DRAFT 62361: POWER SYSTEMS MANAGEMENT AND ASSOCIATED INFORMATION EXCHANGE – INTEROPERABILITY IN THE LONG TERM –

Part 104: CIM Profiles to JSON schema Mapping

Rev 01v20 – 2021-07-06

CONTENTS

FOREWORD 5
INTRODUCTION 7
1 Scope 8
2 Normative references 8
3 Terms and definitions 9
3.1 Common Information Model 9
3.2 Contextual model 9
3.3 JavaScript Object Notation 9
3.4 Profile 9
3.5 Resource Description Format 9
3.6 Unified Modeling Language 10
3.7 Uniform Resource Indicator 10
3.8 JSON schema 10
3.9 JSON Subschema or simply Subschema 10
3.10 JSON Pointers 10
3.11 Web Ontology Language 10
4 System context 11
4.1 CIM 11
4.2 Contextual model 11
4.3 Contextual model artefacts 11
4.3.1 Contextual artefacts and CIM subset 11
4.3.2 Contextual artefacts 12
4.4 Mapping a contextual model to a JSON schema 15
4.4.1 General 15
4.4.2 Traceability 16
4.5 JSON schema Representation 16
4.6 Namespaces 16
5 Mapping specifications 16
5.1 General 16
5.2 Profile mapping 17
5.2.1 Preliminary Background 17
5.2.2 General 18
5.2.3 Profile $id, $schema, title, description, namespace, and type 19
5.2.4 Envelope element 20
5.2.5 Root elements 21
5.2.6 Semantic annotation 23
5.3 Structured classes 25
5.4 Compound classes 26
5.5 Basic types 27
5.6 Simple types 29
5.6.1 Mapping rules 29
5.6.2 Possible facets 30
5.7 DataTypes mapping 32
5.8 Enumeration classes mapping 39

5.9 CodeList classes 40
5.10 Simple properties mapping 40
5.11 Compound properties mapping 44
5.12 Object properties 45
5.12.1 Mapping rules overview 45
5.12.2 Typed object properties mapping 45
5.12.3 By-reference object properties mapping 47
5.12.4 Union object properties mapping 51
5.13 Exclusive property group mapping 60
5.14 Documentation 64
5.14.1 JSON schema documentation 64
5.14.2 General mapping 65
5.15 Names 65
5.16 Mapping order 66
5.16.1 General order 66
5.16.2 Mapping order when there is no superclass 66
5.16.3 Mapping order when there is a superclass 66
5.16.4 Mapping order examples 66
5.17 Changing name rules 67
Annex A (Informative) Introduction to JSON 68
Annex B (Informative) Contextual model representation 70
Annex C (Informative) XSD to JSON schema type and facet mappings 72
Annex D (Informative) Changing name rules examples 74
D.1 Changing name rule context 74
D.2 Changing name rule when CIM association end role name and CIM super class name are the same 74
D.3 Changing name rule when CIM association end rolename is the CIM super class name prefixed by a qualifier followed by an underscore 76
D.4 Changing name rule when CIM association end role name and the CIM super class name are completely different 78
Annex E (Informative) An overview of JSON schema's $id and $ref kewords 80
E.1 Preliminary Background 80
E.1.1 JSON Schema and JSON subschemas 80
E.1.2 Overview of URIs 80
E.1.3 The JSON schema $id keyword 81
E.1.4 The JSON schema $ref keyword and base URIs 83
E.2 Absolute-URI $refs, URI $refs resolution, and vendor implementations 84
E.2.1 Externalizing codelists to support local codelist extensions 84
E.2.2 An applied example of externalization of local codelist extensions 87
E.2.3 The JSON schema specification and vendor implementations 89
Bibliography 94

Figure 1 – Example JSON schema CIM-based profile 17
Figure 2 – Example CIMDatatype class 33
Figure 3 – CIM association from Class1 to a SuperClass declared as a "Union" 51
Figure D.1 – Example of end role name matching super class name 74
Figure D.2 – Contextual model end role name matching super class name 74
Figure D.3 – Example of end role name with a qualifier 76

Figure D.4 – Contextual model end role name with a qualifier 76
Figure D.5 – End role name and super class name different 78
Figure D.6 – Contextual model with end role name different from super class name 78
Figure E.1 – Diagram of the relationships between URIs, URNs, and URLs 81
Figure E.2 – Diagram of the relationships between URIs, URNs, and URLs 91

Table 1 – Contextual model artefacts 15
Table 2 – Basic Types 28
Table 3 – Facets 32
Table B.1 – Contextual model representation 71
Table C.1 – XSD to JSON types and facet mappings 73

INTERNATIONAL ELECTROTECHNICAL COMMISSION

____

POWER SYSTEMS MANAGEMENT AND ASSOCIATED INFORMATION EXCHANGE – INTEROPERABILITY IN THE LONG TERM –

Part 104: CIM Profiles to JSON schema Mapping

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
5) IEC itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.
6) All users should ensure that they have the latest edition of this publication.
7) No liability shall attach to IEC or its directors, employees, servants or agents including individual experts and members of its technical committees and IEC National Committees for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC Publication or any other IEC Publications.
8) Attention is drawn to the Normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.
9) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

IEC 62361-104 has been prepared by subcommittee WG19: Interoperability within TC57 in the long term, of IEC Technical Committee 57: Power systems management and associated information exchange.

The text of this International Standard is based on the following documents:

Draft	Report on voting
XX/XX/FDIS	XX/XX/RVC

Full information on the voting for the approval can be found in the report on voting indicated in the above table.

The language used for the development of this International Standard is English.

This document was drafted in accordance with ISO/IEC Directives, Part 2, and developed in accordance with ISO/IEC Directives, Part 1 and ISO/IEC Directives, IEC Supplement, available at www.iec.ch/members_experts/refdocs. The main document types developed by IEC are described in greater detail at http://www.iec.ch/standardsdev/publications.

The committee has decided that the contents of this document will remain unchanged until the stability date indicated on the IEC website under webstore.iec.ch in the data related to the specific document. At this date, the document will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

The National Committees are requested to note that for this document the stability date is 2023.

THIS TEXT IS INCLUDED FOR THE INFORMATION OF THE NATIONAL COMMITTEES AND WILL BE DELETED AT THE PUBLICATION STAGE.

INTRODUCTION

This standard is one of the IEC 62361 series which define standards that may be used by all Working Groups within TC57. These standards address areas of interest that impact multiple standards and provide consistency for implementations.

This part 104 describes a mapping from CIM profiles to IETF JSON schemas and defines the rules that CIM JSON message payloads must adhere to.

The principal objective of this part 104 is to facilitate the exchange of information in the form of JSON documents whose semantics are defined by the IEC CIM and whose syntax is defined by an IETF JSON schema. This will facilitate the integration of all applications that use the JSON schema message payloads developed by the WGs and implemented independently by different vendors into their systems.

The common information model (CIM) specifies the basis for the semantics for message payload exchanges defined by WG14 and WG16. The profile specifications, which are contained in other parts of the IEC 61968 and IEC 62325 standards, specify the content of the message payloads exchanged. The format/syntax of those payloads is specified in this part 104 document.

The International Electrotechnical Commission (IEC) draws attention to the fact that it is claimed that compliance with this document may involve the use of a patent. IEC takes no position concerning the evidence, validity, and scope of this patent right.

The holder of this patent right has assured IEC that s/he is willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with IEC. Information may be obtained from the patent database available at http://patents.iec.ch.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those in the patent database. IEC shall not be held responsible for identifying any or all such patent rights.

POWER SYSTEM MANAGEMENT AND ASSOCIATED INFORMATION EXCHANGE – INTEROPERABILITY IN THE LONG TERM –

Part 104: CIM Profiles to JSON schema Mapping

1 Scope

The JSON data interchange format was conceived in the early 2000s as a language-independent data format. Prior to that, XML was the primary choice for open data interchange. However, since its inception, transformations in the world of open data sharing have occurred leading to JSON's evolution as an increasingly popular and lighter weight alternative to XML.

This part 104 of IEC 62361 describes a mapping from CIM profiles to IETF JSON schemas.

The purpose of this mapping is to facilitate the exchange of information in the form of JSON documents whose semantics are defined by the IEC CIM and whose syntax is defined by an IETF JSON schema.

2 Normative references

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

IEC 61968-11, *System Interfaces for Distribution Management: Common Information Model Extensions for Distribution*

IEC 61970-301, *Energy Management System Application Program Interface (EMS-API): Common Information Model*

IEC 62325-301*, Framework for Energy Market Communications Common Information Model (CIM) Extensions for Markets*

Standard ECMA-404 The JSON Data Interchange Syntax (Second Edition) December 2017

ISO 8601 Data elements and interchange formats – Information interchange – Representation of dates and times 2004

IETF RFC 8259 The JavaScript Object Notation (JSON) Data Interchange Format December 2017

IETF RFC 6901 JavaScript Object Notation (JSON) Pointer April 2013

IETF RFC 3986 Uniform Resource Identifier (URI): Generic Syntax January 2005

IETF RFC 6596 The Canonical Link Relation April 2012

IETF JSON schema Specification Internet Draft (draft-07) March 2019

IETF JSON schema Specification Internet Draft (draft 2020-12) December 2020

Semantic Annotations for WSDL and XML Schema W3C Recommendation 28 August 2007

3 Terms and definitions

For the purposes of this document, the terms and definitions of IEC 61970-2 apply, as well as the following.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- IEC Electropedia: available at http://www.electropedia.org/
- ISO Online browsing platform: available at http://www.iso.org/obp

3.1 Common Information Model

CIM

abstract model that represents all the major objects in an electric utility enterprise typically needed to model the operational aspects of a utility. CIM is defined in a family of standards of IEC Technical Committee 57.

3.2 Contextual model

a restricted subset of CIM artefacts.

3.3 JavaScript Object Notation

JSON

an open-standard file format, JSON is a text format for the serialization of structured data. It is derived from the object literals of JavaScript, as defined in the ECMAScript Programming Language Standard, Third Edition [ECMA-262]. (See Annex A)

Two standards, RFC 8259 and ECMA-404, were defined in 2017. The ECMA standard describes only the allowed syntax, whereas the RFC covers some security and interoperability considerations.

IETC RFC 8259 makes normative reference to the ECMA-404 standard. There are no inconsistencies in the definition of the term “JSON text” across either specification, however, it should be noted that ECMA-404 allows several practices that IETC RFC 8259 recommends be avoided in the interests of maximum interoperability. This CIM profiles to JSON schema mapping specification aims for maximum interoperability and therefore defers to IETC RFC 8259 in those areas when applicable. When either document is changed, ECMA and the IETF work together to ensure that the two documents stay aligned through such changes.

The official Internet media type for JSON is `application/json`. JSON filenames use the extension `.json`.

3.4 Profile

A profile, as used in this document, is defined in IEC 62361-101[1]. Profile is a uniquely named subset of CIM classes, associations, and attributes needed to accomplish a specific type of interface.

3.5 Resource Description Format

RDF

———————

[1] Currently in development at the time of this writing.

family of World Wide Web Consortium (W3C) specifications originally designed as a metadata data model. It has come to be used as a general method for conceptual description or modeling of information that is implemented in web resources, using a variety of syntax formats.

3.6 Unified Modeling Language

UML

formal and comprehensive descriptive language with diagramming techniques used to represent software systems, from requirements analysis, through design and implementation, to documentation. UML is a standard defined by the Object Management Group (OMG). UML is used to describe CIM.

3.7 Uniform Resource Indicator

URI

string of characters used to identify a name or a resource. Such identification enables interaction with representations of the resource over a network (typically the World Wide Web) using specific protocols. Schemes specifying a concrete syntax and associated protocols define each URI.

3.8 JSON schema

family of Internet Engineering Task Force (IETF) specifications. JSON schema specifies a JSON-based format to define the structure of JSON data for validation, documentation, and interaction control. It provides a contract for the JSON data required by a given application, and how that data can be modified. It is based on some but not all of the concepts from XML Schema (XSD), but is JSON-based. As in XSD, the same serialization/deserialization tools can be used both for the schema and data; and is self-describing. This specification covers both the draft07 release as well as the December 2020 release, which is intended to serve as the final draft to be readied for the IETF RFC standardization process.

There is no official file extension for JSON schema, but an extension of `.schema.json` has been recommended. For the purposes of this specification, this file extension is to be adhered to for IEC profiles based on it so as to broadly promote conformity within IEC standards dependent upon it.

3.9 JSON Subschema or simply Subschema

in the context of this CIM profile to JSON schema mapping document, a subschema is any JSON object schema definition appearing within the `$defs` element of a 2020-12 JSON schema profile or within the `definitions` element of a draft-07 profile.

3.10 JSON Pointers

an IETF specification (RFC 6901), JSON Pointer defines a string syntax for identifying a specific value within a JavaScript Object Notation (JSON) document.

3.11 Web Ontology Language

OWL

family of knowledge representation languages for authoring ontologies. The languages are characterised by formal semantics and RDF/XML-based serializations for the Semantic Web. OWL is endorsed by the World Wide Web Consortium (W3C)

4 System context

4.1 CIM

CIM is an abstract model that represents all the major objects in an electric utility enterprise typically needed to model the operational aspects of a utility. This model includes public classes and attributes for these objects, as well as the relationships between them.

CIM is defined by IEC standards 61968-11, 61970-301 and 62325-301.

The CIM may be augmented with project or application-specific extensions. In that case, the references to the CIM in the foregoing can be read as CIM with extensions.

4.2 Contextual model[2]

The concept of a contextual model is borrowed from the UN/CEFACT modelling approach and may be used in CIM standards formation. The contextual model may be any one of several formats including OWL or a UML subset package. The specific methods to generate the contextual model will be provided in IEC 62361-101.[3]

No specific contextual modelling language is assumed by this specification. However, the artefacts defined in Table 1 are used in this document when referring to the contextual model and are assumed to be capable of expression in whichever language is used.

The mapping specifications (see clause 5) apply to these contextual model artefacts which could be represented in a number of languages. Refer to Annex B for two possible representations.

4.3 Contextual model artefacts

4.3.1 Contextual artefacts and CIM subset

In Table 1, most contextual artefacts are defined in relation to CIM artefacts. The exact relation between these two kinds of artefacts is being defined in 62361-101. Here, we use the term subset: a contextual artefact is a subset of some CIM artefact. Subset means that a contextual artefact could have the same characteristics of its CIM counterpart or a subset of these characteristics.

Examples:

- “IdentifiedObject” class in CIM has four attributes (“mRID”, “aliasName”, “name” and “description”) and one association “Names”, i.e. its characteristics. “IdentifiedObject” **structured class** in contextual model could have the same characteristics of its CIM counterpart or just some of them: so “IdentifiedObject” contextual artefact is defined as a subset of “IdentifiedObject” CIM artefact.
- “name” attribute of CIM “IdentifiedObject” has two characteristics : a cardinality that is optional and a type that is a string. In contextual model, “name” **simple property** of “IdentifiedObject” **structured class** could have the same characteristics of its CIM counterpart or some more restricted ones: example, cardinality of “name” could be restricted to mandatory and/or string length could be defined. So “name” contextual artefact is defined as a subset of “name” CIM artefact.
- “Names” association end role name of CIM “IdentifiedObject” has one characteristic: a cardinality that is 0 to many. In contextual model, “Names” **object property** of “IdentifiedObject” **structured class** could have the same characteristic of its CIM counterpart or a more restricted one: example, cardinality of “name” could be restricted to 1 or 1 to many. So “Names” contextual artefact is defined as a subset of “name” CIM artefact.

For more details on how a contextual model artefact is related to a CIM artefact, see 62361-101.

———————

[2] In the next edition of this document, the content of this clause will be moved into IEC 62361-101, currently under consideration

[3] Currently in Committee Draft (CD) at the time of this writing.

4.3.2 Contextual artefacts

Contextual model artefacts are listed and defined in Table 1: Contextual model artefacts

Contextual model artefact	Definition
Structured class	subset of a non-stereotyped CIM class with additional restrictions. A structured class may have zero or more object properties, compound properties, and simple properties. Any sub class of a structured class is also a structured class.
Concrete class	a structured class that can be instantiated. Concrete class does not necessarily result in a root element. This concept is not used for this JSON schema mapping.
Abstract class	a structured class that cannot be instantiated. This concept is not used for this JSON schema mapping.
Superclass	relative to a given class, a more general class whose extent is a superset of the given class.
Subclass	relative to a given class, a more specific class whose extent is a subset of the given class.
Root class	a class that may have standalone instances which are not the referent of any object property. A contextual model may assign cardinality bounds to a root class limiting the number of standalone instances that may occur.
Union class	a subset of a non-stereotyped CIM superclass defined as a **union** of (some of) its sub classes. Each member of the union is defined as a structured class. Each of these is a sub class of a single, given CIM class. An instance of a union is an instance of one of its constituent structured classes. Example: in CIM, “RegisteredResource” is a superclass of “RegisteredLoad”, “RegisteredTie” and “RegisteredGenerator”. In contextual model, “RegisteredResource” could be a superclass of some of these sub classes. When defined as a union, “RegisteredResource” defined the set of the sub classes (“RegisteredLoad”, “RegisteredTie”…) that are going to be used as the referent classes for the “RegisteredResource” **object property** that in this case will be a **union object property** (see below). Note: this feature is used to get all the elements representing sub classes instances in random order.
Compound class	subset of a CIM class defined as **Compound** with additional restrictions. An instance of a compound class is a structured value. It has one or more properties, but it has no identity distinct from the combination of its property values.

Contextual model artefact	Definition
Basic type	CIM class defined as a **Primitive** (includes **Integer, Decimal, Boolean**, **Duration**, **DateTime**, **Date**, **Time**, **MonthDay**, **String**, **Float**, **Double**[4]). A basic type may be used directly in a contextual model without further definition. By design the JSON schema specification intentionally has a reduced data type set comprised of the following: • string • number • integer • boolean • array • object (i.e. a JSON object) • null The JSON data types of `string`, `integer`, `number`, and `boolean` are therefore those relevant for the mapping of CIM classes defined as **Primitive**. JSON schema data types have the following value spaces: • string – double-quoted with support for Unicode with backslash escaping. • number – double-precision floating-point format. No NaN or Infinity is used in `number`. Can be expressed via an exponent part by beginning with the letter E in uppercase or lowercase, which may be followed by a plus or minus sign. The E and optional sign are followed by one or more digits. • integer – for consistency integer JSON numbers should not be encoded with a fractional part. • boolean – `true` or `false`. • array – an ordered sequence of values. • object (i.e. JSON object) – an unordered collection of key:value pairs. • null – empty

[4] To be defined in a future CIM version.

Contextual model artefact	Definition

Simple type	a subset of a CIM class defined as "Primitive" with additional restrictions.
	As defined above, the value range of such a CIM class is assumed to be one of the JSON schema datatypes referenced.
	The additional restrictions narrow this value range by defining one or more facets for that datatype. Example:
	
	"**$defs**": {
	"TwentyFourCharString" : {
	"**type**": "*string*",
	"**maxLength**": 24
	}
	}
	
	"TwentyFourCharString" is a string whose maximum length is 24 characters.
	A simple type instance does not have simple properties or object properties and has no identity distinct from its value.

Data type	a subset of a CIM class defined as **CIMDatatype** with additional restrictions. A "data type" is a class whose instances carry a value and other properties that give meaning to this value. Data type value and other data type properties could be restricted by additional constraints. An instance of a "data type" class is a structured value. It has one or more properties, but it has no identity distinct from the combination of its property values.
Enumeration class	subset of an **enumeration** CIM class with additional restrictions.
CodeList class	a subset of an **enumeration** CIM class and marked as **CodeList**. Each instance of the enumeration is associated to a "code" whose type is one of the "Basic type".
Simple property	a subset of a CIM class attribute with additional restrictions. The type of a simple property is a **Basic type**, a **Simple type**, a **Data type** or an **Enumeration** Class.
Compound property	subset of a CIM class attribute whose referent is a class defined as a **Compound**.
Object property	a subset of a CIM association with additional restrictions and a specific direction from referring class to referent class. The referent of an object property is an instance of a structured class. The restrictions may narrow the referring or referent classes or place bounds on the cardinality of the object property.
By-reference object property	a subset of a CIM association, as per object property, defined as **by-reference**. The referent of a by-reference property is either an instance of a structured class or an external instance. An external instance is assumed to exist but is not described in the present message. Pragmatically, a "by-reference" property is implemented by quoting the referent's identifier (example "mRID").

Contextual model artefact	Definition
Union object property	object property defined as **union** whose referent class is a superclass or an object property whose referent class is a union class. In CIM, “ResourceCapacity” has an association with “RegisteredResource”, a superclass of “RegisteredLoad”, “RegisteredTie” and “RegisteredGenerator”. The association has two end role names: “ResourceCapacity” and “RegisteredResource”. In contextual model, “ResourceCapacity” could have the **object property** “RegisteredResource” whose referent class is “RegisteredResource”. If this object property is marked as **union** or if the “RegisteredResource” referent class is marked as **union**, then the “RegisteredResource” **object property** is a **union object property**.
Exclusive property group	restriction on a structured class with respect to a group of properties such that only one of the properties may appear in a given instance of the class.
Documentation	prose description accompanying a definition in the CIM or the contextual model.
Categorized documentation	prose description accompanying a definition in the CIM or in the contextual model together with some classifying properties which indicate the category and purpose of the description. This concept is not used for this JSON schema mapping.
Stereotype	an identifier associated with a contextual model class or property that qualifies its usage or semantics, but not its JSON schema mapping. The meaning of each stereotype must be provided in documentation accompanying the contextual model.

Table 1 – Contextual model artefacts

4.4 Mapping a contextual model to a JSON schema

4.4.1 General

The mapping determines:

- a single, standalone JSON schema for a given contextual model;
- indirectly, the syntax of the instance JSON documents to be exchanged;
- the relationship between definitions in the JSON schema and definitions in the contextual model;
- indirectly, the relationship between elements in the JSON documents exchanged and the definitions in the CIM.

The mapping is applied at design time to map a CIM profile to an IETF JSON schema.

In this mapping, a profile defines the semantics of a single type of message payload that will be encoded in JSON.

Therefore:

- The syntax or semantics of any envelope or headers that may be added to the instance documents when they are exchanged is not specified.
- Both the contextual model and its mapped JSON schema are design artefacts and are not necessarily required at the time instance JSON documents are exchanged.
- There is, in general, more than one JSON schema that expresses the same syntax. All these are equivalent from an interoperability perspective. However, some forms of JSON schema can be preferred over others purely from the perspective of design tools.

- The method used to exchange instance JSON documents is not specified.

4.4.2 Traceability

In this mapping, the relationship between elements in the JSON documents exchanged and the definitions in the model of which the contextual model is a subset are expressed. To keep track of these relationships, the mapping defined in this document uses a conceptual derivation of the semantic annotation as defined in "Semantic Annotations for WSDL and XML Schema W3C Recommendation". Details on the derivation are outlined in a later clause.

4.5 JSON schema Representation

The JSON schema mappings will be presented using a notation consisting of an outline of a JSON schema construct with the following conventions:

- Mandatory keywords/attributes are shown in bold, e.g. **`$id`**
- Optional keywords/attributes are shown in standard, e.g. `minItems`
- Literal values corresponding to keywords/attributes are shown in italics e.g. *`object`*
- Alternatives attribute values are shown in brackets separated by vertical bars. e.g. (*object* | *string* | *number*) and if there is a default value it is shown after a colon, e.g.: `object`
- The content of the schema element is introduced by `Content:`
- Content grammar is enclosed in brackets with a separating comma for concatenation or vertical bar for alternatives. e.g. `(annotation, (all | any*))`
- The Kleene operators: **?**, **+**, and ***** are used for at most one, at least one, and any number of repetitions respectively.
- Simple words in plain face refer to definitions elsewhere. e.g. `annotation`

4.6 Namespaces

It should be noted that JSON and the JSON schema specificiation do not natively define namespace support as is commonly understood in the realm of XML and XML Schema.

THIS NAMESPACE CLAUSE PENDING - UNDER FINAL DRAFT REVIEW

5 Mapping specifications

5.1 General

The following clauses define the mapping of a generic contextual model to a JSON schema.

Figure 1 serves as an example to illustrate certain constructs introduced in the following clauses.

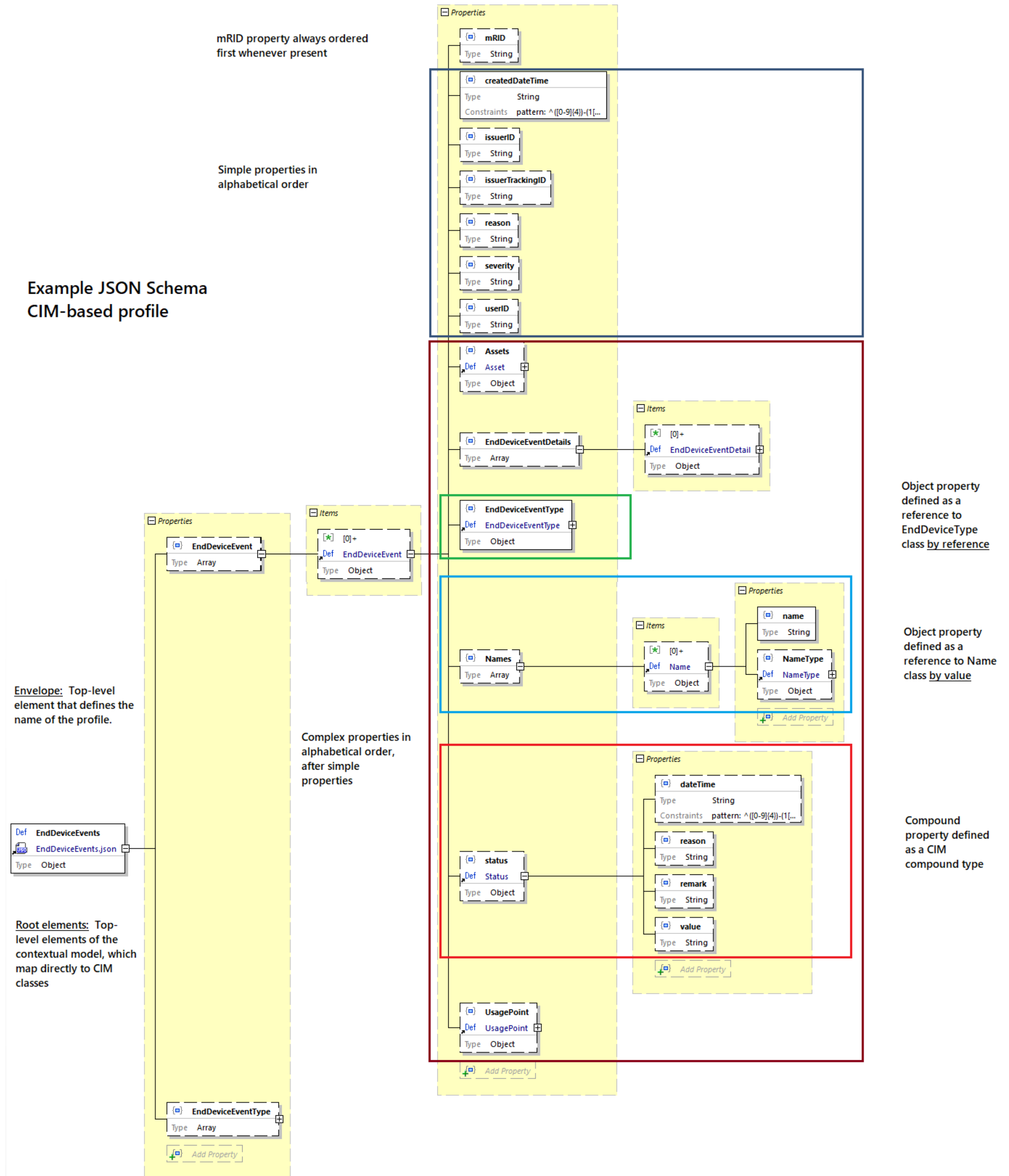


Figure 1 – Example JSON schema CIM-based profile

5.2 Profile mapping

5.2.1 Preliminary Background

This document defines a CIM profile to JSON schema mapping specification that directly applies to the draft 2020-12 version of the JSON schema specification. In addition to conforming to draft 2020-12, the mappings defined herein also conform to draft-07 of the JSON schema specification with any variations explicitly noted where applicable.

When applying this specification for mappings to draft-07 instead of 2020-12 the following variation should be applied:

All references to the "`$defs`" keyword within the JSON schema examples in this document should instead reference and map to the draft-07 "`definitions`" keyword. This keyword was renamed as of draft 2020-12.

For the profile mappings that follow in later clauses, the IETF JSON Pointer specification [RFC6901] is utilized for referencing mapped JSON subschema types. JSON Pointer is a syntax for specifying locations in the JSON schema profile, starting from the document root. JSON Pointer is intended to be easily expressed in JSON string values as well as Uniform Resource Identifier (URI) fragment identifiers.

Most JSON schemas use JSON references to minimize duplication. A JSON reference to a subschema may look similar to the following:

```
{
    "$ref": "https://domain.com/external.schema.json#/$defs/<external-subschema-name>"
}
```

or

```
{
      "$ref": "#/$defs/<internal-subschema-name>"
}
```

The `$ref` keyword serves as the JSON Pointer to a subschema within an external schema, an internal subschema, or a type of property in a subschema. It will appear throughout the clauses that follow and for the purposes of this mapping specification is constrained to only references to internally defined subschemas within the `$defs` element of the JSON schema profile being mapped.

5.2.2 General

A single profile is mapped to a single JSON schema with the following form:

```
{
      "$id": "<Absolute canonical URI for the schema>",
      "$schema": "https://json-schema.org/draft/2020-12/schema",
      "title": "<Title for the profile>",
      "description": "<annotation for the profile>",
      "namespace": "<namespace-uri>",
      "type": "object",
      "additionalProperties": false,
      "properties": {
          Content: (root-type*)
      },
      "$defs": {
          Content: (envelope-type, (complex-type, simple-type, enum-type)*)
      }
}
```

Within the JSON schema specification, the `$defs` keyword provides a standardized location for defining inline re-usable JSON subschemas. Later clauses of this document specify how the various contextual model types map to this location.

5.2.3 Profile $id, $schema, title, description, namespace, and type

The following set of JSON schema keywords and custom keywords must be represented within the root header of all JSON profiles that conform to this mapping specification.

5.2.3.1 The JSON schema `$id` identifier keyword

The value of `$id` for a given profile conforming to this specification must be a canonical URI identifying the profile of the standard form: *<canonical-base-uri>/<envelope-name>.schema.json*

Specifically, for an IEC published profile, the value of $id must identify the profile and the year the corresponding standard is released. The URI of an IEC published profile is defined in the iec.ch domain and is of the standard form: *https://iec.ch/TC57/<year>/<envelope-name>.schema.json*

Note that as previously mentioned, there is no official file extension for JSON schema, but an extension of `.schema.json` has been recommended. For the purposes of this specification, this file extension is to be adhered to for IEC profiles based on it so as to broadly promote conformity within IEC standards dependent upon it.

This URI will also be used to derive a base URI for the purposes of resolving `$ref` references in the subschema mappings within a standalone profile. Further detail on the use of `$id` (s) and `$ref` JSON Pointers within subschemas appears later in this document.

For an extensive overview of these keywords and how they are resolved in various vendor implementations refer to Annex E.

5.2.3.2 The JSON schema `$schema` keyword

As defined in the JSON schema specification, the `$schema` keyword is both used as a JSON schema version identifier and the location of a resource which is itself a JSON schema, which describes any schema written for the particular JSON schema version that the CIM profile is being mapped to. The value of this keyword must be a URI [RFC3986] (containing a scheme) and this URI must be normalized. The profile schema must be valid against the meta-schema identified by this URI. If this URI identifies an actual retrievable resource then that resource should be of media type *application/schema+json*. The `$schema` keyword should be used in a root schema. It must not appear in subschemas within the single profile. Values for this property are defined in other documents and by other parties.

The following `$schema` URIs must be applied when mapping this specification to the indicated JSON schema version:

To conform to draft-07:
https://json-schema.org/draft-07/schema#

To conform to draft 2020-12:
https://json-schema.org/draft/2020-12/schema

Note that when applying this specification for draft-07 mappings the draft-07 URI would be substituted

within the examples in this document.

5.2.3.3 The JSON schema `title` keyword

The JSON schema `title` keyword serves to provide a simple, non-verbose title for identification of the profile (e.g. "EndDeviceEvents"). For this mapping specification the value of `title` is of the standard form: *<envelope-name>*

5.2.3.4 The JSON schema `description` keyword

The `description` keyword specifies a verbose annotation designating the purpose of the profile and any context relevant to the profile's usage. If no verbose annotation is defined for the `description` keyword then an empty string is to be specified.

5.2.3.5 The `namespace` keyword

The `namespace` keyword is a custom keyword applicable only in the context of profiles conforming to this specification. Its value must be a `namespace-uri` that identifies the profile and the year the corresponding standard was released. The `namespace` for each IEC standard profile is allocated by the IEC in the `iec.ch` domain and is of the form: *http://iec.ch/TC57/<year>/<envelope-name>#* (e.g. http://iec.ch/TC57/2011/EndDeviceEvents#)

Finally, the `envelope-name` chosen must be such that the combination of `iec.ch` domain, year, and `envelope-name` produces a `namespace-uri` that is unique among all published schema profiles.

5.2.3.6 The JSON schema `type` keyword

As defined in the JSON schema specification, the JSON schema `type` keyword specifies the JSON data type associated with a JSON schema. The mapping of a single CIM profile to a single JSON schema will always result in a JSON type of `object` for the root JSON schema definition. This top-level root JSON schema `object` definition corresponds to the `envelope-type` definition which is detailed in full in a later clause.

5.2.4 Envelope element

The **Envelope** is the top-level root element of the schema. It is not defined in the contextual model and therefore must not have the name used by any class from the contextual model. It is an element that characterizes the profile (it is the name of the profile). In Figure 1, this is the **EndDeviceEvents** element.

The `envelope-elem` is a single top-level element definition in the profile and maps the `envelope-name` as a JSON subschema definition in the `$defs` element of the JSON schema:

```
{
      ...
      "$defs": {
          // The profile envelope root element definition
           "<envelope-name>": {
                 "$ref": "#"
           },
            ...
      }
}
```

The above envelope-elem definition must be defined in the $defs element of the schema profile and serves to define the top-level global type definition for the envelope-type and is represented as a JSON Pointer self-reference ("#"). Correspondingly, for Figure 1 the envelope-name for this envelope-elem would be **EndDeviceEvents**.

Continuing, for the root type definition of the envelope-type, the following details the mapping that comprises this definition. Specifically, the mapping consists of a corresponding top-level JSON properties definition for each root-prop corresponding to a root class appearing in the profile. The properties are to appear in alphanumeric order by property name:

```
{
     ...
     "type": "object",
     "additionalProperties": false,
     "properties": {
          // each root class maps to a JSON root property definition
          Content: (root-prop*)
      },
     "$defs": {
          "<envelope-name>": {
               "$ref": "#"
          },
              ...
     }
}
```

The next clause details the specifics as to how each root class is mapped to a root-prop definition in the schema.

5.2.5 Root elements

Root elements are the top elements of the contextual model. In Figure 1, it is the **EndDeviceEvent** and **EndDeviceEventType** elements.

Each root-prop is defined as a property in the root-level JSON schema properties and maps to one of the two following JSON schema constructs depending on the max cardinality of the root class in the contextual model. For a max cardinality greater than one, the root class will map to a JSON schema array definition as follows:

```
"<root-name>": {
      "type": "array",
      "items": {
         "$ref": "#/$defs/<root-name>",
      },
      "minItems": <min-items-count>,
      "maxItems": <max-items-count>
}
```

As outlined in the JSON schema specification, the minItems and maxItems array keywords are optional and when not specified, they default to 0 and unbounded respectively. For this mapping specification, these attributes should only appear in a JSON schema profile if they are to contain a value other than the aforementioned defaults.

Applying this mapping to our working example from Figure 1 would result in the **EndDeviceEvent** and **EndDeviceEventType** root classes being mapped as shown next. Note that the JSON $ref in the properties definitions that follow are JSON Pointers that

reference JSON subschema defined within the `$defs` element of the JSON schema. If expanded in this example, the subschema would contain the detailed mapping for the `root-name` of the contextual class corresponding to the `root-prop(s)`:

```
{
      ...
      "type": "object",
      "additionalProperties": false,
      "properties": {
           "EndDeviceEvent": {
                "type": "array",
                "items": {
                     "$ref": "#/$defs/EndDeviceEvent"
                }
            },
            "EndDeviceEventType": {
                 "type": "array",
                 "items": {
                      "$ref": "#/$defs/EndDeviceEventType"
                 }
            }
       },
      "$defs": {
           "EndDeviceEvents": {
                "$ref": "#"
           },
            ...
           // The mappings for the EndDeviceEvent and EndDeviceEventType
           // subschemas referenced by the $ref JSON Pointers above will
           // have subschema definitions declared here.  For the sake of
           // brevity they have been omitted.
      }
}
```

Transitioning on to the second mapping scenario where the root class in the contextual model has a `max` cardinality equal to one. The mapping of a root class for this translates to the following simple JSON schema property definition:

```
"<root-name>": {
      "$ref": "#/$defs/<root-name>",
}
```

In this scenario, no equivalent concept of `minItems` is relevant for this non-array property, rather the `min` cardinality is understood to be either zero or one, while the `max` cardinality, as just highlighted, is implicitly understood to be one. When `min` is one it is considered required and the mapping must also include a JSON schema `required` construct:

```
{
       ...
       "type": "object",
       "additionalProperties": false,
       "properties": {
           "<root-name>": {
                "$ref": "#/$defs/<root-name>",
            }
       },
       // The name of each required property must be included:
      "required": [
```

```
        "<root-name>"
    ],
    "$defs": {
        ...
    }
}
```

Again, applying this to our working example. Assuming for the sake of illustration that in Figure 1 both root classes had their cardinality as `min` equals one and `max` equals one, then their profile mapping would result in:

```
{
    ...
    "type": "object",
    "additionalProperties": false,
    "properties": {
        "EndDeviceEvent": {
            "$ref": "#/$defs/EndDeviceEvent"
        },
        "EndDeviceEventType": {
            "$ref": "#/$defs/EndDeviceEventType"
        }
    },
    "required": [
        "EndDeviceEvent",
        "EndDeviceEventType"
    ],
    "$defs": {
        ...
    }
}
```

Finally, it should be stated that in each of these mapping scenarios, the `root-name` corresponds to the name of the respective root class as it appears in the contextual model being mapped.

5.2.6 Semantic annotation

Within a JSON schema mapping, the goal of traceability is an important one.

In the IEC 62361-100 "CIM profile to XML Schema mapping" specification, this concern was addressed via the use of semantic annotations as defined by the W3C recommendation: "Semantic Annotations for WSDL and XML Schema". Given that this W3C recommendation defines a set of extension attributes for the Web Services Description Language (WSDL) and XML Schema definition language (XSD), it was a natural fit for traceability. However, no comparable concept has yet been formalized within the context of the JSON schema specification or its supporting specifications. Very early work within the W3C Web of Things Working Group has occurred but has not resulted in a formalized approach. With the much greater flexibility introduced via vocabularies in draft 2020-12, the introduction of a vocabulary in support of a future formal semantic annotations specification could easily be introduced once defined, but that is beyond the present scope of this document.

Therefore, in the interim, a conceptual derivation of the W3C recommended semantic annotations follows that aims to accommodate for the same general traceability goals, but within the context of this JSON schema mapping specification. Note that such annotations serve to identify the origin of the type or attribute that appears within the defined JSON schema.

In order to ensure traceability, for each JSON schema element, complex type, and simple type defined in the following clauses, a custom JSON schema "`modelReference`" attribute serves as a semantic annotation and defined as an absolute URIRef (that is the namespace URI within the information model). The JSON schema specification is flexible in that it allows for such custom keywords (keyword/value pairs) to be specified within a JSON schema file.

The attribute definition for this semantic annotation is:

```
"modelReference": (class-ref | referent-class-ref | prop-ref | type-ref
| enum-ref | codelist-ref | enum-value-ref)
```

where:

- `class-ref` is a URIRef designating the structured or compound class (CIM class) of which this contextual model class is a subset. In Fig 1, “Name” is a JSON **object subschema**, mapped from the contextual “Name” structured class, that itself is a subset of the CIM “Name” class, so `class-ref` for the Name JSON **object subschema** is for example: `http://iec.ch/TC57/2016/CIM-schema-cim16#Name`
- `referent-class-ref` is a URIRef designating the structured or compound class (CIM class) of which a contextual model **referent** class is a subset. This is only utilized in the context of “by-reference” object properties. In Fig 1, “EndDeviceEventType” is a JSON **object subschema**, mapped from the contextual “EndDeviceEventType” structured class, itself is a subset of the CIM EndDeviceEventType class. Further, the “EndDeviceEvent” is a JSON **object subschema**, mapped from the contextual “EndDeviceEvent” structured class, itself is a subset of the CIM EndDeviceEvent class. A property named “EndDeviceEventType” defined within the CIM EndDeviceEvent class is declared as a “by-reference” object property whose referent is the EndDeviceEventType class. Therefore, the `referent-class-ref` for the EndDeviceEventType JSON **object subschema** is for example : `http://iec.ch/TC57/2016/CIM-schema-cim16#EndDeviceEventType`
- `prop-ref` is the URIRef designating the property definition (CIM attribute or association) of which this contextual model property is a subset. In Fig 1, “name” is an element, mapped from the contextual “name” **simple property**, that itself is a subset of CIM “name” attribute, so `prop-ref` for “name” element is for example: `http://iec.ch/TC57/2016/CIM-schema-cim16#Name.name`. In Fig 1, “Names” is a property, mapped from the contextual “Names” **object property**, that is a subset of CIM “Names” end role name, so `prop-ref` for “Names” element is for example: `http://iec.ch/TC57/2016/CIM-schema-cim16#IdentifiedObject.Names`.
- `type-ref` is a URIRef designating the class (a CIM Primitive or CIMDatatype class) of which this type is a subset. In Fig 1, `string` is the type of “createdDateTime”, `string` is mapped from the contextual “DateTime” **basic type**, which corresponds to the CIM Primitive “DateTime” so `type-ref` for `string` is for example: `http://iec.ch/TC57/2016/CIM-schema-cim16#DateTime`.
- `enum-ref` is a URIRef designating the enumeration (CIM enumeration) of which this contextual model enumeration is a subset. “UsagePointConnectedKind” is a JSON **object subschema**, mapped from contextual “UsagePointConnectedKind” **enumeration class**, which is a subset of CIM “UsagePointConnectedKind” enumeration, so `enum-ref` for “UsagePointConnectedType” element is for example : `http://iec.ch/TC57/2016/CIM-schema-cim16#UsagePointConnectedKind`.
- `enum-value-ref` is the URIRef designating the enumeration value of the enumeration of which this contextual model enumeration is a subset. “connected” is one of the “UsagePointConnectedKind” **string enumeration** value, mapped from contextual “UsagePointConnectedKind” enumeration “connected” enum-value, which is a subset of CIM “UsagePointConnectedKind” enumerated literals, so `enum-value-ref` for

"connected" enumeration value is for example: `http://iec.ch/TC57/2016/CIM-schema-cim16#UsagePointConnectedKind.connected`.

- `codelist-ref` is a URIRef designating the enumeration of which this contextual model class is a subset.

URIRef(s) are compliant with IETF RFC 3986. The IEC format for these model references take the form `http://iec.ch/TC57/<year>/CIM-schema-cimXX#<class-name>` or `http://iec.ch/TC57/<year>/CIM-schema-cimXX#<class-name>.<property-name>`. Where the `namespace` specifies a `namespace-uri` that identifies both the year (`<year>`) and version (`cimXX`) associated with the published CIM standard that the contextual model is derived from. So in the examples above, URIRefs are associated with the 6th Edition of the IEC 61970-301 CIM standard that was published in 2016 and based on IEC61970**cim16**v33.

5.3 Structured classes

Each structured class in the contextual model, including each root class, is mapped to a JSON subschema definition as follows:

```
{
     ...
     "$defs": {
          "<class-name>": {
               "description": "<annotation>",
               "modelReference": "<class-ref>",
               "type": "object",
               "additionalProperties": false,
               "properties": {
                    Content: ((whole-class | derived-class))
               }
          }
     }
}
```

The `class-name` is the mapped name of the contextual model class.

The `class-ref` is a URIRef designating the class (CIM class) of which this contextual model class is a subset.

The `annotation` content carries documentation from the contextual model and the CIM. This is detailed in the Annotation clause of this document.

The remaining content to appear within the `properties` JSON schema element depends on whether the contextual model class is a sub class of a corresponding contextual model superclass.

If the contextual class is a subclass, the `derived-class` form applies. For this form, given that the JSON schema specification does not support direct extension capabilities as commonly understood in other schema specifications (e.g. XSD); the mapping of "inherited" attributes and associations from the superclass is handled, therefore, in the following manner:

```
Super Class Content:((simple-prop | typed-prop | ref-prop | union-prop |
compound-prop | exclusive-prop)*)

Sub class Content:((simple-prop | typed-prop | ref-prop | union-prop |
compound-prop | exclusive-prop)*)
```

Note that for the content in this mapping of the derived-class form, the attributes and associations from the superclass are ordered and appear within the JSON schema properties element before those of the sub class being mapped.

As for the whole-class form, the content mapping is straight forward and appears as follows:

```
Content:((simple-prop | typed-prop | ref-prop | union-prop | compound-prop
| exclusive-prop)*)
```

The content shown consists of a set of local properties definitions for each property defined in the contextual model as a member of the class.

The JSON properties definitions appear in alphanumeric order by name. However, if one of the JSON property elements is the result of the mapping of the contextual model mRID simple property this JSON property element must appear first. For further details and examples see the clause 5.16 “Mapping Order” later in this document.

The form of each of these JSON property definitions depends on the type of the property and is described in the next set of clauses.

5.4 Compound classes

Each compound class is mapped to a JSON subschema definition as follows:

```
{
      ...
     "$defs": {
          // Each subschema definition appears within the
          // "$defs" element of the profile.
          "<class-name>": {
               "description": "<annotation>",
               "modelReference": "<class-ref>",
               "type": "object",
               "additionalProperties": false,
               "properties": {
                      Content:((simple-prop | typed-prop | compound-prop)*)
               }
          }
          ...
     }
}
```

The class-name is the mapped name of the contextual model compound class.

The class-ref is a URIRef designating the class (CIM class) of which this contextual model compound class is a subset.

The annotation content carries documentation from the contextual model and the CIM. This is detailed in the Annotation clause.

The content consists of one local properties definition for each property defined in the contextual model as a member of the compound class.

The JSON properties definitions appear in alphanumeric order by element name.

The form of each JSON `properties` definition depends on the type of property and is described in the following clauses.

5.5 Basic types

The Basic types within the CIM model as listed in Table 2 have fixed mappings to the JSON schema primitive datatypes as shown. Given that these are Basic types, no higher-order JSON `object` type definitions are created for them in the JSON schema's `$defs` element. Rather, they are reduced to a simple JSON schema primitive type definition that may appear only as a property within a `properties` element defined as part of a JSON subschema definition.

For the date-, time-, and duration-related CIM types in the table, the prescriptive annotation within the CIM model is included for the purposes of highlighting the intent of the respective Basic type. It should be noted that for the mappings of these types to their corresponding JSON schema primitive type that a default `pattern` facet is to be included in the mapping to the JSON schema primitive type shown. Corresponding `pattern` facets, where applicable, are indicated in the table below and, as per the JSON schema specification, are valid regular expressions conforming to clause 15.10.1 in the ECMA 262 regular expression dialect. The regular expressions enforce the representation of date- and time-related data as defined by the ISO 8601 "Data elements and interchange formats – Information interchange – Representation of dates and times" standard and are expressed in the ISO 8601 extended format.

<table>
<tr><th>Basic Type</th><th>Mapped Type</th></tr>
<tr><td>Boolean</td><td>boolean</td></tr>
<tr><td>Decimal</td><td>number</td></tr>
<tr><td>Float</td><td>number</td></tr>
<tr><td>Integer</td><td>integer</td></tr>
<tr><td>Double[5]</td><td>number</td></tr>
<tr><td>String</td><td>string</td></tr>
<tr><td rowspan="3">Date</td><td>CIM model notes:
Date as "yyyy-mm-dd", which conforms with ISO 8601. UTC time zone is specified as "yyyy-mm-ddZ". A local timezone relative UTC is specified as "yyyy-mm-dd(+/-)hh:mm".</td></tr>
<tr><td>string</td></tr>
<tr><td>^(-?([1-9][0-9]{3,}|0[0-9]{3})-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])(Z|(\\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?)$</td></tr>
<tr><td rowspan="3">DateTime</td><td>CIM model notes:
Date and time as "yyyy-mm-ddThh:mm:ss.sss", which conforms with ISO 8601. UTC time zone is specified as "yyyy-mm-ddThh:mm:ss.sssZ". A local timezone relative UTC is specified as "yyyy-mm-ddThh:mm:ss.sss-hh:mm". The second component (shown here as "ss.sss") could have any number of digits in its fractional part to allow any kind of precision beyond seconds.</td></tr>
<tr><td>string</td></tr>
<tr><td>^(-?([1-9][0-9]{3,}|0[0-9]{3})-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])T(([01][0-9]|2[0-3]):[0-5][0-9]:[0-5][0-9](\\.[0-9]+)?|(24:00:00(\\.0+)?))(Z|(\\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?)$</td></tr>
</table>

[5] To be defined in a future CIM version.

<table>
<tr><td rowspan="3">Duration</td><td>CIM model notes:

Duration as "PnYnMnDTnHnMnS" which conforms to ISO 8601, where nY expresses a number of years, nM a number of months, nD a number of days. The letter T separates the date expression from the time expression and, after it, nH identifies a number of hours, nM a number of minutes and nS a number of seconds. The number of seconds could be expressed as a decimal number, but all other numbers are integers.</td></tr>
<tr><td>string</td></tr>
<tr><td>^(-?)P(?=\\d|T\\d)(?:(\\d+)Y)?(?:(\\d+)M)?(?:(\\d+)([DW]))?(?:T(?:(\\d+)H)?(?:(\\d+)M)?(?:(\\d+(?:\\.\\d+)?)S)?)?$</td></tr>
<tr><td rowspan="3">Time</td><td>CIM model notes:

Time as "hh:mm:ss.sss", which conforms with ISO 8601. UTC time zone is specified as "hh:mm:ss.sssZ". A local timezone relative UTC is specified as "hh:mm:ss.sss±hh:mm". The second component (shown here as "ss.sss") could have any number of digits in its fractional part to allow any kind of precision beyond seconds.

The smallest value used may also have a decimal fraction, as in "P0.5Y" to indicate half a year. This decimal fraction may be specified with either a comma or a full stop, as in "P0,5Y" or "P0.5Y"</td></tr>
<tr><td>string</td></tr>
<tr><td>^((([01][0-9]|2[0-3]):[0-5][0-9]:[0-5][0-9](\\.[0-9]+)?|(24:00:00(\\.0+)?))(Z|(\\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?)$</td></tr>
<tr><td rowspan="3">MonthDay</td><td>CIM model notes:

MonthDay format as "--mm-dd", which conforms with XSD data type gMonthDay and ISO8601:2000 Date without year format. “Truncated representation, as specified in [ISO.8601.2000], Sections 5.2.1.3 d), e), and f), is permitted.”</td></tr>
<tr><td>string</td></tr>
<tr><td>^(--(((02)-(0[1-9]|[1][0-9]|2[0-9]))|((0[4689]|(11))-(0[1-9]|[1][0-9]|2[0-9]|(30)))|((0[13578]|(1[02]))-(0[1-9]|[1][0-9]|2[0-9]|(3[0-1])))))$</td></tr>
</table>

Table 2 – Basic Types

Is is important to note that for all JSON schema profiles generated according to this specification that any regular expressions specified within JSON schema `pattern` keywords within the profile MUST utilize a double escape sequence (e.g. \\) instead of the typical single escape sequence normally expressed in regular expressions.

This is required as the JSON schema specification is dependent upon and must adhere to the ECMA-404 “JSON Data Interchange Syntax” standard. Per the ECMA-404, a string is a sequence of Unicode code points wrapped with quotation marks (U+0022). All code points may be placed within the quotation marks except for the code points that must be escaped which, per the standard, includes the reverse solidus (U+005C) i.e. backslash (\) character. The above regular expressions prescribed by this standard take this into account and should be used in the exact format as shown. For further information on the JSON data interchange syntax refer to Annex A in this document.

Applying this to our working example in Figure 1, the **EndDeviceEvent** class has a `createdDateTime` attribute of the CIM Basic type: DateTime. The mapping for this property as specified within Table 2 above would, therefore, result in:

```
{
      ...
     "$defs": {
          "EndDeviceEvent": {
               "description": "Event detected by a device function
associated with end device.",
               "modelReference": "http://iec.ch/TC57/2010/CIM-schema-
cim15#EndDeviceEvent",
               "type": "object",
               "additionalProperties": false,
               "properties": {
                    ...
                    // Simple property mapping of the createdDateTime
                    // attribute results in the following definition...
                    "createdDateTime": {
                         "description": "<attribute annotation>",
                         "modelReference": "http://iec.ch/TC57/2010/CIM-
schema-cim15#ActivityRecord.createdDateTime",
                         "type": "string",
                         // The default pattern facet for ISO-8601 datetime
                         // is included in this basic type mapping
                        "pattern": "^(-?([1-9][0-9]{3,}|0[0-9]{3})-(0[1-
9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])T(([01][0-9]|2[0-3]):[0-5][0-9]:[0-5][0-
9](\\.[0-9]+)?|(24:00:00(\\.0+)?))(Z|(\\+|-)((0[0-9]|1[0-3]):[0-5][0-
9]|14:00))?)$"
                    }
                    ...
               }
          }
     }
}
```

5.6 Simple types

5.6.1 Mapping rules

Each simple type defined in the contextual model is to have a `simple-type` definition mapped to a JSON subschema definition as follows:

```
{
      ...
     "$defs": {
          // Each simple type subschema definition appears
         // within the "$defs" element of the profile.
          "<type-name>": {
               "description": "<annotation>",
               "modelReference": "<type-ref>",
               "type": "<base-type>",
                Content:(facet*)
          }
          ...
     }
}
```

The `type-name` is the mapped name of the contextual model simple type.

The `type-ref` is a URIRef designating the class (CIM class) of which this type is a subset.

The `base-type` is a corresponding JSON primitive type specified in the contextual model (see Table 2).

Each facet is a JSON schema facet corresponding to a facet restriction given in the contextual model for the simple type.

The `facet` definition has the form:

```
{
        ...
      "$defs": {
           "<type-name>": {
                "description": "<annotation>",
                "modelReference": "<type-ref>",
                "type": "<base-type>",
                // Each facet definition appears after the "type"
                // that it applies to.  Facets may only be those
                // relevant to the specified <base-type> as shown
                // in Table 3:
                "<facet-name>": "<facet-value>"
                // ...or alternatively if the facet-value is numerical:
                "<facet-name>": <facet-value>
           }
           ...
      }
}
```

The `facet-name` is one of the allowed facet names as defined by the JSON schema specification and described in Table 3 below.

Example:

```
{
      "$defs": {
           "NonNegativeInteger": {
                "description": "Type used for non-negative integers.",
                "modelReference": "http://iec.ch/TC57/2016/CIM-schema-
cim16#Integer",
                "type": "integer",
                // The (inclusive) minimum facet constraint applied for a
                // non-negative integer
                 "minimum": 0
           }
      }
}
```

This shows the `minimum` (i.e. inclusive minimum) JSON schema facet applied with a value of zero as the `facet-value`.

5.6.2 Possible facets

The possible facets are listed for each basic type in Table 3.

Basic Type	**JSON Primitive Type Mapping**	**Facet**
Boolean	`boolean`	*No facet*

String	`string`	`minLength`
		`maxLength`
		`pattern`
		Per the JSON schema specification, pattern facets must be valid regular expressions conforming to the ECMA 262 regular expression dialect.
Integer	`integer`	`minimum` (inclusive minimum)
		`maximum` (inclusive maximum)
		`exclusiveMinimum`
		`exclusiveMaximum`
		`multipleOf`
		Ranges of numbers are specified using a combination of the `minimum` and `maximum` facets, (or `exclusiveMinimum` and `exclusiveMaximum` for expressing exclusive range). For a value x, the following application of the respective facets holds true: x ≥ `minimum` x > `exclusiveMinimum` x ≤ `maximum` x < `exclusiveMaximum` While it is technically possible to specify both of `minimum` and `exclusiveMinimum` or both of `maximum` and `exclusiveMaximum`, it does not make practical sense to do so. Numbers can be restricted to a multiple of a given number, using the `multipleOf` keyword. It may be set to any positive number.
Decimal **Float** **Double**[6]	`number`	`minimum` (inclusive minimum)
		`maximum` (inclusive maximum)
		`exclusiveMinimum`
		`exclusiveMaximum`
		`multipleOf`

[6] To be defined in next CIM version.

		Ranges of numbers are specified using a combination of the `minimum` and `maximum` facets, (or `exclusiveMinimum` and `exclusiveMaximum` for expressing exclusive range). For a value x, the following application of the respective facets holds true: x ≥ `minimum` x > `exclusiveMinimum` x ≤ `maximum` x < `exclusiveMaximum` While it is technically possible to specify both of `minimum` and `exclusiveMinimum` or both of `maximum` and `exclusiveMaximum`, it does not make practical sense to do so. Numbers can be restricted to a multiple of a given number, using the `multipleOf` keyword. It may be set to any positive number.
Date **Time** **DateTime** **Duration** **MonthDay**	`string`	`pattern`
		Per the JSON schema specification, the `pattern` facet must be a valid regular expression conforming to the ECMA 262 regular expression dialect. For date and time types mapping to a JSON primitive `string` the `pattern` should represent date- and time-related data as defined by the ISO 8601 "Data elements and interchange formats – Information interchange – Representation of dates and times" standard and should be expressed in the ISO 8601 extended format. Note, for date- and time-related mappings to JSON primitives, the use of the `minLength` and `maxLength` facets, though permitted, are not applicable in the context of this mapping specification. Additionally, `minimum` and `maximum` facets as defined in XSD schema for dates have no equivalent in JSON schema and therefore the concepts are not supported.

Table 3 – Facets

These facets are the current set as defined within the JSON schema specification and applicable to this mapping document.

For those familiar with XSD schema and the IEC 62361-100 NDR standard a full set of XSD to JSON schema facet mappings can be found in Annex C.

5.7 DataTypes mapping

Each DataType defined in the contextual model is mapped to a JSON subschema definition:

```
{
     ...
    "$defs": {
        // Each CIM DataType subschema definition appears
        // within the "$defs" element of the profile.
        "<datatype-name>": {
            "description": "<annotation>",
            "modelReference": "<datatype-ref>",
            "type": "object",
            "additionalProperties": false,
            "properties": {,
```

```
                    Content: (value-prop, (related-prop)*)
                }
            }
        }
}
```

The `datatype-name` is the mapped name of the datatype name.

The `datatype-ref` is a URIRef of the CIMDatatype class (CIM class) of which this contextual model datatype is a subset.

Example: in Figure 2 CIM “ActivePower” is a “CIMDatatype”.

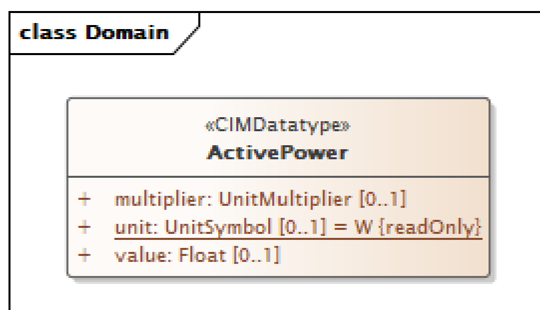


Figure 2 – Example CIMDatatype class

In the contextual model for this example, the corresponding artefact would be the “ActivePower” **data type**, which would map to a JSON subschema definition whose `datatype-name` would be “ActivePower” and whose `datatype-ref` would be “`http://iec.ch/TC57/2016/CIM-schema-cim16#ActivePower`”.

The content of a CIMDatatype is comprised of a `value-prop` and one or more `related-prop` definitions that are related to the `value-prop`. In the Figure 2 example, “ActivePower” has a “value” property declared along with two additional related properties of “unit” and “multiplier” that define the units and associated multiplier that the value property is expressed in. In this case, the unit is declared as a constant designating W (watts) as the units for the value.

The mapping for such CIMDatatype attributes is defined within a JSON `properties` element for the subschema definition for the **data type**. The mapping of the “value” is specifically dependent upon the referent type declared for the “value” attribute in the contextual model:

- If the referent is one of the basic types listed in clause 5.5 the value property’s `value-prop-type` is mapped to the given JSON schema primitive datatype as follows:

```
{
      ...
      "$defs": {
           "<datatype-name>": {
                ...
                "additionalProperties": false,
                "properties": {,
                     "value": {
                          "description": "<value property annotation>",
                          "modelReference": "<value-prop-ref>",
                           "type": "<value-prop-type>"
                     },
```

```
                    ...
                    // Related properties follow here...
                }
            }
        }
}
```

- If the referent is an enumeration class then the `value-prop-type` is its mapped name and is mapped using the JSON `$ref` keyword with a JSON Pointer reference to the corresponding enumeration type subschema definition. This subschema definition is that appearing elsewhere in the `$defs` element and mapped separately as defined within clause 5.8:

```
{
    ...
    "$defs": {
        "<datatype-name>": {
            ...
            "additionalProperties": false,
            "properties": {,
                "value": {
                    "description": "<value property annotation>",
                    "modelReference": "<value-prop-ref>",
                     "$ref": "#/$defs/<value-prop-type>"
                },
                ...
                // Related properties follow here...
            }
        }
    }
}
```

- If the referent is one of the basic types listed in clause 5.5 with additional restrictions, then the `value-prop-type` is mapped to the given JSON schema primitive datatype. The definition of the “value” `property` is constructed with the additional restrictions appearing as applicable facets as previously outlined in clause 5.6:

```
{
     ...
    "$defs": {
        "<datatype-name>": {
            ...
            "additionalProperties": false,
            "properties": {,
                "value": {
                    "description": "<value property annotation>",
                    "modelReference": "<value-prop-ref>",
                    "type": "<value-prop-type>",
                     // Additional restrictions as facets
                    Content: (facets*)
                },
                ...
                // Related properties follow here...
            }
        }
    }
}
```

In the working example from Figure 2, the "ActivePower" CIMDatatype has a "value" attribute whose value space is defined by a type that is the CIM "Primitive" **Float**, and with related "unit" and "multiplier" attributes declared as CIM **UnitSymbol** and **UnitMultiplier** enumerations respectively.

Applying the previously described mapping variants for the "value" `value-prop-type` to this "ActivePower" example and introducing mappings for the related attributes results in the following three mappings. Note that the requirements for `related-prop-name` mappings is described in detail afterwards:

- for the first mapping variant, as declared the "value" property value space or type is the **basic type** "simple precision float" and therefore the `value-prop-type` will be the JSON schema primitive type `number`:

```
{
     "$defs": {
          "ActivePower": {
               "description": "Product of RMS value of the voltage and the 
RMS value of the in-phase component of the current.",
               "modelReference": "http://iec.ch/TC57/2016/CIM-schema-
cim16#ActivePower",
               "type": "object",
               "additionalProperties": false,
               "properties": {
                    "value": {
                         "description": "The value for active power.",
                         "modelReference": "http://iec.ch/TC57/2016/CIM-
schema-cim16#ActivePower.value",
                         // JSON primitive type mapping...
                         "type": "number"
                    },
                    "unit": {
                         "description": "The unit of the value.",
                         "modelReference": "http://iec.ch/TC57/2016/CIM-
schema-cim16#ActivePower.unit",
                         "allOf": [
                              {"$ref": "#/$defs/UnitSymbol"},
                              {"const": "W"}
                         ]
                    },
                    "multiplier": {
                         "description": "The unit multiplier of the value.",
                         "modelReference": "http://iec.ch/TC57/2016/CIM-
schema-cim16#ActivePower.multiplier",
                         "$ref": "#/$defs/UnitMultiplier"
                    }
               },
               "required": [
                    "value"
               ]
          }
     }
}
```

- if the "value" property value space or type is restricted to a "Float" with additional restrictions such as only positive values (facet `minimum` = 0), the `value-prop-type` will be the JSON schema primitive type `number` with the additional restriction expressed as the JSON facet keyword `minimum` (inclusive) whose value is "0":

```
{
      ...
     "$defs": {
          "ActivePower": {
               "description": "Product of RMS value of the voltage and the
RMS value of the in-phase component of the current.",
               "modelReference": "http://iec.ch/TC57/2016/CIM-schema-
cim16#ActivePower",
               "type": "object",
               "additionalProperties": false,
               "properties": {
                    "value": {
                         "description": "The value for active power.",
                         "modelReference": "http://iec.ch/TC57/2016/CIM-
schema-cim16#ActivePower.value",
                         "type": "number",
                         // JSON schema facet introduced for positive values
                         "minimum": 0
                    },
                    "unit": {
                         "description": "The unit of the value.",
                         "modelReference": "http://iec.ch/TC57/2016/CIM-
schema-cim16#ActivePower.unit",
                         "allOf": [
                              {"$ref": "#/$defs/UnitSymbol"},
                              {"const": "W"}
                         ]
                    },
                    "multiplier": {
                         "description": "The unit multiplier of the value.",
                         "modelReference": "http://iec.ch/TC57/2016/CIM-
schema-cim16#ActivePower.multiplier",
                         "$ref": "#/$defs/UnitMultiplier"
                    }
               },
               "required": [
                    "value"
               ]
          }
     }
}
```

- if the “value” property value space or type is instead restricted to a named **enumeration** for example “ActivePowerValueKind”, the subschema `value-prop-type` will be “ActivePowerValueKind” and will map to a `$ref` JSON Pointer:

```
{
     "$defs": {
          "ActivePower": {
               "description": "Product of RMS value of the voltage and the
RMS value of the in-phase component of the current.",
               "modelReference": "http://iec.ch/TC57/2016/CIM-schema-
cim16#ActivePower",
               "type": "object",
               "additionalProperties": false,
               "properties": {
                    "value": {
                         "description": "The value for active power.",
                         "modelReference": "http://iec.ch/TC57/2016/CIM-
schema-cim16#ActivePower.value",
```

```
                        // JSON Pointer referencing the enumeration...
                         "$ref": "#/$defs/ActivePowerValueKind"
                    },
                    "unit": {
                        "description": "The unit of the value.",
                        "modelReference": "http://iec.ch/TC57/2016/CIM-
schema-cim16#ActivePower.unit",
                        "allOf": [
                            {"$ref": "#/$defs/UnitSymbol"},
                            {"const": "W"}
                        ]
                    },
                    "multiplier": {
                        "description": "The unit multiplier of the value.",
                        "modelReference": "http://iec.ch/TC57/2016/CIM-
schema-cim16#ActivePower.multiplier",
                         "$ref": "#/$defs/UnitMultiplier"
                    }
                },
                "required": [
                    "value"
                ]
            }
        }
}
```

Finally, the above examples illustrate the mapping for `related-prop(s)` within CIMDatatypes. This also represents the approach for all possible `related-prop` (s) with the following factors to be considered when mapping:

- the referent type of a `related-prop` directs the manner in which the property is mapped. It will result in one of two possible forms depending on whether or not the referent is one of the basic types listed in Table 2.

- if a related property in the contextual model specifies an initial value then the JSON schema `default` keyword is included in the mapping to reflect the value specified.

- if the related property in the contextual model is marked as a read-only constant (as was the case for "unit" in the working example when declared as "W" for Watt units) then the JSON schema `const` keyword is included in the mapping to reflect the read-only value for the property. Note that only one of either `default` or `const` keywords should be in the mapping but not both.

The mapping for `related-prop` referents that are one of the basic types listed in clause 5.5:

```
{
        "properties": {
            ...
            "<related-property-name>": {
                "description": "<related prop annotation>",
                "modelReference": "<related-prop-ref>",
                "allOf": [
                    {
                        // For a basic type listed in clause 5.5
                         "type": "<related-type-name>"
                    },
                    {
                        // The const keyword is ONLY included when a
```

```
                        // related property is declared as a read-only
                        // constant in the contextual model.
                        "const": "<constant-value>"

                        // OR...

                        // alternatively, use the following when a
                        // related property in the contextual model
                        // is declared as having an initial or default
                        // value that is not read-only.
                        "default": "<default-value>"
                    }
                ]
            }
        },
        "required": [
            // Included as required only if min cardinality is greater
            // than 0 in the contextual model...
            <related-property-name>
        ]
}
```

The mapping for `related-prop` referents that are not one of the basic types:

```
{
        "properties": {
            ...
            "<related-property-name>": {
                "description": "<related prop annotation>",
                "modelReference": "<related-prop-ref>",
                "allOf": [
                    {
                        // For all other referent types not in the set of
                        // basic types
                        "$ref": "#/$defs/<related-type-name>"
                    },
                    {
                        // The const keyword is ONLY included when a
                        // related property is declared as a read-only
                        // constant in the contextual model.
                        "const": "<constant-value>"

                        // OR...

                        // alternatively, use the following when a
                        // related property in the contextual model
                        // is declared as having an initial or default
                        // value that is not read-only.
                        "default": "<default-value>"
                    }
                ]
            }
        },
        "required": [
            // Included as required only if min cardinality is greater
            // than 0 in the contextual model...
            <related-property-name>
        ]
}
```

The `related-prop-name` is the mapped name of a related property (i.e. any property related to the datatype's "value" property). Example "unit", "multiplier", or any other additional related properties that may appear in the datatype.

The form of the `related-type-name` depends on the referent of the datatype's related property:

- If the referent is one of the basic types listed in clause 5.5 the related property's `related-type-name` is mapped to the corresponding JSON schema primitive datatype and is expressed in the JSON schemas as: **"type":** "<related-type-name>"
- If the referent is an enumeration class then the `related-type-name` is its mapped name. This designates the corresponding type definition described in clause 5.8.

The `default-value` string is the value of the initial value if one is set for the contextual model property. When specified the JSON `default` keyword is used to specify the `default-value`.

The `constant-value` string is the value of the fixed value if one is set for the contextual model property. When specified the JSON `const` keyword is used to specify the `constant-value`.

The two `constant-value` and `default-value` are mutually exclusive when used.

Note in these mappings that when a default value or read-only constant is declared in the contextual model then the application of the JSON `allOf` keyword is used to ensure that only that value is allowed for the `related-type-name` property. Using an example, the application of JSON schema's `allOf` will validate that the value is both a valid value in the value space for the property (e.g. one of the valid enumerated values for UnitSymbol) as well as that the value is constrained to a specific value (e.g. via the `const` keyword).

5.8 Enumeration classes mapping

Each enumeration class defined in the contextual model is mapped to an `enum-type` JSON subschema definition as follows:

```
{
      ...
      "$defs": {
           // Each enumeration subschema definition must appear
           // within the "$defs" element of the JSON schema
           // profile.
           "<enum-name>": {
                "description": "<annotation>",
                "modelReference": "<enum-ref>",
                "type": "<enum-type>",
                "enum": [
                     Content: (enum-value*)
                ]
           }
           ...
      }
}
```

The `enum-name` is the mapped name of the contextual model class.

Then `enum-ref` is a URIRef designating the enumeration (CIM enumeration) of which this contextual model enumeration is a subset.

As defined within the JSON schema specification, enum constructs are defined using the JSON `enum` keyword which is declared as a JSON array of a specified `type`. Further, the `type` is one of JSON schema's primitive types with this enumerations mapping constraining the acceptable types to that of `string`, `number`, or `integer`. In practice, most enum mappings from the conceptual model will utilize the `string` type. Thus, the `enum-type` shown will map to one of these basic types.

Each `enum-value` in the array represents one value of the enumeration defined in the contextual model. The `enum-value` is therefore the mapped name of the enumeration value.

5.9 CodeList classes

Each codelist class defined in the contextual model is mapped to a `codelist-type` JSON subschema definition defined as follows:

```
{
      ...
      "$defs": {
           // Each codelist subschema definition must appear
           // within the "$defs" element of the JSON schema
           // profile.
           "<codelist-name>": {
                "description": "<annotation>",
                "modelReference": "<codelist-ref>",
                "type": "<codelist-type>",
                "enum": [
                     Content: (codelist-value*)
                ]
           }
           ...
      }
}
```

The `codelist-name` is the mapped name of the contextual model class.

Then `codelist-ref` is a URIRef designating the enumeration or codelist of which this contextual model class is a subset.

Codelists also utilize the JSON schema `enum` keyword for its mapping of codelists defined in the contextual model. Thus, the `codelist-type` shown in the codelists maps to one of the basic types as previously described for mapping enumeration classes: `string`, `number`, or `integer`.

Each `codelist-value` in the array represents one value from the codelist defined in the contextual model. The `codelist-value` itself is therefore the mapped name of the code value.

For a detailed overview of externalizing codelists in order to add custom local extensions refer to Annex E.

5.10 Simple properties mapping

Each simple property in the contextual model is mapped to a local JSON `property` defined within the `properties` element of the respective JSON subschema:

```
{
      ...
      "$defs": {
```

```
        "<class-name>": {
            ...
            "type": "object",
            "additionalProperties": false,
            "properties": {
                // Each simple property is defined within the
                // properties element of the class.
                "<prop-name>": {
                    "description": "<property annotation>",
                    "modelReference": "<prop-ref>",
                    // For a basic types listed in clause 5.5
                    "type": "<type-name>",
                    // Or for all other referent types...
                    "$ref": "#/$defs/<type-name>",
                }
                ...
            }
        }
    }
}
```

Here the `prop-name` is the mapped name of the property.

The `prop-ref` is the URIRef designating the property definition (CIM property) of which the contextual model property is a subset.

In the example from Figure 1 createdDateTime is a property of the EndDeviceEvent element. In the contextual model, createdDateTime is a simple property of the EndDeviceEvent class. This simple property is related to a corresponding CIM attribute. In CIM the corresponding attribute is the “createdDateTime” attribute defined within the “ActivityRecord” which is inherited by “EndDeviceEvent”. Thus the prop-ref is the URIRef `http://iec.ch/TC57/2016/CIM-schema-cim16#ActivityRecord.createdDatetime`

Finally, the `type-name` may be expressed in one of two forms for a JSON property. As to which form is required, it is dependent upon the type declared on the referent of the simple property being mapped. The following outlines the various types of referents and the corresponding JSON `property` definitions that are to be mapped. Note that the **EndDeviceEvent** and **EndDeviceEventDetail** classes from Figure 1 serve to provide working examples of these various referent types:

- If the referent is one of the basic types listed in Table 2 there will be no JSON subschema definition in the `$defs` element of the JSON schema. Therefore, the `type-name` is simply specified using the name of the relevant JSON primitive datatype.

```
{
    ...
    "$defs": {
        "<class-name>": {
            ...
            "type": "object",
            "additionalProperties": false,
            "properties": {
                "<prop-name>": {
                    "description": "<property annotation>",
                    "modelReference": "<prop-ref>",
                    "type": "<type-name>"
                }
                ...
```

```
            }
        }
    }
}
```

Example: in the contextual model, issuerID is a “String” **basic type**. So, the mapped `type-name` is specified as the JSON primitive type `string`.

```
      "properties": {
           "issuerID": {
                 "description": "Unique identifier of the business
entity originating an end device control.",
                 "modelReference": "http://iec.ch/TC57/2010/CIM-
schema-cim15#EndDeviceEvent.issuerID",
                 "type": "string"
           }
      }
```

Now, per the JSON schema specification, in this instance the issuerID has a minimum cardinality of 0 and a maximum cardinality of 1. A property defined in this manner is always implied to have these lower and upper boundaries for its cardinality. If the simple property is required in the profile (i.e. a minimum cardinality of 1) then the JSON schema `required` keyword must be added to the above mapping definition. The value of the `required` keyword is an array, with the elements contained therein referring to the name of the JSON property(s) that are required. Note that these cardinality conventions are applicable to all simple property referent types.

```
      "properties": {
           "issuerID": {
                 "description": "Unique identifier of the business
entity originating an end device control.",
                 "modelReference": "http://iec.ch/TC57/2010/CIM-
schema-cim15#EndDeviceEvent.issuerID",
                 "type": "string"
           }
           ...
      },
      "required": [
           "issuerID"
      ]
```

- If the referent is a **simple type** then the `$defs` element of the JSON schema will contain a JSON subschema definition for that simple type. In this case, a JSON Pointer must be used within the mapping to refer to the JSON subschema definition of the referent. The `type-name` of the JSON Pointer is the referent’s mapped name. This pointer designates the corresponding type definition described in clause 5.6 and mapped as the JSON subschema for the simple type.

```
{
      ...
      "$defs": {
           "<class-name>": {
                 ...
                 "type": "object",
                 "additionalProperties": false,
                 "properties": {
                      "<prop-name>": {
```

```
                        "description": "<property annotation>",
                        "modelReference": "<prop-ref>",
                        // JSON Pointer to subschema definition
                        "$ref": "#/$defs/<type-name>"
                    }
                }
                ...
            },
            ...
            // The simple type's subschema is defined within this
            // $defs element as well, but omitted here for
            // the sake of brevity.
        }
    }
}
```

Example: in the contextual model, the **EndDeviceEventDetail** class has a simple property named “value” which is declared to be a “StringQuantity” **simple type**. The mapping for this referent would appear as the following JSON Pointer:

```
{
    ...
    "$defs": {
        ...
        "EndDeviceEventDetail": {
            "type": "object",
            "description": "Name-value pair, specific to end device
events.”,
             "modelReference": "http://iec.ch/TC57/2010/CIM-schema-
cim15#EndDeviceEventDetail",
            "additionalProperties": false,
            "properties": {
                ...
                "value": {
                    "description": "<property annotation>",
                    "modelReference": "http://iec.ch/TC57/2010/CIM-
schema-cim15#EndDeviceEventDetail.value",
                    // The JSON Pointer to the simple
                    // type's subschema...
                    "$ref": "#/$defs/StringQuantity"
                }
                ...
            },
            "StringQuantity": {
                "description": "Quantity with string value (when it
is not important whether it is an integral or a floating point
number) and associated unit information.",
                "modelReference": "http://iec.ch/TC57/2010/CIM-
schema-cim15#StringQuantity",
                "type": "string",
                ...
            },
            ...
    }
}
```

- If the referent is a **data type** then the `$defs` element of the JSON schema will contain a JSON subschema definition for that data type. In this case, a JSON Pointer must be used within the mapping to refer to the JSON subschema definition of the referent. The

`type-name` of the JSON Pointer is the referent's mapped name. This pointer correlates to the corresponding type definition described in clause 5.7 and mapped separately as the JSON subschema for the data type.

Example: in CIM "EndDeviceInfo" "ratedVoltage" attribute has CIMDatatype "Voltage" for type. In the contextual model, "ratedVoltage" **simple property** will have "Voltage" **datatype** type. Thus, the mapped `type-name` will be "Voltage" and will be represented in the same manner using a JSON Pointer as illustrated in the previous mapping example for the **simple type**.

- If the referent is an **enumeration** class then the `$defs` element of the JSON schema will contain a JSON subschema definition for that enumeration. In this case, a JSON Pointer must be used within the mapping to refer to the JSON subschema definition of the referent. The `type-name` of the JSON Pointer is the referent's mapped name. This pointer designates the corresponding type definition described in clause 5.8 and mapped as the JSON subschema for the enumeration.

Example: in CIM "ComFunction" "technology" attribute has "ComTechnologyKind" enumeration for type. In the contextual model, the "technology" **simple property** will have "ComTechnologyKind" **enumeration** type. Thus, the mapped `type-name` will be "ComTechnologyKind" and will be represented in the same manner as illustrated for the mapping example for the **simple type**.

- If the referent is a **codelist** class then the `$defs` element of the JSON schema will contain a JSON subschema definition for that codelist. In this case a JSON Pointer must be used within the mapping to refer to the JSON subschema definition of the referent. The `type-name` of the JSON Pointer is the referent's mapped name. This pointer designates the corresponding type definition described in clause 5.9 and mapped as the JSON subschema for the codelist.

5.11 Compound properties mapping

Each compound property in the contextual model has for referent a compound class and is mapped to a local JSON `property` defined within the `properties` element of the respective JSON subschema:

```
{
     ...
     "$defs": {
          "<class-name>": {
               ...
               "type": "object",
               "additionalProperties": false,
               "properties": {
                    // Each compound property is defined within the
                    // properties element of a mapped JSON subschema.
                    "<prop-name>": {
                         "description": "<property annotation>",
                         "modelReference": "<prop-ref>",
                         "$ref": "#/$defs/<type-name>",
                    }
                    ...
               }
          }
     }
}
```

Example of a compound property from Figure 1 is the element **status** under **EndDeviceEvent**.

The prop-name is the mapped name of the property.

The compound-name is the mapped name of the referent compound class.

The prop-ref is the URIRef designating the property definition (CIM property) of which this contextual model property is a subset.

```
      "properties": {
          "status": {
                "description": "Information on consequence of event
resulting in this activity record.",
                "modelReference": "http://iec.ch/TC57/2010/CIM-schema-
cim15#ActivityRecord.status",
                "$ref": "#/$defs/Status"
          }
      }
```

As outlined in the previous clause, per the JSON schema specification, in this instance the status has a minimum cardinality of 0 and a maximum cardinality of 1. A compound property defined in this manner is always implied to have these lower and upper boundaries for its cardinality. If the compound property is required in the profile (i.e. a minimum cardinality of 1) then the JSON schema required keyword must be added to the above mapping definition.

```
      "properties": {
          ...
          "status": {
               "description": "Information on consequence of event resulting
in this activity record.",
               "modelReference": "http://iec.ch/TC57/2010/CIM-schema-
cim15#ActivityRecord.status",
               "$ref": "#/$defs/Status"
          }
          ...
      },
      "required": [
          "status"
      ]
```

5.12 Object properties

5.12.1 Mapping rules overview

Each object property in the contextual model is mapped to a typed-prop, a ref-prop, or a union-prop as a local JSON property defined within the properties element of the respective JSON subschema.

5.12.2 Typed object properties mapping

If the referent of the object of the property is a structured class then the typed-prop form applies. Two forms of mapping exist depending on the cardinality of the upper bound.

For an object property with an upper bound of 1 the following form applies:

```
{
    ...
    "$defs": {
        "<class-name>": {
            ...
            "type": "object",
            "additionalProperties": false,
            "properties": {
                // Each object property is defined within the
                // properties element of a mapped JSON subschema.
                "<prop-name>": {
                    "description": "<property annotation>",
                    "modelReference": "<prop-ref>",
                    "$ref": "#/$defs/<class-name>",
                }
                ...
            },
            // Optionally specified depending on minimal cardinality.
            //  (when min >= 1)
            "required": [
                "<prop-name>"
            ]
        }
    }
}
```

As described earlier, the JSON schema `required` keyword should only include the property name if the minimum cardinality is 1.

For an object property with an upper bound greater than 1 then a mapping form that utilizes a JSON schema `array` construct must be applied:

```
{
    ...
    "$defs": {
        "<class-name>": {
            ...
            "type": "object",
            "additionalProperties": false,
            "properties": {
                // Each object property is defined within the
                // properties element of a mapped JSON subschema.
                "<prop-name>": {
                    "description": "<property annotation>",
                    "modelReference": "<prop-ref>",
                    "type": "array",
                    "items": {
                        "$ref": "#/$defs/<class-name>"
                    },
                    // The value assigned to minItems must match
                    // the lower bound as defined for the
                    // association in the contextual model.
                    // If the cardinality of the association is
                    // 1..4 then both minItems & maxItems would
                    // be declared as follows:
                    "minItems": 1,
                    "maxItems": 4,
                }
                ...
            },
            // Optionally specified depending on minimal cardinality.
            //  (when min >= 1)
```

```
                "required": [
                    "<prop-name>"
                ]
            }
        }
}
```

This mapping highlights several import concepts around how cardinality constraints are expressed in the JSON schema construct for a JSON array. Here the optional JSON array-specific keywords `minItems` and `maxItems` are included in the example.

Per the JSON schema specification, when not specified `minItems` and `maxItems` default to 0 and unbounded respectively. The `maxItems` should only be specified when an array is to be contrained to a specific maximum. When the minimum cardinality is greater than or equal to 1 then both the `minItems` keyword and the `required` array must be specified. The value of `minItems` much match the minimum cardinality as defined in the contextual model and the property name must be added to the `required` array. For cases where the minimum cardinality is exactly 1 in the contextual model it is not sufficient to specify the property in the `required` array, but yet to omit the `minItems` keyword. In this scenario, `minItems` must also be declared with a value of 1.

An example of a typed object property in Figure 1 is the element **Names** under **EndDeviceEvent**. It is easy to recognise, because it results in nesting, i.e., further definition of sub-elements: **name** and **NameType**.

The `prop-name` is the mapped name of the property (in the example: **Names**). The `class-name` is the mapped name of the referent class (in the example: **Name**).

The `prop-ref` is the URIRef designating the property definition (CIM property) of which this contextual model property is a subset.

5.12.3 By-reference object properties mapping

If the object property is defined as “by-reference”, then it is mapped to a `ref-prop`.

The mapping for a `ref-prop` requires that a unique JSON subschema definition be defined within the schema’s `$defs` element that represents a “by-reference” corresponding to the particular contextual model class that is the referent. As such this is not a direct mapping representation of the contextual model class, but rather a representation of the “by-reference” itself. This style of mapping allows for stronger typing to be tied to the “by-reference” via the mapping of a single unique “by-reference” subschema definition per referent class. A `$ref` to this subschema is then used in the profile wherever a “by-reference” object property to the corresponding contextual model class is needed.

The “by-reference” mapping form has two possible variants depending upon cardinality.

For a “by-reference” object property with an upper bounds of 1 the following form applies:

```
{
    ...
    "$defs": {
        // The contextual model class containing the
        // "by-reference" object property
        "<class-name>": {
            "description": "<class annotation>",
            "modelReference": "<class-ref>",
            "type": "object",
            "additionalProperties": false,
            "properties": {
```

```
                    ...
                    // The class's "by-reference" object property
                    "<prop-name>": {
                         "description": "<property annotation>
                         "modelReference": "<prop-ref>",
                         // The JSON Pointer now references the
                         // referent class's unique "by-reference"
                         // subschema
                         "$ref": "#/$defs/<referent-class-name>Ref"
                    },
                    ...
               },
               // Optionally specified depending on minimal cardinality.
               // (when min >= 1)
               "required": [
                    "<prop-name>"
               ]
          },
          // The referent class's unique "by-reference" Ref type definition
          "<referent-class-name>Ref": {
               "description": "<referent annotation>",
               "modelReference": "<referent-class-ref>",
               "type": "object",
               "additionalProperties": false,
               "properties": {
                    "ref": {
                         "modelReference": "<referent-class-ref>",
                         "type": "string"
                    },
                    "referenceType": {
                         "type": "string"
                    },
               },
               // "ref" must always be declared as required
               "required": [
                    "ref"
               ]
          }
          ...
     }
}
```

As described earlier, JSON schema's `required` array should only include the property name if the minimum cardinality is greater than or equal to 1.

For a "by-reference" object property with an upper bound greater than 1, the mapping form that utilizes a JSON schema `array` construct must be applied:

```
{
     ...
     "$defs": {
          // The contextual model class containing the
          // "by-reference" object property
          "<class-name>": {
               "description": "<class annotation>",
               "modelReference": "<class-ref>",
               "type": "object",
               "additionalProperties": false,
               "properties": {
                    ...
                    // The class's "by-reference" object property
                    "<prop-name>": {
```

```
                    "description": "<property annotation>
                    "modelReference": "<prop-ref>",
                    "type": "array",
                    "items": {
                        // An array of JSON Pointers that reference the
                        // referent class's "by-reference" subschema
                        "$ref": "#/$defs/<referent-class-name>Ref"
                    },
                    "minItems": <min>,
                    "maxItems": <max>
                },
                ...
            },
            // Optionally specified depending on minimal cardinality.
            // (when min >= 1)
            "required": [
                "<prop-name>"
            ]
        },
        // The referent class's unique "by-reference" Ref type definition
        "<referent-class-name>Ref": {
            "description": "<referent annotation>",
            "modelReference": "<referent-class-ref>",
            "type": "object",
            "additionalProperties": false,
            "properties": {
                "ref": {
                    "modelReference": "<referent-class-ref>",
                    "type": "string"
                },
                "referenceType": {
                    "type": "string"
                },
            },
            // "ref" must always be declared as required
            "required": [
                "ref"
            ]
        }
        ...
    }
}
```

The `class-name` is the mapped name of class in the contextual model containing the "by-reference" object property.

The `class-ref` is a URIRef designating the class (CIM class) of which this contextual model class is a subset.

The `referent-class-name` is the mapped name of the class (CIM class) of which the contextual model referent is a subset.

The `prop-name` is the mapped name of the "by-reference" property.

The `prop-ref` is the URIRef designating the property definition (CIM property) of which this contextual model property is a subset.

The `referent-class-ref` is a URIRef designating the class (CIM class) of which the contextual model **referent** is a subset.

For the `array` mapping form, the value of `minItems` is the minimum cardinality of the property as defined in the contextual model. The value of `maxItems` is the maximum cardinality taking into account the cardinality defined in the CIM and any restrictions defined in the contextual model. (In the contextual model, if the minimum cardinality is 0 or the maximum cardinality is unbounded, then the `minItems` or `maxItems` respectively can be omitted according to the JSON schema specification.)

For the `array` mapping form, when the minimum cardinality is greater than or equal to 1 then the `minItems` keyword must be declared as part of the definition. In addition, the `prop-name` must be specified in the JSON `required` array. For cases where the minimum cardinality is exactly 1 it is not sufficient to specify the property in the `required` array but yet omit the `minItems` keyword. In this scenario, `minItems` must also be declared with a value of 1.

In a JSON instance file, the “ref” property defined here takes a string representing the identifier used to identify the property's referent: it could be the “mRID” or the “Names.name” of the instance referent class.

The referent may or may not be represented elsewhere in the instance. When the referent is not represented in the instance and thus is external to the instance, it may be defined in the contextual model as an abstract class with no concrete sub classes.

The “referenceType” property could be used to define which kind of identifier is used as the content of the “ref” property. For example, if the “mRID” property is used as an identifier, then the value of “referenceType” could optionally be specified and if so would have a value of “mRID”.

Finally, in this mapping form, the formal convention used for mapping the name of the referent class’s “by-reference” ref type should be highlighted. The convention dictates that a **Ref** suffix be appended to the end of the `referent-class-name` when defining the subschema definition.

It should be noted that though there may be more than one instance of the use of a “by-reference” object property for a particular referent class in the contextual model, there should be only one “by-reference” **Ref** subschema definition defined in the profile for that corresponding referent class.

To illustrate using a working example from Figure 1, the property named **EndDeviceEventType** under **EndDeviceEvent** is declared as a “by-reference” object property whose referent is the **EndDeviceEventType** class in the contextual model. Applying the second mapping option results in:

```
{
      ...
      "$defs": {
           ...
           "EndDeviceEvent": {
                "description": "Event detected by a device function
associated with end device.",
                "modelReference": "http://iec.ch/TC57/2010/CIM-schema-
cim15#EndDeviceEvent",
                "type": "object",
                "additionalProperties": false,
                "properties": {
                     ...
                     "EndDeviceEventType": {
                          "description": "Type of this end device event.",
                          "modelReference": "http://iec.ch/TC57/2010/CIM-
schema-cim15#EndDeviceEvent.EndDeviceEventType",
                          "$ref": "#/$defs/EndDeviceEventTypeRef"
```

```
                    },
                    ...
               },
               "required": [
                    "EndDeviceEventType"
               ]
          },
          "EndDeviceEventTypeRef": {
               "description": "Type of this end device event.",
               "modelReference": "http://iec.ch/TC57/2010/CIM-schema-
cim15#EndDeviceEventType",
               "type": "object",
               "additionalProperties": false,
               "properties": {
                    "ref": {
                         "modelReference": "<class-ref>",
                         "type": "string"
                    },
                    "referenceType": {
                         "type": "string"
                    },
               },
               "required": [
                    "ref"
               ]
          }
          ...
     }
}
```

5.12.4 Union object properties mapping

In the contextual model, a class may have an association with a superclass where the superclass is declared as a "Union" (meaning that sub classes are to be selected).

Figure 3 shows one possible example of a “Union”:

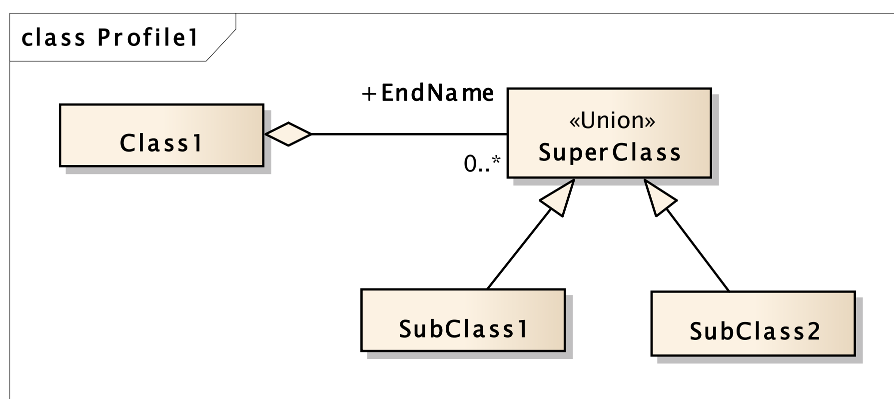


Figure 3 – CIM association from Class1 to a SuperClass declared as a “Union”

When the object property is marked as a union and its referent is a super class, or if the referent of the object of the property is an abstract super class marked as a union (see Annex D for examples), then the `union-prop` mapping form applies. This mapping will appear within the `properties` element of the JSON subschema defined for the owning class (Class1 in the Figure 3 example union).

The property names for each respective subclass are to be mapped depending upon how the union and role names appear within the canonical model. This is fully detailed in Annex D which covers all potential scenarios that may be encountered when mapping the names for the union subclasses.

The `union-prop` mapping may appear in one of two possible forms depending on the maximum cardinality of the union object property as expressed in the canonical model. If the maximum cardinality is greater than or equal to one then the mapping for each subclass will use an array-based mapping; otherwise a non- array-based standard mapping form is used.

Finally, the `union-prop` includes in its mapping form a specific approach to defining the constraints for required properties (i.e. when the minimum cardinality of the union object property is is greater than or equal to 1 in the canonical model).

Transitioning now to illustrations of the possible JSON schema mappings corresponding to the cardinality specified in the canonical model. For a union object property with an upper bounds of 1 the following `union-prop` standard form applies for each union subclass to be mapped into the owning class's `properties` element:

```
{
     ...
     "$defs": {
          "<class-name>": {
               ...
               "type": "object",
               "additionalProperties": false,
               "properties": {
                    // For each of the subclasses to be represented as part
                    // of the union a corresponding object property is to be
                    // defined within the properties element in the JSON
                    // subschema of the owning class...

                    // property for subclass 1
                    "<subclass1-prop-name>": {
                         "description": "<property annotation>",
                         "modelReference": "<prop-ref>",

                          Content: ((branch-prop*)|(branch-ref*))

                    },
                    // property for subclass 2
                    "<subclass2-prop-name>": {
                         "description": "<property annotation>",
                         "modelReference": "<prop-ref>",

                          Content: ((branch-prop*)|(branch-ref*))

                    },
                    ...
                    ...
                    // Nth property for the Nth subclass
                    "<subclassN-prop-name>": {
                         "description": "<property annotation>",
                         "modelReference": "<prop-ref>",

                          Content: ((branch-prop*)|(branch-ref*))

                    }
               },
```

```
                // Specialized mapping of the owning class's required
                // properties to account for union object properties when
                // they are also defined as part of the owning class in the
                // canonical model...
                "allOf": [
                    Content: ((required-non-union-props),
                              (required-union-props)*)
                ]
            }
        }
}
```

For a union object property with an upper bounds greater than or equal to 1 the following `union-prop` array-based mapping form applies for each union subclass to be mapped into the owning class's `properties` element:

```
{
    ...
    "$defs": {
        "<class-name>": {
            ...
            "type": "object",
            "additionalProperties": false,
            "properties": {
                // For union object properties with max cardinality > 1,
                // each subclass to be represented as part of the union
                // must have a corresponding object property defined
                // as an array within the properties element in the JSON
                // subschema of the owning class...

                // property defined as an array for subclass 1
                "<subclass1-prop-name>": {
                    "description": "<property annotation>",
                    "modelReference": "<prop-ref>",
                    "type": "array",
                    "items": {
                        Content: ((branch-prop*)|(branch-ref*))
                    },
                    "minItems": <min>,
                    // maxItems should only specified IF a specific
                    // maximum cardinality is to be specified (e.g. 3)
                    // When not specified it is unbounded...
                    "maxItems": <max>
                },

                // property defined as an array for subclass 2
                "<subclass2-prop-name>": {
                    "description": "<property annotation>",
                    "modelReference": "<prop-ref>",
                    "type": "array",
                    "items": {
                        Content: ((branch-prop*)|(branch-ref*))
                    },
                    "minItems": <min>,
                    // maxItems should only specified IF a specific
                    // maximum cardinality is to be specified...
                    "maxItems": <max>
                },
                ...
                ...

                // Nth property defined as an array for the Nth subclass
```

```
                    "<subclassN-prop-name>": {
                          "description": "<property annotation>",
                          "modelReference": "<prop-ref>",
                          "type": "array",
                          "items": {
                                Content: ((branch-prop*)|(branch-ref*))
                          },
                          "minItems": <min>,
                          // maxItems should only specified IF a specific
                          // maximum cardinality is to be specified...
                          "maxItems": <max>
                    }
               },
               // Specialized mapping of the owning class's required
               // properties to account for union object properties when
               // they are also defined as part of the owning class in the
               // canonical model...
               "allOf": [
                   Content: ((required-non-union-props),
                            (required-union-props)*)
               ]
          }
      }
}
```

The `class-name` is the mapped name of the class owning the `union` object property. In the example in Figure 3, this would correspond to “Class1”.

The `subclass-prop-name(s)` will be the result of applying an applicable change name rule (see clause 5.17 and detailed examples in Annex D) and could be one of the following:

- The subclass name,
- The subclass name prefixed by the same qualifier as the one used in the object property name itself, qualifier followed by an underscore,
- The subclass name prefixed by the object property name followed by an underscore.

The `prop-ref` is the URIRef designating the object property definition (CIM property) of which this contextual model property is a subset.

For the `union-prop` array-based mapping form the value of `minItems` is the minimum cardinality of the property as defined in the contextual model. The value of `maxItems` is the maximum cardinality taking into account the cardinality defined in the CIM and any restrictions defined in the contextual model. (In the contextual model, if the minimum cardinality is 0 or the maximum cardinality is unbounded, then the `minItems` or `maxItems` respectively can be omitted according to the JSON schema specification.)

When the minimum cardinality is greater than or equal to 1 then the `minItems` keyword must be declared as part of the `union-prop` array-based mapping form. In addition, the `union`'s property name must be specified as part of the `union-prop` specialized mapping for handling required union object properties. For cases where the minimum cardinality is exactly 1 it is not sufficient to specify the property in the specialized `required` array but yet omit the `minItems` keyword in the mapping of the property representing the subclass. In this scenario, `minItems` must also be declared with a value of 1 as part of the subclass's property.

The content is a choice of definitions representing properties whose referent are the sub classes of the referent `union` superclass. If the union property is by-reference, the `branch-ref` form applies; otherwise, the `branch-prop` content form applies.

The branch-prop form is as follows:

```
      "$ref": "#/$defs/<subclass-name>"
```

The subclass-name is the mapped name of a sub class of the referent union superclass.

Alternatively, the branch-ref form maps as:

```
      "$ref": "#/$defs/<subclass-name>Ref"
```

The branch-ref form results in the set of $ref pointers to the “by-reference” **Ref** subschemas representing the sub classes of the referent union superclass. Refer to the “By-reference object properties” clause for directives on mapping “by-reference” **Ref** subschemas.

As described in examples in earlier clauses, the typical approach for mapping required properties within a subschema definition is a declaration of the JSON schema required element with an array containing the property names which are required. This approach is insufficient to support the union-prop object property mapping form. A specialized mapping for required arrays is necessary and appears within a JSON schema allOf array defined as part of the union-prop mapping definition:

```
{
     "<class-name>": {
          ...
          // Specialized mapping of the owning class’s required
          // properties to account for union object properties.
          "allOf": [
                Content: ((required-non-union-props),
                          (required-union-props)*)
          ]
     }
}
```

The elements contained in the allOf array will consist of instances of two possible content mappings. The first is the required-non-union-props mapping form. This form is expressed as a JSON schema required array that contains an entry for every required object property in the owning class that is **not** a union property. If no properties meet this criteria then this entry is not included in this specialized required properties mapping. There can be only one required-non-union-props mapping within the allOf array and as a convention it should appear first. This required-non-union-props form maps as follows:

```
{
     "required": [
          "<non-union-prop-name-1>",
          "<non-union-prop-name-2>",
          ...
          "<non-union-prop-name-n>"
     ]
}
```

The non-union-prop-name(s) are the set of all mapped names of required properties that are not a union object property.

The second form is the required-union-props mapping form. This form must be used when a union object property is required (minimum cardinality >= 1). There must be one element of this form in the allOf array for each distinct union object property in the owning class that meets this criteria. So if there are two required (minimum cardinalty >= 1) union object properties for an owning class then there must be two required-union-props mappings defined within the allOf array.

The `required-union-props` form itself maps as a JSON schema `oneOf` array when the `union-prop` standard form is used (i.e. when a union object property's maximum cardinality = 1) and is mapped as follows:

```
{
      "oneOf": [
         { "required": ["<subclass1-prop-name>"] },
         { "required": ["<subclass2-prop-name >"] },
         ...
         ...
         { "required": ["<subclassN-prop-name >"] }
      ]
}
```

In this scenario the use of `oneOf` restricts the instance data to only a single instance of one of the possible options in the union.

Alternatively, the `required-union-props` form must appear as a JSON schema `anyOf` array when the array-based `union-prop` mapping form is used (i.e. when a union object property's maximum cardinality > 1):

```
{
      "anyOf": [
         { "required": ["<subclass1-prop-name>"] },
         { "required": ["<subclass2-prop-name >"] },
         ...
         ...
         { "required": ["<subclassN-prop-name >"] }
      ]
}
```

In this scenario the `anyOf` ensures that at least a single instance of one of the possible options for the union is present but does not restrict it to only a single instance as the use of a `oneOf` would do. In this way the use of an `anyOf` properly represents an unbounded upper cardinality for a required union object property.

The `subclass-prop-name(s)` that appear in the specialized required property mappings are names of the subclass properties resulting from applying an applicable change name rule for a union object property (see Annex D).

Transitioning to a working examples to illustrate union object property mappings in practice. For an initial example, an owning class has a required `mRID` property as well as a required union object property with 3 subclasses (`ComFunction`, `ConnectDiscnnectFunction`, `SimpleEndDeviceFunction`). The maximum cardinality for the owning class's union object property is 1 for the example. Applying the `union-prop` mapping results in the following:

```
{
   "$schema": "https://json-schema.org/draft/2020-12/schema",
   "title": "...",
   "description": "...",
   "namespace": "http://iec.ch/TC57/2020/ExampleUnion#",
   "type": "object",
   "additionalProperties": false,
   "properties": {
      "ExampleUnion": {
         "$ref": "#/$defs/ClassWithUnion"
      }
   },
   "$defs": {
      // Class definition owning the union object properties
      "ClassWithUnion": {
```

```
        "type": "object",
        "additionalProperties": false,
        "properties": {
          // Required non-union object property
          "mRID": {
            "type": "string"
          },
          // The following three subclass options are possible for the
          // required union object property for this example. It is
          // assumed that each property name for its respective subclass
          // is derived as described in Section 5.17 and in Annex D...
          "ComFunction": {
            "$ref": "#/defs/ComFunction"
          },
          "ConnectDisconnectFunction": {
            "$ref": "#/defs/ConnectDisconnectFunction"
          },
          "SimpleEndDeviceFunction": {
            "$ref": "#/defs/SimpleEndDeviceFunction"
          }
        },
        // Specialized mapping for handling required union properties.
        "allOf": [
          {
            // Mapping for all required properties that are not
            // union object properties.
            "required": ["mRID"]
          },
          {
            // Required mapping for the union property that has a
            // minimum and miximum cardinality both declared as 1.
            "oneOf": [
              { "required": ["ComFunction"] },
              { "required": ["ConnectDisconnectFunction"] },
              { "required": ["SimpleEndDeviceFunction"] }
            ]
          }
        ]
      },
      "ComFunction": {
        "type": "object",
        "additionalProperties": false,
        "properties": {
          // Not required
          "name": {
            "type": "string"
          }
        }
      },
      "ConnectDisconnectFunction": {
        "type": "object",
        "additionalProperties": false,
        "properties": {
          // Required
          "mRID": {
            "type": "string"
          },
          // Not required
          "isDelayedDiscon": {
            "type": "boolean"
          }
        },
        "required": [
```

```
            "mRID"
         ]
      },
      "SimpleEndDeviceFunction": {
         "type": "object",
         "additionalProperties": false,
         "properties": {
            // Required
            "mRID": {
               "type": "string"
            }
         },
         "required": [
            "mRID"
         ]
      }
   }
}
```

Continuing with this example let us assume that the `mRID` property is made optional. This would result in the specialized required properties portion of the mapping reduced to just the following:

```
{
         // Specialized mapping for handling required union properties:
         "allOf": [
            // Only the required mapping for the union property is needed
            // now...
            "oneOf": [
               { "required": ["ComFunction"] },
               { "required": ["ConnectDisconnectFunction"] },
               { "required": ["SimpleEndDeviceFunction"] }
            ]
         ]
}
```

Note that, if desired, the `allOf` may be removed altogether as the result of doing so is equivalent to the mapping above.

To apply a final working example for a `union-prop` mapping when the array-base mapping form is required. This example builds on that described previously with the exception that the union object property in this case has a maximum cardinality greater than 1. This results in the following variation for the `union-prop` mapping. Note that an `anyOf` must be used instead of a `oneOf` in the specialized required properties portion of the mapping:

```
{
   "$schema": "https://json-schema.org/draft/2020-12/schema",
   "title": "...",
   "description": "...",
   "namespace": "http://iec.ch/TC57/2020/ExampleUnion#",
   "type": "object",
   "additionalProperties": false,
   "properties": {
      "ExampleUnion": {
         "$ref": "#/$defs/ClassWithUnion"
      }
   },
   "$defs": {
      // Class definition owning the union object properties
      "ClassWithUnion": {
         "type": "object",
         "additionalProperties": false,
         "properties": {
            // Required non-union property
```

```
            "mRID": {
               "type": "string"
            },
            // The following three subclass options appear as arrays for
            // the required union object property for this example. It is
            // assumed that each property name for its respective subclass
            // is derived as described in Section 5.17 and Annex D...
            "ComFunction": {
               "type": "array"
               "items": {
                   "$ref": "#/defs/ComFunction"
               },
               "minItems": 1
            },
            "ConnectDisconnectFunction": {
               "type": "array"
               "items": {
                   "$ref": "#/defs/ConnectDisconnectFunction"
               },
               "minItems": 1
            },
            "SimpleEndDeviceFunction": {
               "type": "array"
               "items": {
                   "$ref": "#/defs/SimpleEndDeviceFunction"
               },
               "minItems": 1
            }
         },
         // Specialized mapping for handling required union properties.
         "allOf": [
            {
               // Mapping for all required properties that are not
               // union object properties.
               "required": ["mRID"]
            },
            {
               // Note that anyOf is used to define the required mapping
               // for a union property that has a minimum cardinality = 1
               // and maximum cardinality > 1 (unbounded)...
               "anyOf": [
                  { "required": ["ComFunction"] },
                  { "required": ["ConnectDisconnectFunction"] },
                  { "required": ["SimpleEndDeviceFunction"] }
               ]
            }
         ]
      },
      "ComFunction": {
         "type": "object",
         "additionalProperties": false,
         "properties": {
            // Not required
            "name": {
               "type": "string"
            }
         }
      },
      "ConnectDisconnectFunction": {
         "type": "object",
         "additionalProperties": false,
         "properties": {
            // Required
```

```
            "mRID": {
               "type": "string"
            },
            // Not required
            "isDelayedDiscon": {
               "type": "boolean"
            }
         },
         "required": [
            "mRID"
         ]
      },
      "SimpleEndDeviceFunction": {
         "type": "object",
         "additionalProperties": false,
         "properties": {
            // Required
            "mRID": {
               "type": "string"
            }
         },
         "required": [
            "mRID"
         ]
      }
   }
}
```

5.13 Exclusive property group mapping

Each exclusive property in a group must be mapped to the form relevant to it's referent type as prescribed in the corresponding object property clause for that referent type. For example, if one of the exclusive properties in the group has a referent that is a structured class then clause 5.12.2 for "Typed object properties mapping" would be applied or if it the object property's referent is a "by-reference", then the object property mapping in clause 5.12.3 for "By-reference object properties mapping" would be applied. This should occur for each exclusive property in the group with each of these distinct properties appearing within the `properties` element of the class containing the exclusive property group. The mapping order of the properties should conform to that as defined in clause 5.16.

Each exclusive property in the group, when mapped according to its appropriate referent type, will result in either its standard mapping form or its array-based mapping form depending on the cardinality of the exclusive property. This will be the natural outcome of compliance to the appropriate mapping for the property.

The `exclusive-props` form that follows maps each of the exclusive properties within a group into the `properties` element of the class that owns the group. It additionally defines a specialized mapping for handling the XOR required properties:

```
{
     ...
     "$defs": {
          // Class definition owning the exclusive property group
          "<class-name>": {
               ...
               "type": "object",
               "additionalProperties": false,
               "properties": {
                    ...
                    ...
```

```
                    // An exclusive property is to be defined for each
                    // exclusive property in the group:
                    "<exclusive-prop-name-1>": {
                         // Property definition relevant to referent type
                    },
                    "<exclusive-prop-name-2>": {
                         // Property definition relevant to referent type
                    },
                    ...
                    ...
                    "<exclusive-prop-name-n>": {
                         // Property definition relevant to referent type
                    }
               },
               // Specialized mapping of JSON schemas required arrays
               // to handle exclusive property groups when a group is
               // defined as part of a class in the canonical model.
               "allOf": [
                    Content: ((required-non-exclusive-props),
                              (exclusive-props-group-all-required |
                               exclusive-props-group-with-optionals)*)
               ]
          }
     }
}
```

The `exclusive-prop-name(s)` are the mapped names of the exclusive properties within the group.

The typical approach for mapping required properties within a subschema definition is a declaration of the JSON schema `required` element. This approach is insufficient to support XOR object property groups. A specialized mapping for `required` arrays is necessary and appears within a JSON schema `allOf` array defined as part of the `exclusive-props` mapping definition. The elements contained in the array will consist of one of three possible mappings.

The first is the `required-non-exclusive-props` mapping form. This form is expressed as a JSON schema `required` array that contains an entry for every required object property in the class that is **not** a member of an exclusive property group. If no properties meet this criteria then this entry is not included in the mapping. There can be only one entry of this form within the `allOf` array and as a convention it should appear first. This `required-non-exclusive-props` form maps as follows:

```
{
     "required": [
          "<non-exclusive-prop-name-1>",
          "<non-exclusive-prop-name-2>",
          ...
          "<non-exclusive-prop-name-n>"
     ]
}
```

The `non-exclusive-prop-name(s)` are the set of all mapped names of required properties that are not a member of an exclusive property group.

The second form is the `exclusive-props-group-all-required` mapping form. This form must be used if all members of an exclusive properties group are required (minimum cardinality >= 1). There should be one element of this form in the `allOf` array for each distinct exclusive properties group that meets this criteria. The `exclusive-props-group-all-required` form is mapped as a `oneOf` array as follows:

```
{
     "oneOf": [
        { "required": ["<exclusive-prop-name-1>"] },
        { "required": ["<exclusive-prop-name-2>"] },
        ...
        ...
        { "required": ["<exclusive-prop-name-n>"] }
     ]
}
```

The `exclusive-prop-name(s)` are the mapped names for all members of the exclusive property group. The use of JSON schema's `oneOf` keyword is representative of the XOR and ensures that one and only one of the properties in the group is present in the instance data.

The `exclusive-props-group-with-optionals` mapping is the third and final form that can appear in the `allOf` array for the specialized mapping for required properties. This final form must be used if there exists one or more member properties of an exclusive property group that are optional (minimum cardinality = 0). There should be one element of this form for each distinct exclusive property group that meets this criteria. The `exclusive-props-group-with-optionals` mapping form itself is defined as a `oneOf` array with exactly two array elements.

The first array element is the XOR representation that ensures that one and only one property is represented from the exclusive property group. However, for exclusive property groups with property members that have cardinality equal to 0 the second array entry is required. This is necessary as in instance data the absence of all members of an exclusive property group is a valid possible scenario when at least one member of the group is optional:

```
{
   "oneOf": [
      {
         // This array entry ensures that one and only one property
         // is represented from the exclusive property group.
         "oneOf": [
            { "required": ["<exclusive-prop-name-1>"] },
            { "required": ["<exclusive-prop-name-2>"] },
            ...
            ...
            { "required": ["<exclusive-prop-name-n>"]
         ]
      },
      {
         // This second entry is needed for the scenario where no
         // properties from the exclusive property group are included
         // in instance data. This defines this scenario as valid.
         "allOf": [
            { "not": { "required": ["<exclusive-prop-name-1>"] },
            { "not": { "required": ["<exclusive-prop-name-2>"] },
            ...
            ...
            { "not": { "required": ["<exclusive-prop-name-n>"]
         ]
      }
   ]
}
```

The `exclusive-prop-name(s)` are the mapped names for all members of the exclusive property group.

Transitioning to a working example to illustrate these mappings in practice. For the example properties, we have an mRID that is required, an exclusive properties group ("group 1") with

three required properties, and an exclusive properties group (“group 2”) consisting of four properties; two of which are required and two of which are optional. Properties shown below have both a naming convention and are color coded to more easily identify group membership. Applying the described mapping results in the following:

```
{
   "$schema": "https://json-schema.org/draft/2020-12/schema",
   "title": "...",
   "description": "...",
   "namespace": "http://iec.ch/TC57/2020/XOR#",
   "type": "object",
   "additionalProperties": false,
   "properties": {
      "XOR": {
         "$ref": "#/$defs/ExampleXOR"
      }
   },
   "$defs": {
      // Class definition owning the exclusive property group
      "ExampleXOR": {
         "type": "object",
         "additionalProperties": false,
         "properties": {
            // Required
            "mRID": {
               "type": "string"
            },
            // Required
            "group1Property1": {
               "type": "string"
            },
            // Required
            "group1Property2": {
               "type": "string"
            },
            // Required
            "group1Property3": {
               "type": "string"
            },

            // Required
            "group2Property1": {
               "type": "string"
            },
            // Required
            "group2Property2": {
               "type": "array",
               "items": {
                  "type": "string"
               },
               "minItems": 1
            },
            // Optional
            "group2Property3": {
               "type": "string"
            },
            // Optional
            "group2Property4": {
               "type": "array",
               "items": {
                  "type": "string"
               },
```

```
                "minItems": 0
            }
        },
        // Specialized mapping for handling required properties...
        "allOf": [
            {
                // Mapping for all required properties that are not
                // members of an exclusive property group.
                "required": ["mRID"]
            },
            {
                // Mapping for exclusive property group 1. This
                // form is only applicable when all member properties
                // in a group are required.
                "oneOf": [
                    { "required": ["group1Property1"] },
                    { "required": ["group1Property2"] },
                    { "required": ["group1Property3"] }
                ]
            },
            {
                // Mapping for exclusive property group 2. This form is
                // needed as two of the member properties are optional.
                "oneOf": [
                    {
                        "oneOf": [
                            { "required": ["group2Property1"] },
                            { "required": ["group2Property2"] },
                            { "required": ["group2Property3"] },
                            { "required": ["group2Property4"] }
                        ]
                    },
                    {
                        "allOf": [
                            { "not": { "required": ["group2Property1"] } },
                            { "not": { "required": ["group2Property2"] } },
                            { "not": { "required": ["group2Property3"] } },
                            { "not": { "required": ["group2Property4"] }
                        ]
                    }
                ]
            }
        ]
    }
  }
}
```

5.14 Documentation

5.14.1 JSON schema documentation

In contrast to the more robust support for representing annotations and documentation in XSD schema, the JSON schema specification by design assumes a much more simplified approach via the `description` keyword. As outlined in the 2nd Edition of the ECMA-404 "JSON Data Interchange Syntax" standard, the content that can be represented within a JSON string and likewise applicable to the value for a JSON schema `description` keyword is prescribed as:

> A string is a sequence of Unicode code points wrapped with quotation marks (U+0022). All characters may be placed within the quotation marks except for the characters that must be escaped: quotation mark (U+0022),

reverse solidus (U+005C), and the control characters U+0000 to U+001F. There are two-character escape sequence representations of some characters.

Thus, the ECMA-404 standard suggests escape sequences for the well-defined codes from U+10000 through U+001F. Control character `\f` (U+000C) and `\n` (U+000A) represent the form feed and line feed characters respectively. Due to nuances in the JSON implementation approaches within various language implementations the processing of escape sequences may vary therefore introducing potential issues during processing.

Therefore this mapping specification prescribes the following approach to mapping the annotation and documentation sourced from the contextual model element or its related CIM element to a corresponding JSON schema `description` element:

- any documentation from the CIM UML should have each CRLF character removed and replaced with a single space with no paragraph breaks. This documentation should be inserted first within a JSON schema's `description` element.
- subsequently, any additional documentation from the contextual model should be processed in the same manner and then follow immediately after any CIM UML documentation in a JSON schema's `description` element.

5.14.2 General mapping

Each Documentation or Categorized Documentation item attached to a contextual model element or its related CIM element should be mapped to a `description` in the corresponding JSON schema definition:

```
{
    ...
    "$defs": {
        "<class-name>": {
            // Class level documentation...
            "description": "<class annotation>",
            "modelReference": "<class-ref>",
            "type": "object",
            "additionalProperties": false,
            "properties": {
                "<prop-name>": {
                    // Attribute/property level annotation...
                    "description": "<property annotation>
                    "modelReference": "<prop-ref>",
                            ...
                },
                ...
            }
        }
    }
}
```

5.15 Names

One objective of this mapping specification is to promote consistency in the results of its mapping of contextual model names with that outlined as part of the IEC 62361-100 “CIM profile to XML Schema mapping” standard.

Therefore the names of classes, properties, and values appearing in the contextual model are to be mapped to the JSON schema according to the same ruleset as specified in IEC 62361-100:

- All mapped names except enumeration values must conform with *Namespaces in XML 1.1* NCName syntax.
- Enumeration values must be mapped to values that conform with *XML1.1* attribute values syntax.

The mapped name is obtained by transliterating the contextual model name, character by character. Each character in the contextual model name that is valid according to the above rules is preserved otherwise it is replaced with an underscore "_".

5.16 Mapping order

5.16.1 General order

All the mapping is done so that all elements appear in a defined order:

- Root elements are ordered according to an alphabetical order
- Elements representing classes properties are ordered according to the following subclauses depending on whether there are or are not superclasses in the contextual model.

5.16.2 Mapping order when there is no superclass

When there is no superclass, elements representing properties are ordered as follows:

- mRID simple property first,
- then other simple properties in alphabetical order,
- then object and compound properties in alphabetical order.

Note: the mapping form of a Union object property (see clause 5.12.4) results in a mapping that includes a distinct property corresponding to each possible subclass of the Union object property. Each of these property names is determined by the application of a change name rules as outlined in Annex D. The resulting names are to appear in the alphabetically ordered object and compound properties list are ordered alphabetically.

5.16.3 Mapping order when there is a superclass

As previously highlighted, the JSON schema specification does not support direct extension capabilities as commonly understood in other schema specifications (e.g. XSD); therefore, when a superclass exists in the contextual model (that, in turn, could potentially also have a superclass), then the JSON `properties` representing the superclass properties are ordered as described in the previous subclause. The element representing sub classes properties are ordered as follows:

- First elements representing superclass properties in the alphabetical order as defined in the previous subclause,
- Then elements representing native properties of the subclass in the same alphabetical order as defined in the previous subclause.

5.16.4 Mapping order examples

In CIM, "MktOrganisation" is inheriting from "Organisation" that inherits from "IdentifiedObject". In the contextual model, we could have two cases:

- "MktOrganisation" **structured class** stands alone and could have as native **simple propertie**s whose CIM counterparts (attributes) are in "IdentifiedObject", for example "mRID" and "name".
- "MktOrganisation" **structured class** inherits for example from an "IdentifiedObject" **super structured class** that has "mRID" and "name" **simple properties**.

If in the contextual model "MktOrganisation" stands alone and has "mRID", "name", "creditFlag" and "lastModified" **simple properties**, the mapping order will be as follows:

- mRID
- creditFlag
- lastModified
- name

If contextual model “MktOrganisation” inherits from “IdentifiedObject”. “IdentifiedObject” has “mRID” and “name” **simple properties** and “MktOrganisation” has “creditFlag” and “lastModified” **simple properties**. The mapping order will be as follows:

- mRID
- name
- creditFlag
- lastModified

5.17 Changing name rules

The rule to change contextual model object property names to schema mapped names could be used to avoid duplication of property name in a `properties` set. This could occur when the contextual model object property:

- is marked as a “union” and its referent is a contextual model super class,
- or is a subset of a CIM association with a CIM super class that is used in the contextual model with a referent that is a subset of a sub class of this CIM super class.

So, in both cases, the contextual model object property name matches a CIM association end role name whose referent is a CIM super class.

Contextual model object property name could be of three kinds:

1) same as the one of the super class name,
2) same as the super class name prefixed by a qualifier with or without an underscore,
3) or not the same as the super class name.

The changing name rule would be as follows for a union object property:

- If the names are the same, the mapped name will be the contextual model sub class name,
- If the names differ just with a qualifier, the mapped name will be the contextual model sub class name prefixed by the same qualifier and the underscore,
- If the names are different, the mapped name will be the contextual model sub class name prefixed by the object property name and an underscore.

The changing name rule would be as follows in the second case:

- If the object property name matches the CIM super class name, the mapped name will be the contextual model sub class name,
- If the object property name matches the CIM super class name plus a qualifier (and an underscore), the mapped name will be the contextual model sub class name prefixed by the same qualifier and the underscore,
- If the names are different, the mapped name will be the contextual model sub class name prefixed by the object property name and an underscore.

Annex A
(Informative)

Introduction to JSON

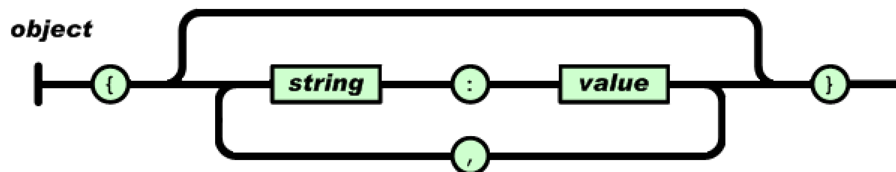
JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures:

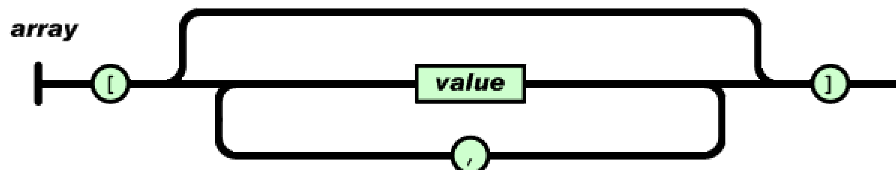
- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

In JSON, they take on these forms:

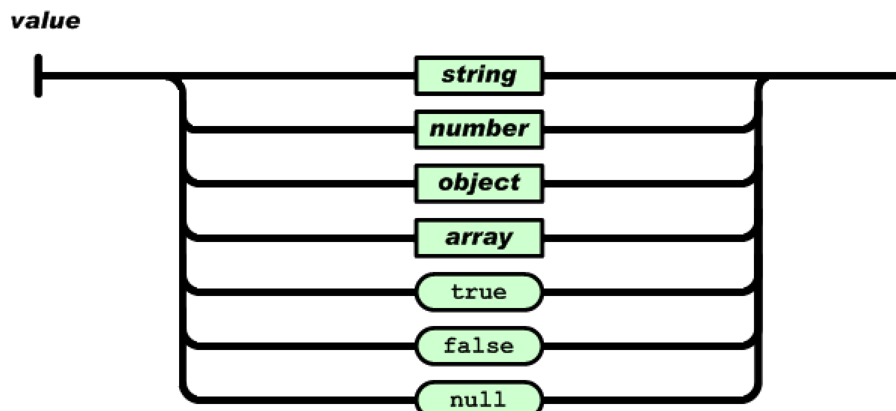
An object is an unordered set of name/value pairs. An object begins with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (comma).



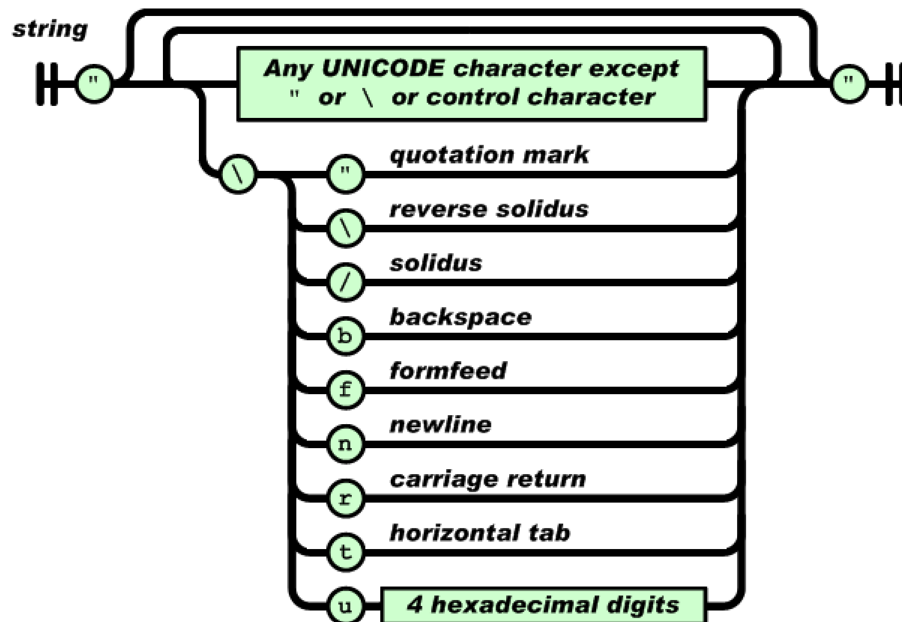
An array is an ordered collection of values. An array begins with [(left bracket) and ends with] (right bracket). Values are separated by , (comma).



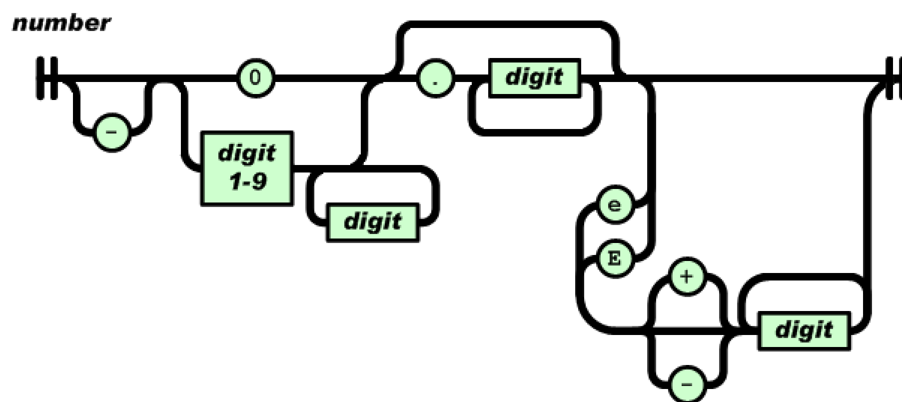
A value can be a string in double quotes, or a number, or true or false or null, or an object or an array. These structures can be nested.



A string is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. A character is represented as a single character string. A string is very much like a C or Java string.



A number is very much like a C or Java number, except that the octal and hexadecimal formats are not used.



Whitespace can be inserted between any pair of tokens. Excepting a few encoding details, this completely describes the language.

Annex B
(Informative)

Contextual model representation

The mapping definitions apply to a contextual model which must exist in some form before a JSON schema can be constructed. The representation of a contextual model and the rules for constructing it are outside the scope of this specification and the following information is not normative.

Two contextual model representations are presently in use: Web Ontology Language (OWL) and Unified Modeling Language (UML). The contextual model concepts used in this specification could be represented in UML and OWL as defined in Table B1

Concept	UML Definition	OWL Definition
Structured class	Class	owl:Class
Concrete class	Class	owl:Class with concrete annotation
Abstract class	Class with “Abstract” qualifier	owl:Class
Root class	Class with “Root” qualifier	owl:Class
Union class	Class with a stereotype to be defined (“Union” for example)	owl:Class with owl:unionOf
Compound class	Class with “Compound” stereotype	owl:Class with compound annotation
Object property	Association end	owl:ObjectProperty with zero or more owl:Restriction
By-reference object property	Association end whose end type is designated by its reference	owl:ObjectProperty with zero or more owl:Restriction and byReference annotation
Union property	Association end with a stereotype to be defined (“Union” for example)	
Compound property	Class attribute whose type is a compound class	
Exclusive property group	Associations with an XOR constraint	
Simple property	Class attribute	owl:DatatypeProperty with zero or more owl:Restriction
BasicType	Datatype with “Primitive” stereotype	XML schema datatype
Simple type	Datatype with “CIMDatatype” stereotype that has only a value attribute and constraints expressing restrictions on the type of the value attribute	rdfs:Datatype
Datatype	Datatype with “CIMDatatype” stereotype and optional constraints	rdfs:Datatype with quantity annotation
Enumeration class	Class with “enumeration” stereotype	owl:Class with owl:oneOf
CodeList	Class with a stereotype to be defined (“CodeList” for example)	owl:Class

Concept	UML Definition	OWL Definition
Subclass/Superclass relationship	Generalized By (inherits)	owl:SubClassOf
Documentation	Notes	rdfs:comment
Categorized documentation	TaggedValue, stereotype, sonstraint	rdfs:comment

Table B.1 – Contextual model representation

Annex C
(Informative)

XSD to JSON schema type and facet mappings

It can be helpful for those familiar with the IEC 62361-100 NDR publication and XSD schema to understand how XSD types and facets may correlate to JSON schema counterparts. The following table provides such a mapping for those types and facets relevant to this document. Where no corresponding equivalent exists from XSD to JSON schema it is noted in the table.

Basic Type XML	XML Facet	JSON Type	JSON Schema Facet
Boolean	*No facet*	*boolean*	*No facet*
String *For named string use pattern:* *Normalized String* *Token String* *NMToken String* *Name String* *NCName String* *AnyURI String*	length	string	minLength, maxLength
	minLength		minLength
	maxLength		maxLength
	pattern		pattern
	whiteSpace		pattern
	enumeration		enum
Integer	totalDigits	integer *see use of MultipleOf*	minimum, maximum
	minInclusive		minimum
	maxInclusive		maximum
	minExclusive		exclusiveMinimum
	maxExclusive		exclusiveMaximum
	enumeration		enum
Float Double	minInclusive	number	minimum
	maxInclusive		maximum
	minExclusive		exclusiveMinimum
	maxExclusive		exclusiveMaximum
	enumeration		enum
	pattern		Unsupported
Decimal	totalDigits	number	Unsupported
	fractionalDigits		Unsupported
	minInclusive		minimum
	maxInclusive		maximum
	minExclusive		exclusiveMinimum
	maxExclusive		exclusiveMaximum
	enumeration		enum
	pattern		Unsupported
date	minInclusive	string *JSON string with pattern*	Unsupported
	maxInclusive		Unsupported
	minExclusive		Unsupported
	maxExclusive		Unsupported
	whitespace		Unsupported
	enumeration		Unsupported

<table>
<tr><td></td><td>pattern</td><td colspan="2">pattern

^(-?([1-9][0-9]{3,}|0[0-9]{3})-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])(Z|(\\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?)$</td></tr>
<tr><td rowspan="7">dateTime</td><td>minInclusive</td><td rowspan="6">string

JSON string with pattern</td><td>Unsupported</td></tr>
<tr><td>maxInclusive</td><td>Unsupported</td></tr>
<tr><td>minExclusive</td><td>Unsupported</td></tr>
<tr><td>maxExclusive</td><td>Unsupported</td></tr>
<tr><td>whitespace</td><td>Unsupported</td></tr>
<tr><td>enumeration</td><td>Unsupported</td></tr>
<tr><td>pattern</td><td colspan="2">pattern

^(-?([1-9][0-9]{3,}|0[0-9]{3})-(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](\\.[0-9]+)?|(24:00:00(\\.0+)?))(Z|(\\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?)$</td></tr>
<tr><td rowspan="7">time</td><td>minInclusive</td><td rowspan="6">string

JSON string with pattern</td><td>Unsupported</td></tr>
<tr><td>maxInclusive</td><td>Unsupported</td></tr>
<tr><td>minExclusive</td><td>Unsupported</td></tr>
<tr><td>maxExclusive</td><td>Unsupported</td></tr>
<tr><td>whitespace</td><td>Unsupported</td></tr>
<tr><td>enumeration</td><td>Unsupported</td></tr>
<tr><td>pattern</td><td colspan="2">pattern

^((([01][0-9]|2[0-3]):[0-5][0-9]:[0-5][0-9](\\.[0-9]+)?|(24:00:00(\\.0+)?))(Z|(\\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?)$</td></tr>
<tr><td rowspan="7">gMonthDay</td><td>minInclusive</td><td rowspan="6">string

JSON string with pattern</td><td>Unsupported</td></tr>
<tr><td>maxInclusive</td><td>Unsupported</td></tr>
<tr><td>minExclusive</td><td>Unsupported</td></tr>
<tr><td>maxExclusive</td><td>Unsupported</td></tr>
<tr><td>whitespace</td><td>Unsupported</td></tr>
<tr><td>enumeration</td><td>Unsupported</td></tr>
<tr><td>pattern</td><td colspan="2">pattern

^(--(((02)-(0[1-9]|[1][0-9]|2[0-9]))|((0[4689]|(11))-(0[1-9]|[1][0-9]|2[0-9]|(30)))|((0[13578]|(1[02]))-(0[1-9]|[1][0-9]|2[0-9]|(3[0-1])))))$</td></tr>
<tr><td rowspan="7">duration</td><td>minInclusive</td><td rowspan="6">string

JSON string with pattern</td><td>Unsupported</td></tr>
<tr><td>maxInclusive</td><td>Unsupported</td></tr>
<tr><td>minExclusive</td><td>Unsupported</td></tr>
<tr><td>maxExclusive</td><td>Unsupported</td></tr>
<tr><td>whitespace</td><td>Unsupported</td></tr>
<tr><td>enumeration</td><td>Unsupported</td></tr>
<tr><td>pattern</td><td colspan="2">pattern

^(-?)P(?=\\d|T\\d)(?:(\\d+)Y)?(?:(\\d+)M)?(?:(\\d+)([DW]))?(?:T(?:(\\d+)H)?(?:(\\d+)M)?(?:(\\d+(?:\\.\\d+)?)S)?)?$</td></tr>
</table>

Table C.1 – XSD to JSON types and facet mappings

Annex D
(Informative)

Changing name rules examples

D.1 Changing name rule context

The rule to change contextual model object property names to schema mapped names could be used to avoid duplication of element name in a sequence or a choice schema design. This could occur when the contextual model object property:

- is marked as a “union” and its referent is a contextual model super class,
- or is a subset of a CIM association with a CIM super class that is used in the contextual model with a referent that is a subset of a subclass of this CIM super class.

So, in both cases, the contextual model object property name matches a CIM association end role name whose referent is a CIM super class.

The examples are using UML and JSON schema to illustrate the problem and the changing name rules.

D.2 Changing name rule when CIM association end role name and CIM super class name are the same

Changing name rule could occur when in the information model there is a class that has an association with a super class and the association end role name is the same than the super class name. In Figure D.1, the association end role name is "SuperClass" and SuperClass name is also "SuperClass":

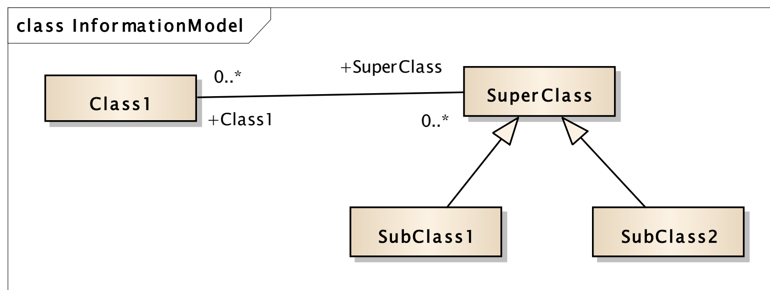


Figure D.1 – Example of end role name matching super class name

In this case, at contextual model level, the super class could be used and marked as a "Union" (meaning that subclasses are selected), see Figure D.2

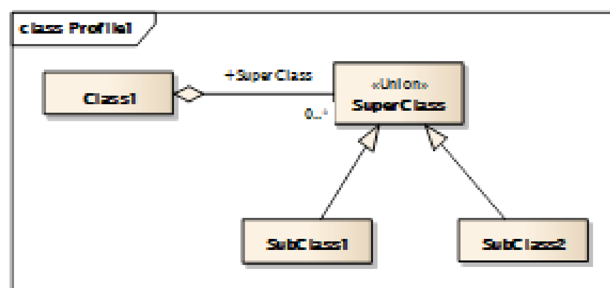


Figure D.2 – Contextual model end role name matching super class name

When mapping to JSON schema according to straight IEC 62361-104, the result will be:

```
{
     ...
     "$defs": {
          "Class": {
               ...
               "type": "object",
               "additionalProperties": false,
               "properties": {
                    // A union object property with max cardinality > 1
                    // is defined as an array within the properties
                    // element of a mapped JSON subschema.
                    "SuperClass": {
                         "description": "<property annotation>",
                         "modelReference": "<prop-ref>",
                         "type": "array",
                         "items": {
                              "$ref": "#/$defs/SubClass1"
                         },
                         "minItems": 0
                    },
                    "SuperClass": {
                         "description": "<property annotation>",
                         "modelReference": "<prop-ref>",
                         "type": "array",
                         "items": {
                              "$ref": "#/$defs/SubClass2"
                         },
                         "minItems": 0
                    },
               }
          }
     }
}
```

As, in a choice, JSON properties can not have the same name but different types, we have to apply the changing name rule, which in this case says that the element name will be the subclass name:

```
{
     ...
     "$defs": {
          "Class": {
               ...
               "type": "object",
               "additionalProperties": false,
               "properties": {
                    "SubClass1": {
                         "description": "<property annotation>",
                         "modelReference": "<prop-ref>",
                         "type": "array",
                         "items": {
                              "$ref": "#/$defs/SubClass1"
                         },
                         "minItems": 0
                    },
                    "SubClass2": {
```

```
                        "description": "<property annotation>",
                        "modelReference": "<prop-ref>",
                        "type": "array",
                        "items": {
                            "$ref": "#/$defs/SubClass2"
                        },
                        "minItems": 0
                    },
                }
            }
        }
}
```

D.3 Changing name rule when CIM association end rolename is the CIM super class name prefixed by a qualifier followed by an underscore

Changing name rule could occur when in the information model there is a class that has an association with a super class and the association end role name is the same than the super class name prefixed by a qualifier followed by an underscore. In Figure D.3, SuperClass name is "SuperClass" and association end role name is "Qualifier_SuperClass":

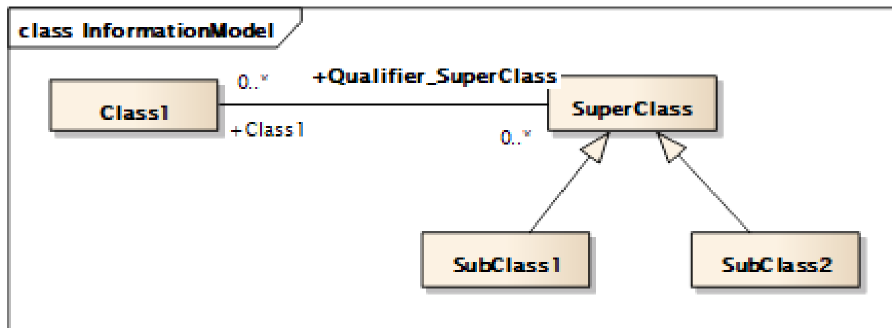


Figure D.3 – Example of end role name with a qualifier

In this case, at contextual model level, the super class could be used and marked as a "Union" (meaning that subclasses are selected), see Figure D.4:

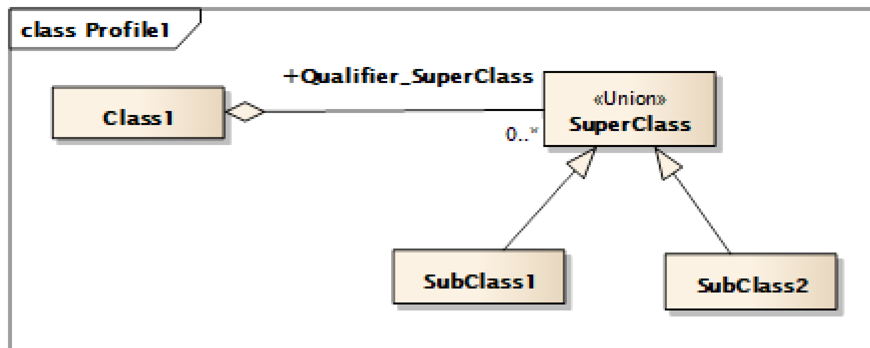


Figure D.4 – Contextual model end role name with a qualifier

When mapping to JSON schema according to straight IEC 62361-104, the result will be:

```
{
    ...
```

```
    "$defs": {
        "Class": {
            ...
            "type": "object",
            "additionalProperties": false,
            "properties": {
                "Qualifier_SuperClass": {
                    "description": "<property annotation>",
                    "modelReference": "<prop-ref>",
                    "type": "array",
                    "items": {
                        "$ref": "#/$defs/SubClass1"
                    },
                    "minItems": 0
                },
                "Qualifier_ SuperClass": {
                    "description": "<property annotation>",
                    "modelReference": "<prop-ref>",
                    "type": "array",
                    "items": {
                        "$ref": "#/$defs/SubClass2"
                    },
                    "minItems": 0
                },
            }
        }
    }
}
```

As, in a choice, JSON properties can not have the same name but different types, we have to apply the changing name rule, which in this case says that the element name will be the subclass name prefixed by the qualifier followed by an underscore:

```
{
    ...
    "$defs": {
        "Class": {
            ...
            "type": "object",
            "additionalProperties": false,
            "properties": {
                "Qualifier_SubClass1": {
                    "description": "<property annotation>",
                    "modelReference": "<prop-ref>",
                    "type": "array",
                    "items": {
                        "$ref": "#/$defs/SubClass1"
                    },
                    "minItems": 0
                },
                "Qualifier_SubClass2": {
                    "description": "<property annotation>",
                    "modelReference": "<prop-ref>",
                    "type": "array",
                    "items": {
                        "$ref": "#/$defs/SubClass2"
                    },
                    "minItems": 0
                },
            }
        }
```

```
        }
}
```

D.4 Changing name rule when CIM association end role name and the CIM super class name are completely different

Changing name rule could occur when in the information model there is a class that has an association with a super class and the association end role name is different than the super class name. In Figure D.5, SuperClass name is "SuperClass" and association end role name is "EndRoleName":

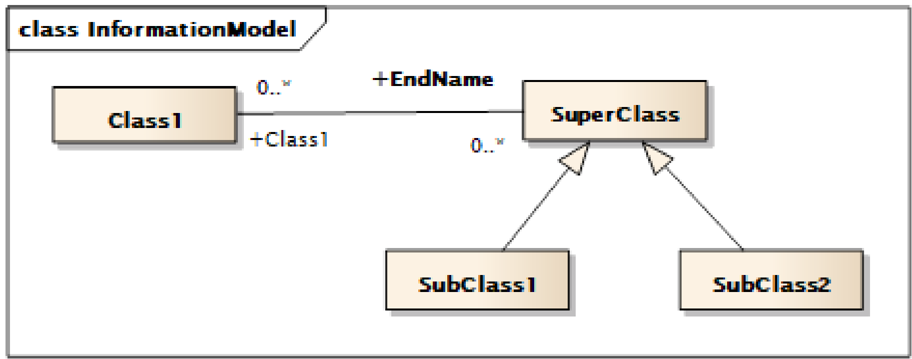


Figure D.5 – End role name and super class name different

In this case, at contextual model level, the super class could be used and marked as a "Union" (meaning that subclasses are selected), see Figure D.6.

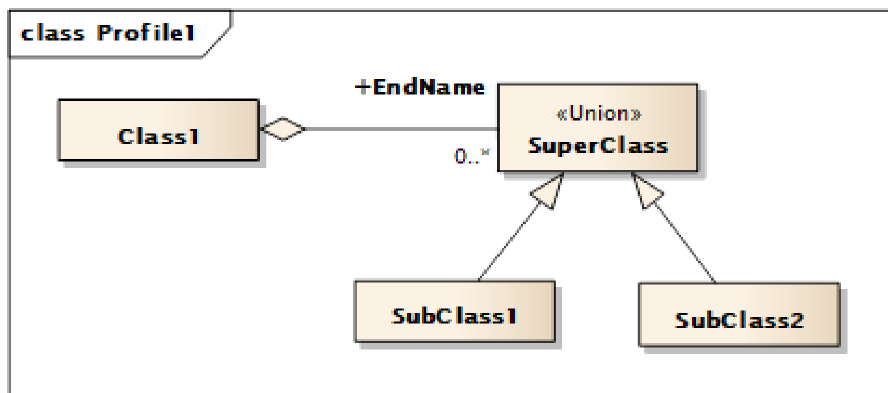


Figure D.6 – Contextual model with end role name different from super class name

```
{
     ...
     "$defs": {
          "Class": {
               ...
               "type": "object",
               "additionalProperties": false,
               "properties": {
```

```
                    "EndRoleName": {
                         "description": "<property annotation>",
                         "modelReference": "<prop-ref>",
                         "type": "array",
                         "items": {
                              "$ref": "#/$defs/SubClass1"
                         },
                         "minItems": 0
                    },
                    "EndRoleName": {
                         "description": "<property annotation>",
                         "modelReference": "<prop-ref>",
                         "type": "array",
                         "items": {
                              "$ref": "#/$defs/SubClass2"
                         },
                         "minItems": 0
                    },
               }
          }
     }
}
```

As, in a choice, JSON properties can not have the same name but different types, we have to apply the changing name rule, which in this case says that the element name will be the subclass name prefixed by the association end role name followed by an underscore:

```
{
     ...
     "$defs": {
          "Class": {
               ...
               "type": "object",
               "additionalProperties": false,
               "properties": {
                    "EndRoleName_SubClass1": {
                         "description": "<property annotation>",
                         "modelReference": "<prop-ref>",
                         "type": "array",
                         "items": {
                              "$ref": "#/$defs/SubClass1"
                         },
                         "minItems": 0
                    },
                    "EndRoleName_SubClass2": {
                         "description": "<property annotation>",
                         "modelReference": "<prop-ref>",
                         "type": "array",
                         "items": {
                              "$ref": "#/$defs/SubClass2"
                         },
                         "minItems": 0
                    },
               }
          }
     }
}
```

Annex E
(Informative)

An overview of JSON schema's $id and $ref kewords

This annex serves to provide an extensive informative overview of how the `$id` and `$ref` keywords function within JSON schemas and correspondingly their use in IEC profiles adhering to this specification. It will also provide clarity on common misunderstandings as to how `$ref` references are resolved relative to the `$id` of the schemas and/or subschemas that they may reference. Finally, an explanation of the differences that may be observed when resolving these references in products across different vendors is covered.

E.1 Preliminary Background

To facilitate an overview of this topic, a revisit of common JSON schema terminology as well as the purpose of the `$id` and `$ref` keywords within JSON schema is necessary.

E.1.1 JSON Schema and JSON subschemas

In the clauses that follow the term "JSON schema" implies a single standalone file representing a profile mapping from a given contextual model as defined within this specification. Such profiles are considered self-contained JSON schema with no `$ref` JSON Pointers to external schemas. In this context JSON Pointers only reference JSON subschemas (i.e. "internal subschemas") within the same standalone JSON schema profile. "Subschemas" are then understood to be any JSON object schema definition that appears within the `$defs` element in the standalone profile. This mapping specification document takes this approach as it eliminates the potential complexities that accompany references to external schemas.

An alternate approach is to define a set of standalone JSON schemas that as a collective represent a set of JSON schemas needed to perform validation. In this approach, one schema from the set would be understood to be the "primary" or "root" schema and would have defined in its `$defs` element relevant internal subschemas. These internal subschemas would then utilize `$ref` JSON Pointers to either other internal subschemas or to the subschemas defined in another external standalone schema in the set. Organizing or decomposing JSON schemas in this manner is often done to promote reuse of common JSON subschema definitions.

In this Annex, a final working example for CodeLists that uses this alternate approach will be used to help illustrate how `$id` and `$ref` keywords function in that context.

E.1.2 Overview of URIs

Given that JSON schema's `$id` and `$ref` keywords leverage URIs it is beneficial to formally define them. The terms URI (Universal Resource Identifier) and URL (Universal Resource Locator) are often used interchangeably; however, they are not identical. For a JSON schema a URI is a string of characters used as an identifier for a specific resource.

A URL is a special type of URI identifier that may also indicate how to locate or access the resource. Such URLs include the protocol or scheme (e.g. HTTP(s)) used to obtain the resource. In this case it is referred to as a URL even though it is also a URI.

A URN (Uniform Resource Name) is a URI that uses the urn scheme. URNs are globally unique persistent identifiers assigned within defined namespaces. URNs cannot be used to directly locate an item or resource and they are typically not directly resolvable without an intermediary to determine a corresponding URL.

Figure E.1 is provided to aid in better understanding the relationships between URIs, URNs, and URLs and their associated value spaces. The values indicated in the diagram are simply examples for the sake of illustration.

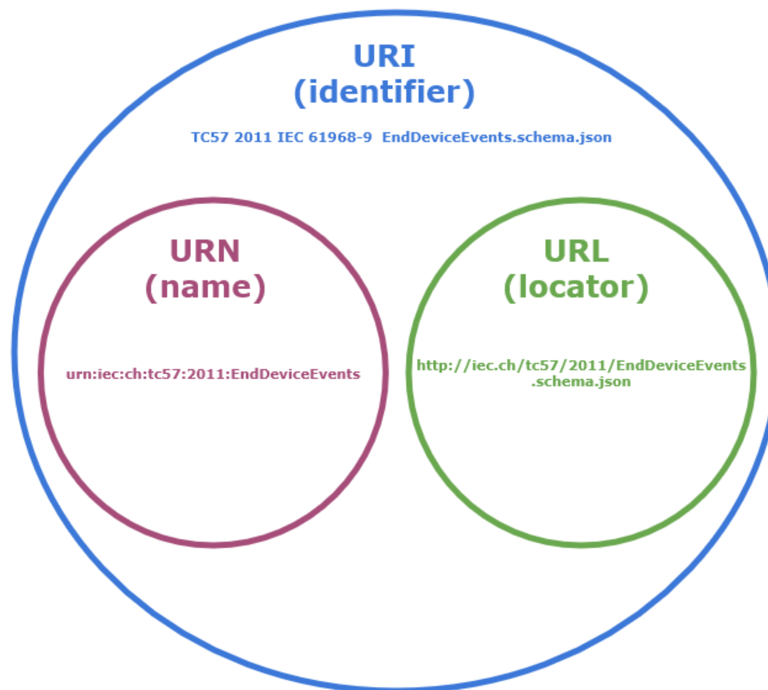


Figure E.1 – Diagram of the relationships between URIs, URNs, and URLs

E.1.3 The JSON schema $id keyword

To facilitate an understanding of the JSON schema's $id keyword relevant aspects prescribed in the specification should be highlighted:

- An `$id` keyword within a JSON schema is not strictly required by the specification though it is recommended.
- When present, the value must be a string and must represent a valid URI-reference. This URI-reference should be normalized and must result in an absolute-URI (without a fragment).
- If present, the `$id` keyword is understood to identify a schema resource with its canonical URI. The URI is an identifier and not necessarily a network locator (URL).
- In the case of an `$id` expressed as a network-addressable URL, a schema need not be downloadable from the URL.
- This URI also serves as the base URI for relative URI-references in keywords within the schema resource, in accordance with RFC 3986 regarding base URIs embedded in content.
- The presence of `$id` in a subschema indicates that the subschema constitutes a distinct schema resource within a single schema document. For a subchema `$id` that is a relative URI-reference, the base URI for resolving that reference is the URI of the parent schema resource. If no parent schema object explicitly identifies itself as a resource with `$id`, the base URI is that of the entire document.

To summarize the key points, the `$id` keyword's value is a URI-reference that serves two important purposes:

- It declares a unique identifier for a schema.
- It declares a base URI against which `$ref` URI-references are resolved.

As also highlighted, an `$id` can also be specified as part of the definition of a subschema. When specified, it provides an additional way to refer to the subschema without using JSON Pointer. This means you can refer to a subschema by a unique name, rather than by where it appears in the JSON tree.

The following schema will be used as the basis for providing examples for comparison of these two types of `$ref` styles. Specific elements in the schema are color coded to facilitate an understanding of how the two reference types are derived.

```
{
      "$id": "http://iec.ch/TC57/2020/Substations.schema.json",
      "$schema": "https://json-schema.org/draft/2020-12/schema",
      "title": "Substations",
         "Substations": {
            "type": "array",
            "items": {
               "$ref": "<Reference>"
            }
         },
      "$defs": {
          "Substation": {
              "$id": "#Substation",
              "title": "Substation",
              "description": "Detailed description of a Subtation...",
              "modelReference": "http://iec.ch/TC57/2020/CIM-schema-
cim18#Substation",
              "type": "object",
              "additionalProperties": false,
              "properties": {
                   ...
              }
         }
      }
}
```

Using the JSON Pointer format for the value of the `$ref` results in the following internal reference:

```
{
   "$ref": "#/$defs/Substation"
}
```

Note that the color blue correlates to how the reference is resolved. In this first example, it is relevant to note that for the JSON subschema definition the `$id` keyword, though included in the example, could be removed as it is not needed nor used by the JSON Pointer reference.

Using the unique name of the `Substation` subschema example as defined in its `$id` results in the following alternate value for the `$ref` to the subschema:

```
{
   "$ref": "#Substation"
}
```

In this instance the color red emphasizes that the reference is resolved against the unique name value specified by the subschema's `$id`.

Though technically both `$ref` formats should be valid, in practice vendor implementations are inconsistent in support for resolving of the second. Both have been included here, however, for the sake of thoroughness. For the purposes of this mapping specification only JSON Pointers are utilized in `$ref` values as it is universally supported across implementations.

E.1.4 The JSON schema $ref keyword and base URIs

To understand the use of JSON schema's `$ref` keyword it is important to understand how a base URI is established for a JSON schema as defined by the draft-07 and 2020-12 specifications . This is necessary as any URIs declared by the `$ref` keyword in the schema will be resolved relative to the schema's base URI.

For a standalone self-contained schema this is straightforward as all `$ref` are resolvable to subschemas contained in the same schema and are expressed as JSON Pointers. The base URI of these `$ref` (s) is that defined in the `$id` of the standalone schema itself and therefore expressed as a JSON Pointer `$ref` (s).

This becomes potentially more complicated when resolving `$ref` pointers for a set of distinct JSON schemas (the alternate approach) that have a designated primary schema containing external references to others in the set. This complexity is due to the difference in vendor implementations for the tooling.

To summarize and highlight the relevant aspects prescribed in the JSON specification for the `$ref` keyword:

- The specifications define the `$ref` keyword which can be used to reference a schema to be applied to the current instance location.
- The value of the `$ref` keyword must be a string which is a URI-reference. Resolved against the current URI base, it produces the URI of the schema to apply. This resolution is safe to perform on schema load, as the process of evaluating an instance cannot change how the reference resolves.
- As the value of `$ref` is a URI-Reference, it allows the possibility to externalise or divide a schema across multiple files, and provides the ability to validate recursive structures through self-reference.
- The resolved URI produced by the `$ref` keyword is not necessarily a network locator, only an identifier. A schema need not be downloadable from the address if it is a network-addressable URL, and implementations should not assume they should perform a network operation when they encounter a network-addressable URI.

Two examples of URI reference approaches for a `$ref` appear next.

First, the following is an example illustrating a relative JSON Pointer for a `$ref`. The JSON Pointer references a subschema in the same document and the base URI is implied to be that specified in the `$id` of the document itself:

```
{
   "$ref": "#/$defs/EndDeviceEvent"
}
```

Second, to illustrate an example of the use of an absolute-URI results in the reference shown in the next schema extract:

```
{
   "$ref": "http://iec.ch/TC57/2020/codelists.schema.json#/$defs/CodeList"
}
```

For the above `$ref` absolute-URI, when broken down and analyzed what is observed is that the JSON Pointer is referencing an external JSON schema. The `$id` of that schema is the exact value that appears before the fragment ('#') within the `$ref` absolute-URI. The specified fragment in the absolute-URI (i.e. `#/$defs/CodeList`) indicates that the `$ref` is referencing the CodeList subschema within the `$defs` element of that external JSON schema:

```
{
      "$id": "http://iec.ch/TC57/2020/codelists.schema.json",
      "$schema": "https://json-schema.org/draft/2020-12/schema",
      "title": "CodeList",
      "description": "",
      "namespace": "http://iec.ch/TC57/2021/CodeList#",
      "type": "object",
      "additionalProperties": false,
      "properties": {
          ...
      },
      "$defs": {
          "CodeList": {
              "description": "..."
              "type": "string",
              "enum": [
                  ...
              ]
          },
          ...
      }
}
```

Per the JSON schema specification, the URI that appears in both the absolute `$ref` and in the `$id` (`http://iec.ch/TC57/2020/codelists.schema.json`) of this example need not be network resolvable. This means that if an end user of a set of schema profiles knows that the `$id` and `$ref(s)` in the schemas are network addressable and wants to take advantage of that fact they must ensure that the JSON schema tooling or library implementation they choose supports this feature.

E.2 Absolute-URI $refs, URI $refs resolution, and vendor implementations

Within this clause the application and resolution of $ref JSON Pointers to external schemas across vendor implementations is covered. This is done through a description of an implementation approach for a potential real world requirement for extending codelists (see clause 5.9). It was chosen for its practical benefit in illustrating both `$ref` external references as well as local codelist extensions using the multiple standalone schemas approach to promote subschema resuse and ease of maintenance.

The first subclause provides a generalized "template" for the approach with the subsequent subclause providing a working example applied in practice.

E.2.1 Externalizing codelists to support local codelist extensions

As normatively prescribed in the CodeList classes clause 5.9 in this document, each codelist class or enumeration defined in the contextual model is mapped to a `codelist-type` JSON subschema definition as follows:

```
{
     ...
     "$defs": {
          // Each codelist subschema definition must appear
          // within the $defs element of the JSON schema
          // profile.
```

```
          "<codelist-name>": {
               "description": "<annotation>",
               "modelReference": "<codelist-ref>",
               "type": "<codelist-type>",
               "enum": [
                    Content: (codelist-value*)
               ]
          }
          ...
     }
}
```

The subchema for the codelist class can then be referenced elsewhere in the standalone schema profile using the internal JSON Pointer:

```
{
   "$ref": "#/$defs/<codelist-name>"
}
```

This serves as the starting point in understanding one possible approach to externalizing a codelist in order to allow for local extensions to the codelist.

The first step would be to convert all such internal JSON Pointers to external JSON Pointers pointing to a new external secondary standalone schema. The outcome of converting the example internal JSON Pointer just shown to one that references a new standalone schema would be:

```
{
   "$ref": "https://iec.ch/TC57/<year>/<codelist-
name>.schema.json#/$defs/<codelist-name>"
}
```

With the converted `$ref` now referring to the `$id` defined within the new standalone JSON schema:

```
{
   "$id": "https://iec.ch/TC57/<year>/<codelist-name>.schema.json"
}
```

The original codelist subschema definition is then migrated from the primary schema into the new standalone schema and renamed to `Standard<codelist-name>`. The prefixing of the codelist in this manner indicates that the subschema is the standard set of codelists as originally defined in the canonical model.

A new definition for `<codelist-name>` must then be defined in this secondary schema to fulfil the absolute-URI of the external JSON Pointers now used in the primary schema. This new definition is then defined to be the union of the standard set of codelist values and the set of local codelist extensions that are needed. In JSON schema the union is achieved via the use of the `oneOf` keyword which declares that a particular codelist value is valid if it is within the value space of either the standard enumeration or the local extensions enumeration.

This results in the following informative mapping for the new secondary schema:

```
{
     "$id": "https://iec.ch/TC57/<year>/<codelist-name>.schema.json",
     "$schema": "https://json-schema.org/draft/2020-12/schema",
     "title": "<title for the profile>",
     "description": "<annotation for the profile>",
     "namespace": "<namespace-uri>",
     "type": "object",
     "additionalProperties": false,
```

```
      "$defs": {
          // Each codelist mapping now is comprised of two components.
          // A migrated and renamed definition of the standard codelist
          // and a new definition for the codelist that is a union…
          "Standard<codelist-name>": {
               "description": "<annotation>",
               "modelReference": "<codelist-ref>",
               "type": "string",
               "enum": [
                    Content: (codelist-values*)
               ]
          },
          "<codelist-name>": {
               "oneOf": [
                    {
                         "$ref": "#/$defs/Standard<codelist-name>"
                    },
                    {
                         // Here a JSON Pointer to an external JSON schema
                         // definitions file is referenced. The JSON Pointer
                         // further references the Local<codelist-name>
                         // enum definition in that file that defines the
                         // set of local codelist extensions...
                         "$ref": "https://iec.ch/TC57/<year>/Local<codelist-
name>.schema.json#/$defs/Local<codelist-name>"
                    }
               ]
          }
          ...
      }
}
```

At this point it should be obvious that a one-to-one correlation of an external standalone schema file per codelist class is implied. This is sufficient for a handful of codelists that may need local extensions. However, if many such extensions are needed then a better suggested approach is to manage them all within a single secondary external schema with all codelists mapped within that file as just described. When this approach is used then all of the external JSON Pointer references in the primary schema should be updated to reference the single secondary schema.

In the above informative mapping for the secondary external standalone schema the JSON Pointer below is introduced that then points to a final external JSON schema containing the local extensions:

```
{
   "$ref": "https://iec.ch/TC57/<year>/Local<codelist-
name>.schema.json#/$defs/Local<codelist-name>"
}
```

This reduces maintenance and isolates local changes to just an update or replacement of the JSON schema containing the definition of the local extensions. If many local extensions are needed and a single secondary standalone schema is used it does not immediately imply that a single standlone schema should also be used for the corresponding local extensions. It may be logical to do so depending on the rate of change of certain codelists or it may be more beneficial to keep a separate local extensions schema for each codelist.

Completing this externalized codelist mapping, the final informative standalone schema for local extensions would appear as follows:

```
{
      "$id": "https://iec.ch/TC57/<year>/Local<codelist-name>.schema.json",
      "$schema": "https://json-schema.org/draft/2020-12/schema",
      "title": "<Title for the local codelists>",
```

```
      "description": "<annotation for local codelists extensions file>",
      "namespace": "<namespace-uri>",
      "$defs": {
         // Local set of extensions to the codelist
         "Local<codelist-name>": {
              "description": "<annotation>",
              "type": "string",
              "enum": [
                   Content: (local-codelist-value*)
              ]
         }
         ...
      }
}
```

The informative naming convention for local codelists is the `codelist-name` prefixed with `Local` as shown.

The `local-codelist-value` would correlate to one or more local extensions for code values needed that do not exist in the standard codelist class or enumeration.

E.2.2 An applied example of externalization of local codelist extensions

An applied example of externalized local codelist extensions is now described to illustrate issues in resolving `$id` and `$ref` pointers in different vendor implementations.

The following consists of three distinct standalone JSON schemas. The first is a primary schema produced according to this mapping specification but with its codelists externalized as described in the previous clause in this annex. Note that elements within the examples are color coded to facilitate an easier understanding as to how base URI and JSON Pointers are resolved across schemas.

The primary schema appears as follows:

```
{
      "$id": "http://iec.ch/TC57/2020/AcknowledgementDocument.schema.json",
      "$schema": "https://json-schema.org/draft/2020-12/schema",
      "title": "AcknowledgementDocument.schema.json",
      "description": "A profile that represents an acknowledgment receipt
of an an electronic document.",
      "namespace": "http://iec.ch/TC57/2018/AcknowledgementDocument#",
      "type": "object",
      "additionalProperties": false,
      "properties": {
         "Acknowledgement_MarketDocument": {
            "$ref": "#/$defs/Acknowledgement_MarketDocument"
         }
      },
      "$defs": {
         "Acknowledgement_MarketDocument": {
            "description": "An electronic document that is used..."
            "type": "object",
            "additionalProperties": false,
            "properties": {
                "mRID": {
                    "$ref": "#/$defs/PartyID_String"
                }
            },
            "required": [
                "mRID"
            ]
```

```
        },
        "PartyID_String": {
            "description": "The identification of...",
            "type": "object",
            "additionalProperties": false,
            "properties": {
                "codingScheme": {
                     // External $ref to the externalized and redefined
                     // codelist.
                    "$ref": "http://iec.ch/TC57/2020/urn-entsoe-eu-wgedi-
codelists.schema.json#/$defs/CodingSchemeTypeList"
                },
                "value": {
                    "description": "Main core value space.",
                    "type": "string",
                    "maxLength": 16
                }
            },
            "required": [
                "codingScheme",
                "value"
            ]
        },
        "AcknowledgementDocument": {
            "$ref": "#"
        }
    }
}
```

In the above primary schema we observe an external reference to the secondary schema with the secondary schema fully mapped as:

```
{
      "$id": "http://iec.ch/TC57/2020/urn-entsoe-eu-wgedi-
codelists.schema.json",
      "$schema": "https://json-schema.org/draft/2020-12/schema",
      "title": "urn-entsoe-eu-wgedi-codelists.schema.json",
      "description": "Complete compilation of ENTSO-E codelists...",
      "namespace": "urn:entsoe.eu:wgedi:codelists",
      "type": "object",
      "additionalProperties": false,
      "$defs": {
         "CodingSchemeTypeList": {
              "id": "#CodingSchemeTypeList",
              "title": "CodingSchemeTypeList",
              "description": "Codification scheme used to identify...",
              // Defined as the union of the standard and local codelists
              "oneOf": [
                   {
                        "$ref": "#/$defs/StandardCodingSchemeTypeList"
                   },
                   {
                        "$ref": "http://iec.ch/TC57/2020/urn-entsoe-eu-
local-extension-types.schema.json#/$defs/LocalCodingSchemeTypeList"
                   }
              ]
         },
         "StandardCodingSchemeTypeList": {
              "id": "#StandardCodingSchemeTypeList",
              "title": "",
              "description": "",
              "type": "string",
```

```
                "enum": [
                     "A01",
                     "A02",
                     "A10",
                     "NAD",
                     "NAL"
                ]
           }
      }
}
```

Finally, the `$ref` in the secondary schema resolves to the below schema where local extensions to the standard codelist are defined:

```
{
      "$id": "https://iec.ch/TC57/2020/urn-entsoe-eu-local-extension-
types.schema.json",
      "$schema": "https://json-schema.org/draft/2020-12/schema",
      "title": "urn-entsoe-eu-local-extension-types.schema.json",
      "description": "Complete compilation of ENTSO-E codelists...",
      "namespace": "urn:entsoe.eu:local:extensions:types",
      "type": "object",
      "additionalProperties": false,
      "$defs": {
          "LocalCodingSchemeTypeList": {
                "description": "Local extensions to CodingSchemeTypeList...",
                "type": "string",
                "enum": [
                     "B01"
                ]
           }
      }
}
```

To summarize, the set of `$id` identifiers in this example are

- all valid identifiers and in compliance with both the JSON schema specification as well as this IEC specification,
- expressed as URLs but which are not network resolvable,
- used as the base URIs within absolute-URI `$ref` JSON Pointers.

These `$id` (s) will be referenced in the clause that follows covering vendor implementations.

E.2.3 The JSON schema specification and vendor implementations

To close out this Annex using the applied example, we focus on resolution of the following `$ref` that appears in the primary schema:

```
{
      "$id": "http://iec.ch/TC57/2020/AcknowledgementDocument.schema.json",
      ...
      "$defs": {
        ...
        "PartyID_String": {
            ...
            "properties": {
                "codingScheme": {
                     // External $ref to an externalized codelist schema…
                    "$ref": "http://iec.ch/TC57/2020/urn-entsoe-eu-wgedi-
```

```
codelists.schema.json#/$defs/CodingSchemeTypeList"
                    },
                    ...
               },
               ...
          },
          ...
      }
}
```

As stated in the JSON schema specification:

> “The resolved URI produced by these keywords [$ref] is not necessarily a network locator, only an identifier. A schema need not be downloadable from the address if it is a network-addressable URL, and implementations SHOULD NOT assume they should perform a network operation when they encounter a network-addressable URI.”[7]

What is essential to recognize is that the `$id` declared for a schema and potentially utilized within `$ref` pointers is simply that – an identifier. Ultimately, the author of a schema may intend for the resource to be network resolvable and therefore may define the `$id` for their schema to correlate accordingly. However, as just highlighted, the JSON schema specification is explicit that implementations should not assume they should perform a network operation for network-addressable URIs. This is important to understand so as not to conclude that a particular schema’s `$id` is correct or incorrect based on how a particular vendor’s tooling behaves. In practice, an `$id` may resolve in one vendor toolset and not in another yet be a perfectly valid `$id` conforming to the specification.

Though network resolution is not required by the spec most all implementations universally support network resolvable URLs. Where implementations commonly fall short though is how they subsequently fallback in resolving URLs that are determined to not be network resolvable.

For those libraries or implementations that support network retrieval an attempt would be made to retrieve the resource from:

> `https://iec.ch/TC57/2020/urn-entsoe-eu-local-extension-types.schema.json`

This would occur even if the `AcknowledgementDocument.schema.json` primary resource had been loaded from a different location such as the local filesystem.

Assuming the URI is not resolvable via network retrieval, it is a common fallback to attempt to resolve the resource from the local file system. This approach may be provided as a matter of convenience and convention by a particular vendor’s tooling. On this subject the JSON schema specification does not define prescriptive behaviour and validator implementations therefore may vary across vendors in exactly how they try to locate the referenced schema.

The JSON schema spec itself within the “Loading a referenced schema” clause states:

> “Implementations SHOULD be able to associate arbitrary URIs with an arbitrary schema and/or automatically associate a schema's "$id"-given URI, depending on the trust that the validator has in the schema. Such URIs and schemas can be supplied to an implementation prior to processing instances, or may be noted within a schema document as it is processed…”[7]

In practice, what is observed is that non-UI based implementations (such as libraries) typically conform to the spec by supporting programmatic or configuration-based capabilities for arbitrarily mapping URIs to schemas prior to processing. Such libraries are primarily intended for validation purposes and not for the creation or editing of JSON schemas.

———————

[7] For the full content of section 9.1.2 “Loading a referenced schema” refer to the 2020-12 draft specification at https://tools.ietf.org/html/draft-bhutton-json-schema-00

For implementations associated with common UI-based tooling that allows users to create, edit and validate JSON schema the ability to associate arbitrary URIs with an arbitrary schema isn't supported. It is within such tooling that the lack of directives within the JSON specification becomes more noticable. A set of JSON schemas may be completely valid per the spec with valid absolute-URI `$id` (s) and external `$ref` (s) such as is the case for the set of schemas in this applied working example. Yet, the `$ref` (s) may be flagged as unresolvable as shown in Figure E.2:

```
33        "PartyID_String": {
34            "id": "#PartyID_String",
35            "title": "PartyID_String",
36            "description": "The identification of an actor in the energy market.",
37            "modelReference": "http://iec.ch/TC57/2010/CIM-schema-cim15#String",
38            "type": "object",
39            "additionalProperties": false,
40            "properties": {
41                "codingScheme": {
42                    "$ref": "http://iec.ch/TC57/2020/urn-entsoe-eu-wgedi-codelists.schema.json#/$defs/CodingSchemeTypeList"
43                },   Loading referenced schema file failed: http://iec.ch/TC57/2020/urn-entsoe-eu-wgedi-codelists.schema.json
44                "value": {
45                    "description": "Main Core value Space.",
46                    "modelReference": "http://iec.ch/TC57/2010/CIM-schema-cim15#String.value",
47                    "type": "string",
48                    "maxLength": 16
49                }
50            },
51            "required": [
52                "codingScheme",
53                "value"
54            ]
55        },
```

Figure E.2 – $ref (s) flagged as unresolvable

Here the three standalone schemas are co-located in the same directory on the file system. For the purposes of this working example let us assume a Windows OS with the following directory location:

```
C:\Users\Public\Documents\TC57\2020\
```

In vendor implementations a fallback to the local file system is unable to resolve the location of the schemas for the `$id` (s) and `$ref` (s) as declared in their original normative form per clause 5.2.3.1.

Manually modifying the `$id` (s) and `$ref` (s) as shown below for each schema allows the `$ref` (s) to be resolved against the local file system. The result of these suggested informative modifications is technically nothing more than an alternative set of textual identifiers for `$id` as per the spec. Given that these changes result in an `$id` that also happens to correspond to the name of a file available on the local file system is a useful convention that allows many UI-based tools to resolve resources using the fallback approach.

The suggested changes to the three schemas from the applied example are as follows:

The primary schema for the set would be updated with the results reflected by the following:

```
{
   "$id": "
file:///C:/Users/Public/Documents/TC57/2020/AcknowledgementDocument.schema.
json",
   ...
   "$defs": {
      "PartyID_String": {
         ...
         "properties": {
            "codingScheme": {
               ...
               "$ref": "urn-entsoe-eu-wgedi-
codelists.schema.json#/$defs/CodingSchemeTypeList"
```

```
                }
            }
        }
    }
}
```

What is noteworthy in the above modified primary schema is that the `$ref` JSON Pointer is expressed using just the name of the JSON schema file. In this scenario, per the JSON schema specification, the resolution by the tooling of the `urn-entsoe-eu-wgedi-codelists.schema.json` file is achieved by using the base URI (i.e. `file:///C:/Users/Public/Documents/TC57/2020/`) as derived from the absolute-URI of the `$id` of this primary schema. The result would be a proper absolute URI on the file system resolvable by most vendor implementations.

Continuing, modifications to the secondary schema containing the redefined CodingSchemaTypeList would be:

```
{
   "$id": "file:///C:/Users/Public/Documents/TC57/2020/urn-entsoe-eu-wgedi-codelists.schema.json",
   ...
   "$defs": {
      "CodingSchemeTypeList": {
         "oneOf": [
            ...
            {
               "$ref": "urn-entsoe-eu-local-extension-types.schema.json#/$defs/LocalCodingSchemeType"
            }
         ]
      }
      ...
   }
}
```

And finally the local codelist extensions schema would have its `$id` changed to:

```
{
   "$id": "file:///C:/Users/Public/Documents/TC57/2020/urn-entsoe-eu-local-extension-types.schema.json",
   ...
   "$defs": {
      "LocalCodingSchemeType": {
         "id": "#LocalCodingSchemeType",
         "title": "LocalCodingSchemeType",
         "description": "Local extensions to CodingSchemeType...",
         "type": "string",
         "enum": [
            "B01"
         ]
      }
   }
}
```

To summarize, a profile that strictly adheres to this IEC mapping specification requires an absolute-URI `$id` as normatively prescribed in clause 5.2.3.1. Given that profiles that comply with this specification result in a single standalone and self-contained file, vendor implementation variances should pose no issues when resolving internal `$ref` JSON Pointers for standalone schemas. However, when working with sets of schemas or externalizing schemas for resuse as described in this annex, tooling issues may surface when resolving `$id`

(s) and external `$ref` (s). In such cases, to allow for resolution on the local file system, the values for the `$id` and `$ref` references may be altered as outlined in this clause.

Bibliography

CIM Users Group – The community of CIM users. http://cimug.ucaiug.org

W3C: Extensible Markup Language (XML) 1.0 http://www.w3.org/XML/

IEC 60050 series, International Electrotechnical Vocabulary

IEC 61968 System Interfaces For Distribution Management series

IEC 62325 Deregulated Market Communications series

OWL Web Ontology Language Reference W3C Recommendation 10 February 2004

Unified Modelling Language (UML) Specification V2.2, Object Management Group

UN/CEFACT Core Component Technical Specification 3.0, 29 September 2009

___