf": [
            {
              "type": "string",
              "format": "uri"
            },
            {
              "$ref": "http://example.org/iso/ISO_19100-
series_Standards.json#/$defs/MD_MaintenanceInformation"
            }
          ]
        },
        "metadataScope": {
          "type": "array",
          "items": {
            "oneOf": [
              {
                "type": "string",
                "format": "uri"
              },
              {
                "$ref": "http://example.org/iso/ISO_19100-
series_Standards.json#/$defs/MD_MetadataScope"
              }
            ]
          },
          "uniqueItems": true
        },
        "portrayalCatalogueInfo": {
          "type": "array",
          "items": {
            "oneOf": [
              {
                "type": "string",
                "format": "uri"
              },
              {
                "$ref": "http://example.org/iso/ISO_19100-
series_Standards.json#/$defs/MD_PortrayalCatalogueRef"
              }
            ]
          },
          "uniqueItems": true
        },
        "referenceSystemInfo": {
          "type": "array",
          "items": {
```

```
            "oneOf": [
              {
                "type": "string",
                "format": "uri"
              },
              {
                "$ref": "http://example.org/iso/ISO_19100-
series_Standards.json#/$defs/MD_ReferenceSystem"
              }
            ]
          },
          "uniqueItems": true
        },
        "resourceLineage": {
          "type": "array",
          "items": {
            "oneOf": [
              {
                "type": "string",
                "format": "uri"
              },
              {
                "$ref": "http://example.org/iso/ISO_19100-
series_Standards.json#/$defs/LI_Lineage"
              }
            ]
          },
          "uniqueItems": true
        },
        "spatialRepresentationInfo": {
          "type": "array",
          "items": {
            "oneOf": [
              {
                "type": "string",
                "format": "uri"
              },
              {
                "$ref": "http://example.org/iso/ISO_19100-
series_Standards.json#/$defs/MD_SpatialRepresentation"
              }
            ]
          },
          "uniqueItems": true
        }
      },
      "required": [
        "identificationInfo",
        "metaContact",
        "resourceMetadataDateTime",
        "type"
```

```
      ]
    },
    ...
  }
}
```

B.3. U.S. Intelligence Community Metadata Schema

The ShapeChange configuration from Listing 94 derives a JSON Schema from the U.S. Intelligence Community Metadata schema.

Listing 95 shows the resulting JSON Schema definition for type *SecurityAttributesGroupType*.

Listing 94. ShapeChange configuration for deriving a JSON Schema for the U.S. Intelligence Community Metadata schema

```
<?xml version="1.0" encoding="UTF-8"?>
<ShapeChangeConfiguration xmlns="http://www.interactive-instruments.de/ShapeChange/Configuration/1.1" xmlns:sc="http://www.interactive-instruments.de/ShapeChange/Configuration/1.1"
xmlns:xi="http://www.w3.org/2001/XInclude"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.interactive-instruments.de/ShapeChange/Configuration/1.1
file:/C:/NAS/UGAS20/resources/schema/ShapeChangeConfiguration.xsd">
 <input>
  <parameter name="inputModelType" value="SCXML"/>
  <parameter name="inputFile" value="C:/NAS/UGAS20/full_NAS/X-3/NAS_X-3_SCXML.xml"/>
  <parameter name="appSchemaNameRegex" value="Intelligence Community Metadata"/>
  <parameter name="publicOnly" value="true"/>
  <parameter name="checkingConstraints" value="disabled"/>
  <parameter name="sortedSchemaOutput" value="true"/>
  <xi:include href="C:/NAS/UGAS20/GCSR/StandardAliases-GCSR.xml"/>
  <packages>
   <PackageInfo packageName="NSG Application Schema" ns="http://example.com/nas" nsabr="nas" xsdDocument="nas.xsd" version="X-3"/>
   <PackageInfo packageName="ISO 19103 Conceptual schema language" ns="http://example.com/iso/19103" nsabr="iso19103" xsdDocument="iso19103.xsd" version="1.0"/>
   <PackageInfo packageName="ISO 19109 Rules for application schema" ns="http://example.com/iso/19109" nsabr="iso19109" xsdDocument="iso19109.xsd" version="1.0"/>
   <PackageInfo packageName="ISO 19112 Spatial Reference System Using Geographic Identifier" ns="http://example.com/iso/19112" nsabr="iso19112" xsdDocument="iso19112.xsd" version="1.0"/>
   <PackageInfo packageName="ISO 19115-1 Metadata - Fundamentals" ns="http://example.com/iso/19115-1" nsabr="iso19115-1" xsdDocument="iso19115-1.xsd" version="1.0"/>
   <PackageInfo packageName="ISO 19115-2 Metadata - Extensions for acquisition and
```

```
   processing" ns="http://example.com/iso/19115-2" nsabr="iso19115-2"
xsdDocument="iso19115-2.xsd" version="1.0"/>
18    <PackageInfo packageName="ISO 19119 Services" ns="http://example.com/iso/19119"
nsabr="iso19119" xsdDocument="iso19119.xsd" version="1.0"/>
19    <PackageInfo packageName="ISO 19123 Schema for coverage geometry and functions"
ns="http://example.com/iso/19123" nsabr="iso19123" xsdDocument="iso19123.xsd"
version="1.0"/>
20    <PackageInfo packageName="ISO 19157 Data Quality"
ns="http://example.com/iso/19157" nsabr="iso19157" xsdDocument="iso19157.xsd"
version="1.0"/>
21    <PackageInfo packageName="ISO 19136-1 Geography Markup Language (GML) - Part 1:
Fundamentals" ns="http://example.com/iso/19136-1" nsabr="iso19136-1"
xsdDocument="iso19136-1.xsd" version="1.0"/>
22    <PackageInfo packageName="ISO 19136-2 Geography Markup Language (GML) - Part 2:
Extended schemas and encoding rules" ns="http://example.com/iso/19136-2"
nsabr="iso19136-2" xsdDocument="iso19136-2.xsd" version="1.0"/>
23
24    <PackageInfo packageName="International Association of Oil and Gas Producers"
ns="urn:x-ogp:spec:schema-xsd:EPSG:2.1:dataset" nsabr="iogp" xsdDocument="iogp.xsd"
version="2.1"/>
25    <PackageInfo packageName="Extensible Markup Language (XML)"
ns="http://example.com/xml" nsabr="xml" xsdDocument="xml.xsd" version="1.0"/>
26    <PackageInfo packageName="Sensor Model Language" ns="http://example.com/ogc/sml"
nsabr="sml" xsdDocument="ogcSml.xsd" version="1.0"/>
27    <PackageInfo packageName="Intelligence Community Metadata" ns="urn:us:gov:ic"
nsabr="icm" xsdDocument="icm.xsd" version="2.0"/>
28   </packages>
29  </input>
30  <log>
31   <parameter name="reportLevel" value="INFO"/>
32   <parameter name="logFile" value="C:/NAS/UGAS20/ICM/results/log.xml"/>
33  </log>
34  <targets>
35   <Target
class="de.interactive_instruments.ShapeChange.Target.JSON.JsonSchemaTarget"
mode="enabled">
36    <targetParameter name="outputDirectory"
value="C:/NAS/UGAS20/ICM/results/json_schema"/>
37    <targetParameter name="sortedOutput" value="true"/>
38    <targetParameter name="jsonSchemaVersion" value="2019-09"/>
39    <targetParameter name="jsonBaseUri" value="http://example.org"/>
40    <targetParameter name="entityTypeName" value="type"/>
41    <targetParameter name="inlineOrByReferenceDefault" value="inlineOrByReference"/>
42    <targetParameter name="writeMapEntries" value="true"/>
43    <targetParameter name="defaultEncodingRule" value="usicmJsonSchemaRule"/>
44    <rules>
45     <EncodingRule name="usicmJsonSchemaRule">
46      <rule name="rule-json-prop-derivedAsReadOnly"/>
47      <rule name="rule-json-prop-readOnly"/>
48      <rule name="rule-json-prop-voidable"/>
49      <rule name="rule-json-prop-initialValueAsDefault"/>
```

```
    <rule name="rule-json-cls-union-propertyCount"/>
    <rule name="rule-json-cls-name-as-entityType"/>
    <rule name="rule-json-cls-basictype"/>
   </EncodingRule>
  </rules>
  <xi:include href="C:/NAS/UGAS20/GCSR/StandardMapEntries_JSON.xml"/>
 </Target>
</targets>
</ShapeChangeConfiguration>
```

Listing 95. Example of U.S. ICM JSON Schema - type SecurityAttributesGroupType - as defined in the NAS X-3 UML model

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://example.org/icm/Intelligence_Community_Metadata.json",
  "$defs": {
    ...
    "SecurityAttributesGroupType": {
      "type": "object",
      "properties": {
        "type": {
          "type": "string"
        },
        "resClassification": {
          "$ref":
```

```
"http://example.org/icm/Intelligence_Community_Metadata.json#/$defs/ResClassificationCodeBasedText"
        },
        "resOwnerProducer": {
          "$ref":
"http://example.org/icm/Intelligence_Community_Metadata.json#/$defs/ResOwnerProducerCodeBasedText"
        },
        "resJoint": {
          "type": "boolean"
        },
        "resSciControls": {
          "$ref":
"http://example.org/icm/Intelligence_Community_Metadata.json#/$defs/ResSciControlsCodeBasedText"
        },
        "resSpAccReqProgIdent": {
          "$ref":
"http://example.org/icm/Intelligence_Community_Metadata.json#/$defs/Token"
        },
        "resAtomicEnergyMarkings": {
          "$ref":
"http://example.org/icm/Intelligence_Community_Metadata.json#/$defs/ResAtomicEnergyMarkingsCodeBasedText"
        },
        "resDissemControls": {
          "$ref":
"http://example.org/icm/Intelligence_Community_Metadata.json#/$defs/ResDissemControlsCodeBasedText"
        },
        "resDisplayOnlyTo": {
          "type": "string"
        },
        "resFgnGovInfoOpenSource": {
          "$ref":
"http://example.org/icm/Intelligence_Community_Metadata.json#/$defs/ResFgnGovInfoOpenSourceCodeBasedText"
        },
        "resFgnGovInfoProtSource": {
          "$ref":
"http://example.org/icm/Intelligence_Community_Metadata.json#/$defs/ResFgnGovInfoProtSourceCodeBasedText"
        },
        "resReleasableTo": {
          "$ref":
"http://example.org/icm/Intelligence_Community_Metadata.json#/$defs/ResReleasableToCodeBasedText"
        },
        "resNonIntelComMarkings": {
          "$ref":
```

```
"http://example.org/icm/Intelligence_Community_Metadata.json#/$defs/ResNonIntelComMark
ingsCodeBasedText"
        },
        "resClassifiedBy": {
          "type": "string"
        },
        "resCompilationReason": {
          "type": "string"
        },
        "resDerivClassifiedBy": {
          "type": "string"
        },
        "resClassificationReason": {
          "type": "string"
        },
        "resNonUSControls": {
          "$ref":
"http://example.org/icm/Intelligence_Community_Metadata.json#/$defs/ResNonUSControlsCo
deBasedText"
        },
        "resDerivedFrom": {
          "type": "string"
        },
        "resDeclassDate": {
          "$ref":
"http://example.org/icm/Intelligence_Community_Metadata.json#/$defs/ResDeclassDateStru
cText"
        },
        "resDeclassEvent": {
          "type": "string"
        },
        "resDeclassExemption": {
          "$ref":
"http://example.org/icm/Intelligence_Community_Metadata.json#/$defs/ResDeclassExemptio
nCodeBasedText"
        },
        "resHasApproximateMarkings": {
          "type": "boolean"
        }
      },
      "required": [
        "resClassification",
        "resOwnerProducer",
        "type"
      ]
    },
    ...
  }
}
```

B.4. NAS SWE Common Implementation Schema

The ShapeChange configuration from Listing 96 derives a JSON Schema from the SWE Common implementation schema defined by the NAS. That schema is a subset of the SWE Common 2.0 schema, which uses different type and property names, and has an inheritance relationship to ISO 19103 Record.

Listing 97 shows the resulting JSON Schema.

Listing 96. ShapeChange configuration for deriving a JSON Schema for the NAS SWE Common implementation schema

```
<?xml version="1.0" encoding="UTF-8"?>
<ShapeChangeConfiguration xmlns="http://www.interactive-instruments.de/ShapeChange/Configuration/1.1" xmlns:sc="http://www.interactive-instruments.de/ShapeChange/Configuration/1.1"
xmlns:xi="http://www.w3.org/2001/XInclude"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.interactive-instruments.de/ShapeChange/Configuration/1.1
file:/C:/NAS/UGAS20/resources/schema/ShapeChangeConfiguration.xsd">
 <input>
  <parameter name="inputModelType" value="SCXML"/>
  <parameter name="inputFile" value="C:/NAS/UGAS20/full_NAS/X-3/NAS_X-3_SCXML.xml"/>
  <parameter name="appSchemaNameRegex" value="SML - SWE Common"/>
  <parameter name="publicOnly" value="true"/>
  <parameter name="checkingConstraints" value="enabled"/>
  <parameter name="sortedSchemaOutput" value="true"/>
  <xi:include href="C:/NAS/UGAS20/GCSR/StandardAliases-GCSR.xml"/>
  <packages>
   <PackageInfo packageName="SML - SWE Common" ns="http://example.com/nas/sweCommon" nsabr="swe" xsdDocument="nasSweCommon.xsd" version="1.0"/>
  </packages>
 </input>
 <log>
  <parameter name="reportLevel" value="INFO"/>
  <parameter name="logFile" value="C:/NAS/UGAS20/SWE_Common_2.0/results/nas/log.xml"/>
 </log>
 <targets>
  <Target class="de.interactive_instruments.ShapeChange.Target.JSON.JsonSchemaTarget" mode="enabled">
   <targetParameter name="outputDirectory" value="C:/NAS/UGAS20/SWE_Common_2.0/results/nas/json_schema"/>
   <targetParameter name="sortedOutput" value="true"/>
   <targetParameter name="writeMapEntries" value="true"/>
   <targetParameter name="jsonSchemaVersion" value="2019-09"/>
   <targetParameter name="jsonBaseUri" value="http://example.org"/>
```

```
    <targetParameter name="entityTypeName" value="type"/>
    <targetParameter name="inlineOrByReferenceDefault" value="inlineOrByReference"/>
    <targetParameter name="defaultEncodingRule" value=
  "nasSweCommon20JsonSchemaRule"/>
    <rules>
     <EncodingRule name="nasSweCommon20JsonSchemaRule">
      <rule name="rule-json-prop-initialValueAsDefault"/>
      <rule name="rule-json-cls-union-typeDiscriminator"/>
      <rule name="rule-json-cls-name-as-entityType"/>
     </EncodingRule>
    </rules>
    <xi:include href="C:\NAS\UGAS20\GCSR\StandardMapEntries_JSON.xml"/>
    <mapEntries>
     <MapEntry type="TimeIndeterminateValueType" rule="nasSweCommon20JsonSchemaRule"
  targetType="string" param="keywords{const=now}"/>
    </mapEntries>
   </Target>
  </targets>
</ShapeChangeConfiguration>
```

Listing 97. JSON Schema derived from the NAS SWE Common implementation schema

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://example.org/swe/SML_-_SWE_Common.json",
  "$defs": {
    "SensorWebBoolean": {
      "allOf": [
        {
          "$ref": "http://example.org/swe/SML_-
  _SWE_Common.json#/$defs/SensorWebSimpleComponent"
        },
        {
          "type": "object",
          "properties": {
            "sweBooleanValue": {
              "type": "boolean"
            }
          },
          "required": [
            "sweBooleanValue"
          ]
        }
      ]
    },
    "SensorWebCategory": {
      "allOf": [
        {
          "$ref": "http://example.org/swe/SML_-
  _SWE_Common.json#/$defs/SensorWebSimpleComponent"
        },
```

```
28          {
29            "type": "object",
30            "properties": {
31              "sweCategoryCodeSpace": {
32                "type": "string",
33                "format": "uri"
34              },
35              "sweCategoryValue": {
36                "type": "string"
37              }
38            },
39            "required": [
40              "sweCategoryValue"
41            ]
42          }
43        ]
44      },
45      "SensorWebCategoryRange": {
46        "allOf": [
47          {
48            "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebSimpleComponent"
49          },
50          {
51            "type": "object",
52            "properties": {
53              "sweCategoryRangeCodeSpace": {
54                "type": "string",
55                "format": "uri"
56              },
57              "sweCategoryRangeValue": {
58                "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebTokenPair"
59              }
60            },
61            "required": [
62              "sweCategoryRangeValue"
63            ]
64          }
65        ]
66      },
67      "SensorWebCount": {
68        "allOf": [
69          {
70            "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebSimpleComponent"
71          },
72          {
73            "type": "object",
74            "properties": {
75              "sweCountValue": {
```

```
            "type": "integer"
          }
        },
        "required": [
          "sweCountValue"
        ]
      }
    ]
  },
  "SensorWebCountRange": {
    "allOf": [
      {
        "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebSimpleComponent"
      },
      {
        "type": "object",
        "properties": {
          "sweCountRangeValue": {
            "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebIntegerPair"
          }
        },
        "required": [
          "sweCountRangeValue"
        ]
      }
    ]
  },
  "SensorWebDataComponent": {
    "allOf": [
      {
        "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebIdentifiable"
      },
      {
        "type": "object",
        "properties": {
          "sweDataComponentDefinition": {
            "type": "string",
            "format": "uri"
          }
        }
      }
    ]
  },
  "SensorWebDataRecord": {
    "allOf": [
      {
        "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebDataComponent"
```

```
123         },
124         {
125           "type": "object",
126           "properties": {
127             "sweField": {
128               "type": "array",
129               "minItems": 1,
130               "items": {
131                 "$ref": "http://example.org/swe/SML_-
    _SWE_Common.json#/$defs/SensorWebDataComponent"
132               },
133               "uniqueItems": true
134             }
135           },
136           "required": [
137             "sweField"
138           ]
139         }
140       ]
141     },
142     "SensorWebIdentifiable": {
143       "allOf": [
144         {
145           "$ref": "http://www.opengis.net/to/be/determined/Record.json"
146         },
147         {
148           "type": "object",
149           "properties": {
150             "type": {
151               "type": "string"
152             },
153             "sweIdentifier": {
154               "type": "string",
155               "format": "uri"
156             },
157             "sweLabel": {
158               "type": "string"
159             },
160             "sweDescription": {
161               "type": "string"
162             }
163           },
164           "required": [
165             "type"
166           ]
167         }
168       ]
169     },
170     "SensorWebIntegerPair": {
171       "type": "object",
172       "properties": {
```

```
        "type": {
          "type": "string"
        },
        "sweIntegerPairItem": {
          "type": "array",
          "minItems": 2,
          "maxItems": 2,
          "items": {
            "type": "integer"
          },
          "uniqueItems": true
        }
      },
      "required": [
        "sweIntegerPairItem",
        "type"
      ]
    },
    "SensorWebQuality": {
      "oneOf": [
        {
          "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebQuantity"
        },
        {
          "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebQuantityRange"
        },
        {
          "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebCategory"
        },
        {
          "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebText"
        }
      ]
    },
    "SensorWebQuantity": {
      "allOf": [
        {
          "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebSimpleComponent"
        },
        {
          "type": "object",
          "properties": {
            "sweQuantityUom": {
              "type": "string",
              "format": "uri"
            },
```

```
            "sweQuantityValue": {
              "type": "number"
            }
          },
          "required": [
            "sweQuantityUom",
            "sweQuantityValue"
          ]
        }
      ]
    },
    "SensorWebQuantityRange": {
      "allOf": [
        {
          "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebSimpleComponent"
        },
        {
          "type": "object",
          "properties": {
            "sweQuantityRangeUom": {
              "type": "string",
              "format": "uri"
            },
            "sweQuantityRangeValue": {
              "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebRealPair"
            }
          },
          "required": [
            "sweQuantityRangeUom",
            "sweQuantityRangeValue"
          ]
        }
      ]
    },
    "SensorWebRealPair": {
      "type": "object",
      "properties": {
        "type": {
          "type": "string"
        },
        "sweRealPairItem": {
          "type": "array",
          "minItems": 2,
          "maxItems": 2,
          "items": {
            "type": "number"
          },
          "uniqueItems": true
        }
```

```
      },
      "required": [
        "sweRealPairItem",
        "type"
      ]
    },
    "SensorWebSimpleComponent": {
      "allOf": [
        {
          "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebDataComponent"
        },
        {
          "type": "object",
          "properties": {
            "sweDataComponentQuality": {
              "type": "array",
              "items": {
                "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebQuality"
              },
              "uniqueItems": true
            }
          }
        }
      ]
    },
    "SensorWebText": {
      "allOf": [
        {
          "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebSimpleComponent"
        },
        {
          "type": "object",
          "properties": {
            "sweTextValue": {
              "type": "string"
            }
          },
          "required": [
            "sweTextValue"
          ]
        }
      ]
    },
    "SensorWebTime": {
      "allOf": [
        {
          "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebSimpleComponent"
```

```
        },
        {
          "type": "object",
          "properties": {
            "sweReferenceDateTime": {
              "type": "string",
              "format": "date-time"
            },
            "sweTemporalUom": {
              "type": "string",
              "format": "uri"
            },
            "sweTemporalValue": {
              "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebTimePosition"
            }
          },
          "required": [
            "sweTemporalUom",
            "sweTemporalValue"
          ]
        }
      ]
    },
    "SensorWebTimePair": {
      "type": "object",
      "properties": {
        "type": {
          "type": "string"
        },
        "sweTimePairItem": {
          "type": "array",
          "minItems": 2,
          "maxItems": 2,
          "items": {
            "$ref": "http://example.org/swe/SML_-
_SWE_Common.json#/$defs/SensorWebTimePosition"
          },
          "uniqueItems": true
        }
      },
      "required": [
        "sweTimePairItem",
        "type"
      ]
    },
    "SensorWebTimePosition": {
      "oneOf": [
        {
          "type": "number"
        },
```

```
364         {
365           "type": "string",
366           "format": "date"
367         },
368         {
369           "type": "string",
370           "format": "time"
371         },
372         {
373           "type": "string",
374           "format": "date-time"
375         },
376         {
377           "type": "string",
378           "const": "now"
379         }
380       ]
381     },
382     "SensorWebTimeRange": {
383       "allOf": [
384         {
385           "$ref": "http://example.org/swe/SML_-
    _SWE_Common.json#/$defs/SensorWebSimpleComponent"
386         },
387         {
388           "type": "object",
389           "properties": {
390             "sweReferenceDateTime": {
391               "type": "string",
392               "format": "date-time"
393             },
394             "sweTemporalUom": {
395               "type": "string",
396               "format": "uri"
397             },
398             "sweTemporalRangeValue": {
399               "$ref": "http://example.org/swe/SML_-
    _SWE_Common.json#/$defs/SensorWebTimePair"
400             }
401           },
402           "required": [
403             "sweTemporalRangeValue",
404             "sweTemporalUom"
405           ]
406         }
407       ]
408     },
409     "SensorWebTokenPair": {
410       "type": "object",
411       "properties": {
412         "type": {
```

```
          "type": "string"
        },
        "sweTokenPairItem": {
          "type": "array",
          "minItems": 2,
          "maxItems": 2,
          "items": {
            "type": "string"
          },
          "uniqueItems": true
        }
      },
      "required": [
        "sweTokenPairItem",
        "type"
      ]
    }
  }
}
```

B.5. SWE Common 2.0 JSON Schema

The ShapeChange configuration from Listing 98 derives a JSON Schema from the SWE Common 2.0 conceptual schema, which is close to the JSON encoding described by the OGC Best Practices document JSON Encoding Rules SWE Common / SensorML.

Listing 98. ShapeChange configuration for deriving a JSON Schema for SWE Common 2.0

```
<?xml version="1.0" encoding="UTF-8"?>
<ShapeChangeConfiguration xmlns="http://www.interactive-
instruments.de/ShapeChange/Configuration/1.1" xmlns:sc="http://www.interactive-
instruments.de/ShapeChange/Configuration/1.1"
xmlns:xi="http://www.w3.org/2001/XInclude"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.interactive-
instruments.de/ShapeChange/Configuration/1.1
https://shapechange.net/resources/schema/ShapeChangeConfiguration.xsd">
 <input>
  <parameter name="inputModelType" value="EA7"/>
  <parameter name="inputFile" value="sweCommon.eap"/>
  <parameter name="appSchemaNamespaceRegex"
value="^http://www\.opengis\.net/swe/2\.0"/>
  <parameter name="publicOnly" value="true"/>
  <parameter name="checkingConstraints" value="enabled"/>
  <parameter name="sortedSchemaOutput" value="true"/>
  <xi:include href="https://shapechange.net/resources/config/StandardAliases.xml"/>
 </input>
 <log>
  <parameter name="reportLevel" value="INFO"/>
  <parameter name="logFile" value="results/log.xml"/>
```

```
15 </log>
16 <transformers>
17  <Transformer
```

```
class="de.interactive_instruments.ShapeChange.Transformation.Adding.AttributeCreator"
id="trf">
18    <advancedProcessConfigurations>
19     <AttributeDefinition>
20      <classSelection>
21       <PackageSelector schemaNameRegex="SWE Common Data Model 2.0"/>
22       <ClassSelector nameRegex="AbstractDataComponent"/>
23      </classSelection>
24      <name>name</name>
25      <multiplicity>0..1</multiplicity>
26      <taggedValues>
27       <TaggedValue name="sequenceNumber" value="0"/>
28      </taggedValues>
29      <type>CharacterString</type>
30     </AttributeDefinition>
31     <AttributeDefinition>
32      <classSelection>
33       <PackageSelector schemaNameRegex="SWE Common Data Model 2.0"/>
34       <ClassSelector nameRegex="AbstractSWE"/>
35      </classSelection>
36      <name>id</name>
37      <multiplicity>0..1</multiplicity>
38      <taggedValues>
39       <TaggedValue name="sequenceNumber" value="0"/>
40      </taggedValues>
41      <type>CharacterString</type>
42     </AttributeDefinition>
43    </advancedProcessConfigurations>
44   </Transformer>
45  </transformers>
46  <targets>
47   <Target
class="de.interactive_instruments.ShapeChange.Target.JSON.JsonSchemaTarget"
mode="enabled" inputs="trf">
48    <targetParameter name="outputDirectory" value="results/json_schema"/>
49    <targetParameter name="sortedOutput" value="true"/>
50    <targetParameter name="writeMapEntries" value="true"/>
51    <targetParameter name="jsonSchemaVersion" value="draft-07"/>
52    <targetParameter name="entityTypeName" value="type"/>
53    <!-- NOTE: for a 2019-09 schema, the value would be "#/$defs/Reference" -->
54    <targetParameter name="byReferenceJsonSchemaDefinition"
value="#/definitions/Reference"/>
55    <targetParameter name="inlineOrByReferenceDefault" value="inlineOrByReference"/>
56    <targetParameter name="defaultEncodingRule" value="sweCommon20JsonSchemaRule"/>
57    <rules>
58     <EncodingRule name="sweCommon20JsonSchemaRule">
59      <rule name="rule-json-prop-initialValueAsDefault"/>
60      <rule name="rule-json-cls-union-typeDiscriminator"/>
61      <rule name="rule-json-cls-name-as-entityType"/>
62     </EncodingRule>
```

```
    </rules>
    <mapEntries>
     <MapEntry type="CharacterString" rule="*" targetType="string" param=""/>
     <!-- Mapping Boolean (not the one defined by SWE Common itself) to 'string',
    following the Best Practice; should use 'boolean'. Future work item for SWE Common
    SWG. -->
     <MapEntry type="Boolean" rule="*" targetType="string"
    param="ignoreForTypeFromSchemaSelectedForProcessing"/>
     <MapEntry type="Integer" rule="*" targetType="integer" param=""/>
     <MapEntry type="Real" rule="*" targetType="number" param=""/>
     <MapEntry type="DateTime" rule="*" targetType="string"
    param="formatted{format=date-time}"/>
     <MapEntry type="ScopedName" rule="*" targetType="string"
    param="formatted{format=uri}"/>
     <MapEntry type="SC_CRS" rule="*" targetType="string"
    param="formatted{format=uri}"/>
     <MapEntry type="CI_Party" rule="*" targetType="string" param=""/>
     <MapEntry type="TM_Position" rule="*" targetType="#/definitions/TimePosition"
    param=""/>
     <MapEntry type="TimePair" rule="*" targetType="string" param=""/>
     <MapEntry type="RealPair" rule="*" targetType="string" param=""/>
     <MapEntry type="TokenPair" rule="*" targetType="string" param=""/>
     <MapEntry type="IntegerPair" rule="*" targetType="string" param=""/>
     <MapEntry type="Dictionary" rule="*" targetType="#/definitions/Reference"
    param=""/>
     <MapEntry type="NilValue" rule="*" targetType="string" param=""/>
     <MapEntry type="UnitOfMeasure" rule="*"
    targetType="#/definitions/UnitReference" param=""/>
     <MapEntry type="TM_TemporalCRS" rule="*" targetType="string"
    param="formatted{format=uri}"/>
     <MapEntry type="UomTime" rule="*" targetType="#/definitions/UnitReference"
    param=""/>
     <!-- No target type for Any and EncodedValues, so that anything can be encoded
    for properties with such a type as value type. -->
     <MapEntry type="Any" rule="*" targetType="" param=""/>
     <MapEntry type="EncodedValues" rule="*" targetType="" param=""/>
    </mapEntries>
   </Target>
  </targets>
</ShapeChangeConfiguration>
```

Explanations are required for a number of items in the configuration.

- The ShapeChange workflow contains a model transformation which adds an optional "name" attribute to the SWE Common *AbstractDataComponent* type. This is necessary to support soft-typing as used in SWE Common encodings. The attribute is optional to allow an instantiable subtype of *AbstractDataComponent* to be encoded without the "name".
- The Best Practices document defines the JSON encoding as a mapping from and to the SWE Common XML encoding. Mapping of XML attributes is described in generic terms in section 5.3.5 of the Best Practices document. The example in section 5.4.2 of that document shows the

JSON encoding of an "id" XML attribute. The workflow therefore also adds an optional "id" attribute, to support the encoding of "id" XML attributes that can be present in XML encoded SWE Common data.

- The JSON Schema target parameter *byReferenceJsonSchemaDefinition* as well as the target type of some map entries (e.g. for type *UnitOfMeasure*) contain JSON pointers which reference definitions from the resulting JSON Schema for SWE Common. However, the referenced definitions are not generated by ShapeChange because the conceptual schema of SWE Common - at least the one available for testing in UGAS-2020 - does not define these types. The JSON Schema definitions therefore need to be defined manually, which is similar to the XML Schema of SWE Common, where *basic_types.xsd* contains handcrafted XML Schema definitions. The additional JSON Schema definitions are shown in Listing 99; they need to be added manually to the JSON Schema file produced by ShapeChange.
- The JSON Schema target parameter *inlineOrByReferenceDefault* has been set to "inlineOrByReference", in order to support what is possible in the XML encoding of SWE Common, for example to encode the "constraint" of a *Category* inline or by reference. Note that the Best Practices document explains by-reference encoding in section 5.2.3, requirement 9.
- No map entry is defined for type *Any*. That results in the "extension" property of the JSON Schema definition for *AbstractSWE*.extension to have no type restriction (other than the value being a JSON array). A similar approach is taken for type *EncodedValues* (which is the value type of *DataStream*.values).
- Map entries for other types, such as *SC_CRS*, have a target type that reflects the implementation of that type in the SWE Common XML encoding.
- The configuration also highlights a number of issues:
  - According to the Best Practices document, section 5.3.6, type *Boolean* is mapped to a JSON string. More specifically, anything that is not a decimal, float, double, or any type derived from these shall be encoded as JSON string - thus also a boolean value. Since JSON Schema supports type *boolean*, mapping the conceptual type *Boolean* to this JSON schema type would be more appropriate. In addition, the SWE Common schema itself defines a type called *Boolean*. This creates a conflict, since the map entry for *Boolean* should not apply to the SWE Common type *Boolean*. A flag therefore had to be added to the map entry (using a map entry parameter), to indicate that the map entry shall not be applied to a type if that type is owned by one of the schemas selected for processing (by the ShapeChange JSON Schema target).
  - A *NilValue* cannot fully be represented in JSON. In the SWE Common XML encoding, *NilValue* is a simple type (string), with a "reason" attribute; a *NilValue* is not represented by an object element. However, according to the Best Practices document, section 5.3.5, XML attributes - such as the "reason" attribute - can only be represented for types that are represented by an object element in XML. The case of a simple type with additional attributes does not appear to be covered by the encoding rules defined in the Best Practices document. *NilValue* therefore is mapped to JSON string.
  - A *TimePair* can only be represented as a JSON string. In the SWE Common XML encoding, *TimePair* is a list of length 2, with each value having a simple type that represents a TM_Position. This cannot be defined in JSON Schema, at least not in the way required by the encoding rules defined in the Best Practices document. A revision of these encoding rules should consider using an array of length 2 to represent pairs, rather than using the string

representation found in the XML encoding. The situation is similar for other pair types defined by SWE Common 2.0. For the time being, though, mapping pair types to JSON string should support the JSON encoding as defined in the Best Practices document.

- Type discriminator unions may need to be encoded using the "anyOf" keyword instead of "oneOf" - or even using an if-then-else construct that applies JSON Schema definitions based upon the "type" of a given JSON encoded SWE Common object. The reason is that the content model of JSON Schema definitions of SWE Common types is not distinct enough to avoid cases where a JSON object matches multiple JSON Schema definitions contained in the "oneOf". For example, the SWE Common *Quality* union contains the types *Category* and *Text*, which have almost identical content model, the only difference being the optional "codeSpace" in *Category*.
- SWE Common uses the abstract type *AbstractDataComponent* in a number of important places, especially as value type of properties in *DataRecord*, *DataArray*, and *DataStream*. As described in Class Specialization and Property Ranges, JSON Schema does not directly support class specialization. The generated JSON Schemas thus only check that a value of such properties is valid against the JSON Schema definition of *AbstractDataComponent*, but does not validate the value against the JSON Schema definition of the value type. Implementing JSON Schema checks as described in the Note of Class Specialization and Property Ranges could be a solution, particularly since JSON encoded SWE Common data component types have a member to encode the type name, and because SWE Common types are typically not extended.

Listing 99. Additional JSON Schema definitions for SWE Common 2.0, which must be added manually

```
{
 "$schema": "http://json-schema.org/draft-07/schema#",
 "definitions": {
  "Reference": {
   "type": "object",
   "properties": {
    "href": {
     "type": "string",
     "format": "uri"
    },
    "role": {"type": "string"},
    "arcrole": {"type": "string"}
   },
   "required": ["href"]
  },
  "TimePosition": {
   "type": "object",
   "anyOf": [
    {"type": "number"},
    {
     "type": "string",
     "format": "date"
    },
    {
     "type": "string",
```

```
      "format": "time"
     },
     {
      "type": "string",
      "format": "dateTime"
     },
     {
      "type": "string",
      "const": "now"
     }
    ],
    "required": ["type"]
   },
   "UnitReference": {
    "oneOf": [
     {"$ref": "#/definitions/Reference"},
     {
      "type": "object",
      "properties": {
       "code": {"type": "string"},
       "required": ["code"]
      }
     }
    ]
   }
  }
}
```

Annex C: Examples of SPARQL-based SHACL Functions

The following listing shows how SPARQL-based functions can be defined to support arithmetics in SHACL node expressions (one of the advanced features of SHACL that are in development).

```
# baseURI: http://example.org/shacl/extensions
# imports: http://datashapes.org/dash
# prefix: shaclex

@prefix shaclex: <http://example.org/shacl/extensions#> .
@prefix dash: <http://datashapes.org/dash#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example.org/shacl/extensions>
  a owl:Ontology ;
  owl:imports <http://datashapes.org/dash> ;
  sh:declare [
        sh:prefix "rdf" ;
        sh:namespace "http://www.w3.org/1999/02/22-rdf-syntax-ns#"^^xsd:anyURI ;
    ] ;
    sh:declare [
        sh:prefix "rdfs" ;
        sh:namespace "http://www.w3.org/2000/01/rdf-schema#"^^xsd:anyURI ;
    ]
.

####################
# multiply
####################
shaclex:multiply
    a sh:SPARQLFunction ;
    rdfs:comment "Implements the SPARQL 1.1 '*' operator." ;
    sh:parameter [
        sh:path shaclex:op1 ;
        sh:description "The first operand" ;
    ] ;
    sh:parameter [
        sh:path shaclex:op2 ;
        sh:description "The second operand" ;
    ] ;
    sh:select """
        SELECT ($op1 * $op2 AS ?result)
        WHERE {
        }
```

```
        """ .

#####################
# divide
#####################
shaclex:divide
    a sh:SPARQLFunction ;
    rdfs:comment "Implements the SPARQL 1.1 '/' operator." ;
    sh:parameter [
        sh:path shaclex:op1 ;
        sh:description "The first operand" ;
    ] ;
    sh:parameter [
        sh:path shaclex:op2 ;
        sh:description "The second operand" ;
    ] ;
    sh:select """
        SELECT ($op1 / $op2 AS ?result)
        WHERE {
        }
        """ .

#####################
# add
#####################
shaclex:add
    a sh:SPARQLFunction ;
    rdfs:comment "Implements the SPARQL 1.1 '+' operator." ;
    sh:parameter [
        sh:path shaclex:op1 ;
        sh:description "The first operand" ;
    ] ;
    sh:parameter [
        sh:path shaclex:op2 ;
        sh:description "The second operand" ;
    ] ;
    sh:select """
        SELECT ($op1 + $op2 AS ?result)
        WHERE {
        }
        """ .

#####################
# subtract
#####################
shaclex:subtract
    a sh:SPARQLFunction ;
    rdfs:comment "Implements the SPARQL 1.1 '-' operator." ;
    sh:parameter [
        sh:path shaclex:op1 ;
        sh:description "The first operand" ;
```

```
    ] ;
    sh:parameter [
        sh:path shaclex:op2 ;
        sh:description "The second operand" ;
    ] ;
    sh:select """
        SELECT ($op1 - $op2 AS ?result)
        WHERE {
        }
        """ .

#####################
# equals
#####################
shaclex:equals
    a sh:SPARQLFunction ;
    rdfs:comment "Implements the SPARQL 1.1 '=' operator." ;
    sh:parameter [
        sh:path shaclex:op1 ;
        sh:description "The first operand" ;
    ] ;
    sh:parameter [
        sh:path shaclex:op2 ;
        sh:description "The second operand" ;
    ] ;
    sh:returnType xsd:boolean ;
    sh:select """
        SELECT ($op1 = $op2 AS ?result)
        WHERE {
        }
        """ .

#####################
# notEquals
#####################
shaclex:notEquals
    a sh:SPARQLFunction ;
    rdfs:comment "Implements the SPARQL 1.1 '!=' operator." ;
    sh:parameter [
        sh:path shaclex:op1 ;
        sh:description "The first operand" ;
    ] ;
    sh:parameter [
        sh:path shaclex:op2 ;
        sh:description "The second operand" ;
    ] ;
    sh:returnType xsd:boolean ;
    sh:select """
        SELECT ($op1 != $op2 AS ?result)
        WHERE {
        }
```

```
        """ .

####################
# lessThan
####################
shaclex:lessThan
    a sh:SPARQLFunction ;
    rdfs:comment "Implements the SPARQL 1.1 '<' operator." ;
    sh:parameter [
        sh:path shaclex:op1 ;
        sh:description "The first operand" ;
    ] ;
    sh:parameter [
        sh:path shaclex:op2 ;
        sh:description "The second operand" ;
    ] ;
    sh:returnType xsd:boolean ;
    sh:select """
        SELECT ($op1 < $op2 AS ?result)
        WHERE {
        }
        """ .

####################
# lessThanOrEquals
####################
shaclex:lessThanOrEquals
    a sh:SPARQLFunction ;
    rdfs:comment "Implements the SPARQL 1.1 '<=' operator." ;
    sh:parameter [
        sh:path shaclex:op1 ;
        sh:description "The first operand" ;
    ] ;
    sh:parameter [
        sh:path shaclex:op2 ;
        sh:description "The second operand" ;
    ] ;
    sh:returnType xsd:boolean ;
    sh:select """
        SELECT ($op1 <= $op2 AS ?result)
        WHERE {
        }
        """ .

####################
# greaterThan
####################
shaclex:greaterThan
    a sh:SPARQLFunction ;
    rdfs:comment "Implements the SPARQL 1.1 '>' operator." ;
    sh:parameter [
```

```
        sh:path shaclex:op1 ;
        sh:description "The first operand" ;
    ] ;
    sh:parameter [
        sh:path shaclex:op2 ;
        sh:description "The second operand" ;
    ] ;
    sh:returnType xsd:boolean ;
    sh:select """
        SELECT ($op1 > $op2 AS ?result)
        WHERE {
        }
        """ .

#####################
# greaterThanOrEquals
#####################
shaclex:greaterThanOrEquals
    a sh:SPARQLFunction ;
    rdfs:comment "Implements the SPARQL 1.1 '>=' operator." ;
    sh:parameter [
        sh:path shaclex:op1 ;
        sh:description "The first operand" ;
    ] ;
    sh:parameter [
        sh:path shaclex:op2 ;
        sh:description "The second operand" ;
    ] ;
    sh:returnType xsd:boolean ;
    sh:select """
        SELECT ($op1 >= $op2 AS ?result)
        WHERE {
        }
        """ .
```

Annex D: Revision History

Table 14. Revision History

Date	Editor	Release	Primary clauses modified	Descriptions
Mar 12, 2020	Johannes Echterhoff, Clemens Portele	0.1	all	initial commit
Apr 3, 2020	Johannes Echterhoff, Clemens Portele		6,9	draft of the JSON Schema and OpenAPI chapters
Apr 14, 2020	Johannes Echterhoff, Clemens Portele		6,9	changes after review by Paul Birkel
Apr 21, 2020	Johannes Echterhoff, Clemens Portele		6,9	more updates after feedback and implementation
May 6, 2020	Johannes Echterhoff		7	initial version of Core Profile
May 12, 2020	Johannes Echterhoff		B	JSON schemas for external schemas
Jul 31, 2020	Johannes Echterhoff, Clemens Portele		4,5,6,7,8,C	initial version of the SHACL chapter, Core Profile updates
Aug 13, 2020	Johannes Echterhoff, Clemens Portele		6,7,B	JSON schemas for the NAS and external schemas, Core Profile updates
Sep 3, 2020	Johannes Echterhoff, Clemens Portele		2,5,7	Update summary and Core Profile
Oct 12, 2020	Johannes Echterhoff, Clemens Portele		2,4,5,6,7	Updates after review, complete list of acronyms, extensions to the Core Profile
Oct 29, 2020	Clemens Portele		6	Update Core Profile based on discussions in Oct 15 call

Annex E: Bibliography

- Internet Engineering Task Force (IETF). Draft draft-handrews-json-schema-02: **JSON Schema: A Media Type for Describing JSON Documents** [online]. Edited by A. Wright, H. Andrews, B. Hutton, and G. Dennis. 2019 [viewed 2020-04-20]. Available at https://tools.ietf.org/html/draft-handrews-json-schema-02
- Internet Engineering Task Force (IETF). Draft draft-handrews-json-schema-validation-02: **JSON Schema Validation: A Vocabulary for Structural Validation of JSON** [online]. Edited by A. Wright, H. Andrews, and B. Hutton. 2019 [viewed 2020-04-20]. Available at https://tools.ietf.org/html/draft-handrews-json-schema-validation-02
- OGC Best Practices: JSON Encoding Rules SWE Common / SensorML, available online at http://docs.opengeospatial.org/bp/17-011r2/17-011r2.html
- OGC Testbed-14: Application Schema-based Ontology Development Engineering Report (OGC 18-032r2), available online at http://docs.opengeospatial.org/per/18-032r2.html
- OGC Testbed-14: Application Schemas and JSON Technologies Engineering Report (OGC 18-091r2), available online at http://docs.opengeospatial.org/per/18-091r2.html
- OGC Testbed-12 ShapeChange Engineering Report (OGC 16-020), available online at http://docs.opengeospatial.org/per/16-020.html
- OWS-9 System Security Interoperability (SSI) UML-to-GML-Application-Schema (UGAS) Conversion Engineering Report, available online at https://portal.opengeospatial.org/files/?artifact_id=51784