HDMF Specification Language

Release v2.1.0-beta

Jan 11, 2023

Table of Contents

The HDMF specification language defines formal structures for describing the organization of complex data using basic concepts, e.g., Groups, Datasets, Attributes, and Links.

See the full language description, release notes, and credits below.

CHAPTER 1

HDMF Specification Language

Version: v2.1.0-beta[1]

Last modified: Jan 11, 2023

1.1 Introduction

In order to support the formal and verifiable specification of HDMF data file formats, HDMF defines and uses the HDMF specification language. The specification language is defined in YAML (or optionally JSON) and defines formal structures for describing the organization of complex data using basic concepts, e.g., Groups, Datasets, Attributes, and Links. A specification typically consists of a declaration of a namespace and a set of schema specifications. Data publishers can use the specification language to extend the format in order to store types of data not supported by the HDMF core format (Section 1.2).

See also:

- For detailed description of the hdmf-common data format, see here: http://hdmf-common-schema.readthedocs.io/en/latest/index.html
- The mapping of objects described in the specification language to HDF5 is described in more detail in the NWB storage docs: http://nwb-storage.readthedocs.io/en/latest/
- Data structures for interacting with the specification language documents (e.g., namespace and specification YAML/JSON files) are available as part of HDMF. For further details, see the HDMF docs available here: http://HDMF.readthedocs.io/en/latest/index.html

1.2 Extensions

As mentioned, extensions to the core format are specified via custom user namespaces. Each namespace must have a unique name (i.e., must be different from core). The schema of new data types (groups, datasets, etc.) are then specified in separate schema specification files. While it is possible to define multiple namespaces in the same file,

[1] The version number given here is for the specification language and is independent of the version number for the specification itself.

most commonly, each new namespace will be defined in a separate file with corresponding schema specifications being stored in one ore more additional YAML (or JSON) files. One or more namespaces can be used simultaneously, so that multiple extensions can be used at the same time while avoiding potential name and type collisions between extensions (as well as with the core specification).

The specification of namespaces is described in detail next in Section 1.3, and the specification of schema specifications is described in Section 1.4 and subsequent sections.

Tip: The `spec` package as part of the hdmf Python API provides dedicated data structures and utilities that support programmatic generation of extensions via Python programs, compared to writing YAML (or JSON) extension documents by hand. One main advantage of using hdmf is that it is easier to use and maintain. E.g., using hdmf helps ensure compliance of the generated specification files with the current specification language and the Python programs can often easily be just rerun to generate updated versions of extension files (with little to no changes to the program itself).

Tip: The `hdmf-docutils` package includes tools to generate Sphinx documentation from format specifications. In particular the executable `hdmf_init_sphinx_extension_doc` provides functionality to set up documentation for a format or extension defined by a namespace (similar to the documentation for NWB core namespace at http://nwb-schema.readthedocs.io/en/latest/). Use `hdmf_init_sphinx_extension_doc --help` to view the list of options for generating the docs. The package also includes the executable `hdmf_generate_format_docs` which is used for generating actual reStructuredText files and figures from YAML/JSON specification sources. For an example see: http://pynwb.readthedocs.io/en/latest/example.html#documenting-extensions

---

See also:

For examples on how to create and use extensions in PyNWB, see:

- http://pynwb.readthedocs.io/en/latest/example.html#extending-nwb : Examples showing how to extend NWB
- http://pynwb.readthedocs.io/en/latest/tutorials.html#extensions : Tutorial showing how to define and use extensions

1.3 Namespaces

Namespaces are used to define a collections of specifications, to enable users to develop extensions in their own namespace and, hence, to avoid name/type collisions. Namespaces are defined in separate YAML files. The specification of a namespace looks as follows:

```
# hdmf-schema-language 2.2.0
namespaces:
- doc: NWB namespace
  name: NWB
  full_name: NWB core
  version: 1.2.0
  date: 2019-05-22
  author:
  - Andrew Tritt
  - Oliver Ruebel
  - Ryan Ly
  - Ben Dichter
  - Keith Godfrey
  - Jeff Teeters
```

(continues on next page)

(continued from previous page)

```
  contact:
  - ajtritt@lbl.gov
  - oruebel@lbl.gov
  - rly@lbl.gov
  - bdichter@lbl.gov
  - keithg@alleninstitute.org
  - jteeters@berkeley.edu
  schema:
  - source: nwb.base.yaml
    data_types: null
    doc : Base nwb types
    title : Base types
  - ...
```

The top-level key must be `namespaces`. The value of `namespaces` is a list with the specification of one (or more) namespaces.

The beginning of the file must begin with a comment that starts with 'hdmf-schema-language' followed by a space and the version string of the specification language used by this namespace, e.g., `hdmf-schema-language 2.2.0`. Files without this comment are assumed to be defined using hdmf-schema-language 2.1.0.

1.3.1 Namespace declaration keys

1.3.1.1 doc

Text description of the namespace.

1.3.1.2 name

Unique name used to refer to the namespace.

1.3.1.3 full_name

Optional string with extended full name for the namespace.

1.3.1.4 version

Version string for the namespace

1.3.1.5 date

Date the namespace has been last modified or released. Formatting is `%Y-%m-%d %H:%M:%S`, e.g., `2017-04-25 17:14:13`.

1.3.1.6 author

List of strings with the names of the authors of the namespace.

1.3.1.7 contact

List of strings with the contact information for the authors. Ordering of the contacts should match the ordering of the authors.

1.3.1.8 schema

List of the schema to be included in this namespace. The specification looks as follows:

```
- source: nwb.base.yaml
- source: nwb.ephys.yaml
  doc: Types related to EPhys
  title: EPhys
  data_types:
  - ElectricalSeries
- namespace: core
  data_types:
  - Interface
```

- `source` describes the name of the YAML (or JSON) file with the schema specification. The schema files should be located in the same folder as the namespace file.
- `namespace` describes a named reference to another namespace. In contrast to `source`, this is a reference by name to a known namespace (i.e., the namespace is resolved during the build and must point to an already existing namespace). This mechanism is used to allow, e.g., extension of a core namespace (here the NWB core namespace) without requiring hard paths to the files describing the core namespace. Either `source` or `namespace` must be specified, but not both.
- `data_types` is an optional list of strings indicating which data types should be included from the given specification source or namespace. The default is `data_types: null` indicating that all data types should be included.
- `doc` is an optional key for source files with a doc string to further document the content of the source file.
- `title` is an option key for source files to provide a descriptive title for a file for documentation purposes.

Attention: As with any language, we can only use what is defined. This means that similar to include or import statements in programming languages, e.g., Python, the `source` and `namespace` keys must be in order of use. E.g., `nwb.ephys.yaml` defines `ElectricalSeries` which inherits from `Timeseries` that is defined in `nwb.base.yaml`. This means that we have to list `nwb.base.yaml` before `nwb.ephys.yaml` since otherwise `Timeseries` would not be defined when `nwb.ephys.yaml` is trying to use it.

1.4 Schema specification

The schema specification defines the groups, datasets and relationship that make up the format. Schemas may be distributed across multiple YAML files to improve readability and to support logical organization of types. Schema files should have the `groups` key and/or the `datasets` key at the top level.

The beginning of all schema files must begin with a comment that starts with 'hdmf-schema-language' followed by a space and the version string of the specification language used by this namespace, e.g., `hdmf-schema-language 2.2.0`. Files without this comment are assumed to be defined using hdmf-schema-language 2.1.0. The comment at the beginning of schema files must be the same as the comment at the start of the namespace file that includes the schema files.

This is the main part of the format specification. It is described in the following sections.

Note: Schema specifications are agnostic to namespaces, i.e., a schema (or type) becomes part of a namespace by including it in the namespace as part of the `schema` description of the namespace. Hence, the same schema can be reused across namespaces.

1.5 Groups

Groups are specified as part of the top-level list or via lists stored in the key `groups`. The specification of a group is described in YAML as follows:

```
# Group specification
-   data_type_def: Optional new data type for the group
    data_type_inc: Optional data type the group should inherit from
    name: Optional fixed name for the group. A group must either have a unique data␣
↪type or a unique, fixed name.
    default_name: Default name for the group
    doc: Required description of the group
    quantity: Optional quantity identifier for the group (default=1).
    linkable: Boolean indicating whether the group is linkable (default=True)
    attributes: Optional list of attribute specifications describing the attributes␣
↪of the group
    datasets: Optional list of dataset specifications describing the datasets␣
↪contained in the group
    groups: Optional list of group specifications describing the sub-groups contained␣
↪in the group
    links: Optional list of link specifications describing the links contained in the␣
↪group
```

The key/value pairs that make up a group specification are described in more detail next in Section Section 1.5.1. The keys should be ordered as specified above for readability and consistency with the rest of the schema.

1.5.1 Group specification keys

1.5.1.1 `data_type_def` and `data_type_inc`

The concept of a data type is similar to the concept of Class in object-oriented programming. A data type is a unique identifier for a specific type of group (or dataset) in a specification. By assigning a data type to a group (or dataset) enables others to reuse that type by inclusion or inheritance (*Note:* only groups (or datasets) with a specified type can be reused).

- `data_type_def`: This key is used to define (i.e., create) a new data type and to assign that type to the current group (or dataset).
- `data_type_inc`: The value of the `data_type_inc` key describes the base type of a group (or dataset). The value must be an existing type.

Both `data_type_def` and `data_type_inc` are optional keys. To enable the unique identification, every group (and dataset) must either have a fixed name and/or a unique data type. This means, any group (or dataset) with a variable name must have a unique data type.

The data type is determined by the value of the `data_type_def` key or if no new type is defined then the value of `data_type_inc` is used to determine type. Or in other words, the data type is determined by the last type in the

ancestry (i.e., inheritance hierarchy) of an object.

Reusing existing data types

The combination of `data_type_inc` and `data_type_def` provides an easy-to-use mechanism for reuse of type specifications via inheritance (i.e., merge and extension of specifications) and inclusion (i.e., embedding of an existing type as a component, such as a subgroup, of a new specification). Here an overview of all relevant cases:

data_type_inc	data_type_def	Description
not set	not set	define a standard dataset or group without a type
not set	set	create a new data type from scratch
set	not set	include (reuse) data type without creating a new one (include)
set	set	merge/extend data type and create a new type (inheritance/merge)

Example: Reuse by inheritance

```
# Abbreviated YAML specification
-   data_type_def: Series
    datasets:
    - name: A

-   data_type_def: MySeries
    data_type_inc: Series
    datasets:
    - name: B
```

The result of this is that `MySeries` inherits dataset `A` from `Series` and adds its own dataset `B`, i.e., if we resolve the inheritance, then the above is equivalent to:

```
# Result:
-   data_type_def: MySeries
    datasets:
    - name: A
    - name: B
```

Example: Reuse by inclusion

```
# Abbreviated YAML specification
-   data_type_def: Series
    datasets:
    - name: A

-   data_type_def: MySeries
    groups:
    - data_type_inc: Series
```

The result of this is that `MySeries` now includes a group of type `Series`, i.e., the above is equivalent to:

```
-  data_type_def: MySeries
   groups:
   - data_type_inc: Series
     datasets:
       - name: A
```

Note: The keys `data_type_def` and `data_type_inc` were introduced in version 1.2a to simplify the concepts of inclusion and merging of specifications and replaced the keys `include` and `merge` (and `merge+`).

1.5.1.2 name

String with the optional fixed name for the group.

Note: Every group must have either a unique fixed `name` or a unique data type determined by `data_type_def` or `data_type_inc` to enable the unique identification of groups when stored on disk.

1.5.1.3 default_name

Default name of the group.

Note: Only one of either `name` or `default_name` (or neither) should be specified. The fixed name given by `name` will always overwrite the behavior of `default_name`.

1.5.1.4 doc

The value of the group specification `doc` key is a string describing the group. The `doc` key is required.

Note: In earlier versions (before version 1.2a) this key was called `description`

1.5.1.5 quantity

The `quantity` describes how often the corresponding group (or dataset) can appear. The `quantity` indicates both minimum and maximum number of instances. Hence, if the minimum number of instances is `0` then the group (or dataset) is optional and otherwise it is required. The default value is `quantity=1`. If `name` is defined, `quantity` may not be >1.

value	minimum quantity	maximum quantity	Comment
`` `zero_or_many` `` or `` `*` ``	`0`	`unlimited`	Zero or more instances
`` `one_or_many` `` or `` `+` ``	`1`	`unlimited`	One or more instances
`` `zero_or_one` `` or `` `?` ``	`0`	`1`	Zero or one instances
`` `1`,`2`,`3`,... ``	`n`	`n`	Exactly `n` instances

Note: The `quantity` key was added in version 1.2a of the specification language as a replacement of the `` `quantity_flag` `` that was used to encode quantity information via a regular expression as part of the main key of the group.

1.5.1.6 linkable

Boolean describing whether the this group can be linked.

1.5.1.7 attributes

List of attribute specifications describing the attributes of the group. See Section 1.6 for details.

```
attributes:
- doc: Unit of measurement
  name: unit
  dtype: text
- ...
```

1.5.1.8 datasets

List of dataset specifications describing all datasets to be stored as part of this group. See Section 1.8 for details.

```
datasets:
- name: data1
  doc: My data 1
  type: int
  quantity: '?'
- name: data2
  doc: My data 2
  type: text
  attributes:
  - ...
- ...
```

1.5.1.9 groups

List of group specifications describing all groups to be stored as part of this group.

```
groups:
- name: group1
  quantity: '?'
- ...
```

1.5.1.10 links

List of link specifications describing all links to be stored as part of this group. See Section 1.7 for details.

```
links:
- doc: Link to target type
  name: link name
  target_type: type of target
  quantity: '?'
- ...
```

1.5.1.11 _required

Attention: The `\_required` key has been removed in version 2.0. An improved version may be added again in later version of the specification language.

1.6 Attributes

Attributes are specified as part of lists stored in the key `attributes` as part of the specifications of `groups` and `datasets`. Attributes are typically used to further characterize or store metadata about the group or dataset they are associated with. Similar to datasets, attributes can define arbitrary n-dimensional arrays, but are typically used to store smaller data. The specification of an attributes is described in YAML as follows:

```
...
attributes:
- name: Required string describing the name of the attribute
  dtype: Required string describing the data type of the attribute
  dims: Optional list describing the names of the dimensions of the data array stored␣
↪by the attribute (default=None)
  shape: Optional list describing the allowed shape(s) of the data array stored by␣
↪the attribute (default=None)
  value: Optional constant, fixed value for the attribute.
  default_value: Optional default value for variable-valued attributes. Only one of␣
↪value or default_value should be set.
  doc: Required string with the description of the attribute
  required: Optional boolean indicating whether the attribute is required␣
↪(default=True)
```

The keys should be ordered as specified above for readability and consistency with the rest of the schema.

1.6.1 Attribute specification keys

1.6.1.1 `name`

String with the name for the attribute. The `name` key is required and must specify a unique attribute on the current parent object (e.g., group or dataset)

1.6.1.2 `dtype`

String specifying the data type of the attribute. Allowable values are:

`dtype` **spec value**	**storage type**	**size**
• "float" • "float32"	single precision floating point	32 bit
• "double" • "float64"	double precision floating point	64 bit
• "long" • "int64"	signed 64 bit integer	64 bit
• "int" • "int32"	signed 32 bit integer	32 bit
• "short" • "int16"	signed 16 bit integer	16 bit
• "int8"	signed 8 bit integer	8 bit
• "uint64"	unsigned 64 bit integer	64 bit
• "uint32"	unsigned 32 bit integer	32 bit
• "uint16"	unsigned 16 bit integer	16 bit
• "uint8"	unsigned 8 bit integer	8 bit
• "numeric"	any numeric type (i.e., any int, uint, float)	8 to 64 bit
• "text" • "utf" • "utf8" • "utf-8"	8-bit Unicode	variable (UTF-8 encoding)
• "ascii" • "bytes"	ASCII text	variable (ASCII encoding)
• "bool"	8 bit integer with valid values 0 or 1	8 bit
• "isodatetime" • "datetime"	ISO 8601 datetime string, e.g., 2018-09-28T14:43:54.123+02:00	variable (ASCII encoding)

Note: The precision indicated in the specification is interpreted as a minimum precision. Higher precisions may be

used if required by the particular data. In addition, since valid ASCII text is valid UTF-8-encoded Unicode, ASCII text may be used where 8-bit Unicode is required. 8-bit Unicode cannot be used where ASCII is required.

1.6.1.2.1 Reference `dtype`

In addition to the above basic data types, an attribute or dataset may also store references to other data objects. Reference `dtypes` are described via a dictionary. E.g.:

```
dtype:
      target_type: ElectrodeGroup
      reftype: object
```

`target_type` here describes the `data_type` of the target that the reference points to and `reftype` describes the kind of reference. Currently the specification language supports two main reference types.

`reftype` **value**	**Reference type description**
• "ref" • "reference" • "object"	Reference to another group or dataset of the given `target_type`
• region	Reference to a region (i.e. subset) of another dataset of the given `target_type`

1.6.1.2.2 Compound `dtype`

Compound data types are essentially a `struct`, i.e., the data type is a composition of several primitive types. This is useful to specify complex types, e.g., for storage of complex numbers consisting of a real and imaginary components, vectors or tensors, as well to create table-like data structures. Compound data types are created by defining a list of the form:

```
dtype:
- name: <name of the data value>
  dtype: <one of the above basic dtype stings or references>
  doc: <description of the data>
- name: ...
  dtype: ...
  doc: ...
- ...
```

Note: Currently only "flat" compound types are allowed, i.e., a compound type may not contain other compound types but may itself only consist of basic dtypes, e.g., float, string, etc. or reference dtypes.

Below is an example from an older version of the NWB format specification showing the use of compound data types to create a table-like data structure for storing metadata about electrodes.

```
datasets:
- doc: 'a table for storing queryable information about electrodes in a single table'
  dtype:
  - name: id
```

(continues on next page)

(continued from previous page)

```
    dtype: int
    doc: a user-specified unique identifier
  - name: x
    dtype: float
    doc: the x coordinate of the channels location
  - name: y
    dtype: float
    doc: the y coordinate of the channels location
  - name: z
    dtype: float
    doc: the z coordinate of the channels location
  - name: imp
    dtype: float
    doc: the impedance of the channel
  - name: location
    dtype: ascii
    doc: the location of channel within the subject e.g. brain region
  - name: filtering
    dtype: ascii
    doc: description of hardware filtering
  - name: description
    dtype: utf8
    doc: a brief description of what this electrode is
  - name: group
    dtype: ascii
    doc: the name of the ElectrodeGroup this electrode is a part of
  - name: group_ref
    dtype:
        target_type: ElectrodeGroup
        reftype: object
    doc: a reference to the ElectrodeGroup this electrode is a part of
  attributes:
    - doc: Value is 'a table for storing data about extracellular electrodes'
      dtype: text
      name: help
      value: a table for storing data about extracellular electrodes
  data_type_inc: NWBData
data_type_def: ElectrodeTable
```

1.6.1.3 `dims`

Optional key describing the names of the dimensions of the array stored as value of the attribute. If the attribute stores an array, `dims` specifies the list of dimensions. If no `dims` is given, then attribute stores a scalar value.

In case there is only one option for naming the dimensions, the key defines a single list of strings:

```
...
dims:
- dim1
- dim2
```

In case the attribute may have different forms, this will be a list of lists:

```
...
dims:
```

(continues on next page)

(continued from previous page)

```
- - num_times
- - num_times
  - num_channels
```

Each entry in the list defines an identifier/name of the corresponding dimension of the array data.

1.6.1.4 shape

Optional key describing the shape of the array stored as the value of the attribute. The description of `shape` must match the description of dimensions in so far as if we name two dimensions in `dims` than we must also specify the `shape` for two dimensions. We may specify `null` in case that the length of a dimension is not restricted, e.g.:

```
...
shape:
- null
- 3
```

Similar to `dims` shape may also be a list of lists in case that the attribute may have multiple valid shape options, e.g.:

```
...
shape:
- - 5
- - null
  - 5
```

The default behavior for shape is:

```
...
shape: null
```

indicating that the attribute/dataset is a scalar.

1.6.1.5 value

Optional key specifying a fixed, constant value for the attribute. Default value is None, i.e., the attribute has a variable value to be determined by the user (or API) in accordance with the current data.

1.6.1.6 default_value

Optional key specifying a default value for attributes that allow user-defined values. The default value is used in case that the user does not specify a specific value for the attribute.

Note: Only one of either `value` or `default_value` should be specified (or neither) but never both at the same time, as `value` would always overwrite the `default_value`.

1.6.1.7 doc

`doc` specifies the documentation string for the attribute and should describe the purpose and use of the attribute data. The `doc` key is required.

1.6.1.8 required

Optional boolean key describing whether the attribute is required. Default value is True.

1.7 Links

The link specification is used to specify links to other groups or datasets. The link specification is a dictionary with the following form:

```
links:
- name: Link name
  doc: Required string with the description of the link
  target_type: Type of target
  quantity: Optional quantity identifier for the group (default=1).
```

Note: When mapped to storage, links should always remain identifiable as such. For example, in the context of HDF5, this means that soft links (or external links) should be used instead of hard links.

The keys should be ordered as specified above for readability and consistency with the rest of the schema.

1.7.1 Link specification keys

1.7.1.1 name

Optional key specifying the `name` of the link.

1.7.1.2 target_type

`target_type` specifies the key for a group in the top level structure of a namespace. It is used to indicate that the link must be to an instance of that structure.

1.7.1.3 doc

`doc` specifies the documentation string for the link and should describe the purpose and use of the linked data. The `doc` key is required.

1.7.1.4 quantity

Optional key specifying how many allowable instances for that link. Default is 1. Same as for groups. See Section 1.5.1.5 for details.

1.8 Datasets

Datasets are specified as part of lists stored in the key `datasets` as part of group specifications. The specification of a datasets is described in YAML as follows:

```
- datasets:
  - data_type_def: Optional new data type for the group
    data_type_inc: Optional data type the group should inherit from
    name: fixed name of the dataset
    default_name: default name of the dataset
    dtype: Optional string describing the data type of the dataset
    dims: Optional list describing the names of the dimensions of the dataset
    shape: Optional list describing the shape (or possible shapes) of the dataset
    value: Optional to fix value of dataset
    default_value: Optional to set a default value for the dataset
    doc: Required description of the dataset
    quantity: Optional quantity identifier for the group (default=1).
    linkable: Boolean indicating whether the group is linkable (default=True)
    attributes: Optional list of attribute specifications describing the attributes␣
↪of the group
```

The specification of datasets looks quite similar to attributes and groups. Similar to attributes, datasets describe the storage of arbitrary n-dimensional array data. However, in contrast to attributes, datasets are not associated with a specific parent group or dataset object but are (similar to groups) primary data objects (and as such typically manage larger data than attributes). The key/value pairs that make up a dataset specification are described in more detail next in Section Section 1.8.1. The keys should be ordered as specified above for readability and consistency with the rest of the schema.

1.8.1 Dataset specification keys

1.8.1.1 `data_type_inc` and `data_type_def`

Same as for groups. See Section 1.5.1.1 for details.

1.8.1.2 `name`

String with the optional fixed name for the dataset

Note: Every dataset must have either a unique fixed `name` or a unique data type determined by `data_type_def` or `data_type_inc` to enable the unique identification of groups when stored on disk.

1.8.1.3 `default_name`

Default name of the group.

Note: Only one of either `name` or `default_name` (or neither) should be specified. The fixed name given by `name` would always overwrite the behavior of `default_name`.

1.8.1.4 `dtype`

String describing the data type of the dataset. Same as for attributes. See Section 1.6.1.2 for details. `dtype` may be omitted for abstract classes. Best practice is to define `dtype` for most concrete classes.

1.8.1.5 shape

List describing the shape of the dataset. Same as for attributes. See Section 1.6.1.4 for details.

1.8.1.6 dims

List describing the names of the dimensions of the dataset. Same as for attributes. See Section 1.6.1.3 for details.

1.8.1.7 value and default_value

Same as for attributes. See Section 1.6.1.5 and Section 1.6.1.6 for details.

1.8.1.8 doc

The value of the dataset specification `doc` key is a string describing the dataset. The `doc` key is required.

Note: In earlier versions (before version 1.2a) this key was called `description`

1.8.1.9 quantity

Same as for groups. See Section 1.5.1.5 for details.

1.8.1.10 linkable

Boolean describing whether the this dataset can be linked.

1.8.1.11 attributes

List of attribute specifications describing the attributes of the dataset. See Section *Attributes* for details.

```
attributes:
- ...
```

1.9 Relationships

Note: Future versions will add explicit concepts for modeling of relationships, to replace the implicit relationships encoded via shared dimension descriptions and implicit references in datasets in previous versions of the specification language.

CHAPTER 2

Release Notes

2.1 Version 2.1.0 (Upcoming)

- First release as hdmf-schema-language.
- Remove legacy description of the *specs* or *spec* key.
- Add specification for the specification language used by each file.
- Add dtypes that are already supported in `hdmf.spec`: short, uint64, bytes, and datetime.
- Clarify that if *name* is defined on a group/dataset/link specification, *quantity* may not be >1.

2.2 Version 2.0.2 (March, 2020)

- add `value` and `default_value` as optional keys of a dataset.
- `dtype` changed from required to optional for datasets.

2.3 Version 2.0.1 (March, 2019)

- Added support for specifying a `title` and `doc` for `source` files as part of the `schema` portion of a `namespace` specification. This was added to improve documentation of individual source files and to support sorting of types by source file with meaningful titles and text as part of autogenerated docs.
- Updated the docs for `quantity` to indicate that the default value is `1` if not specified.

2.4 Version 2.0.0 (January, 2019)

2.4.1 Summary

- **Simplify reuse of data_types:**
  - Added new key: `data_type_def and ```data_type_inc` (which in combination replace the keys `data_type`, `include` and `merge`). See below for details.
  - Removed key: `include`
  - Removed key: `merge`
  - Removed key: `merge+`
  - Removed key: `data_type` (replaced by data_type_inc and data_type_def)
  - Removed `\_properties` key. The primary use of the key is to define abstract specifications. However, as format specifications don't implement functions but define a layout of objects, any spec (even if marked abstract) could still be instantiated and used in practice without limitations. Also, in the current instantiation of NWB:N this concept is only used for the `Interface` type and it is unclear why a user should not be able to use it. As such this concept was removed.
  - To improve compliance of NWB:N inheritance mechanism with established object-oriented design concepts, the option of restricting the use of subclasses in place of parent classes was removed. A subclass is always also a valid instance of a parent class. This also improves consistency with the NWB:N principle of a minimal specification that allows users to add custom data. This change affects the `allow_subclasses` key of links and the subclasses option of the removed `include` key.
- **Improve readability and avoid collision of keys by replacing values encoded in keys with dedicated key/value pairs:**
  - **Explicit encoding of names and types:**
    - Added `name` key
    - Removed <...> name identifier (replaced by empty `name` key)
    - Added `groups` key (previously groups were indicated by "/" as part of object's key)
    - Added `datasets` key (previously datasets were indicated by missing "/" as part of the object's key)
    - Added `links` key (previously this was a key on the group and dataset specification). The concept of links is with this now a first-class type (rather than being part of the group and dataset specs).
    - Removed `link` key on datasets as this functionality is now fully implemented by the `links` key on groups.
    - Removed / flag in keys to identify groups (replaced by `groups` and `datasets` keys)
  - **Explicit encoding of quantitites:**
    - Added new key `quantity` (which replaces the `quantity_flag`). See below for details.
    - Removed `quantity_flag` as part of keys
    - Removed *Exclude_in'* key. The key is currently not used in the NWB core spec. This feature is superseded by the ability to overwrite the `quantity` key as part of the reuse of `neurodata_types`

  - Removed `\_description` key. The key is no longer need because name conflicts with datasets and groups are no longer possible since the name is now explicitly encoded in a dedicated key/value pair.

- **Improve human readability:**
  - Added support for YAML in addition to JSON
  - Values, such as, names, types, quantities etc. are now explicitly encoded in dedicated key/value pairs rather than being encoded as regular expressions in keys.
- **Improve direct interpretation of data:**
  - Remove `references` key. This key was used in previous versions of NWB to generate implicit data structures where datasets store references to part of other metadata structures. These implicit data structures violate core NWB principles as they hinder the direct interpretation of data and cannot be interpreted (neither by human nor program) based on NWB files alone without having additional information about the specification as well. Through simple reorganization of metadata in the file, all instances of these implicit data structures were replaced by simple links that can be interpreted directly.
- **Simplified specification of dimensions for datasets:**
  - Renamed `dimensions` key to `dims`
  - Added key `shape` to allow the specification of the shape of datasets
  - **Removed custom keys for defining structures as types for dimensions:**
    - `unit` keys from previous structured dimensions are now `unit` attributes on the datasets (i.e., all values in a dataset have the same units)
    - The length of the structs are used to define the length of the corresponding dimension as part of the `shape` key
    - `alias` for components of dimensions are currently encoded in the dimensions name.
- **Added support for default vs. fixed name for groups and datasets:**
  - Added `default_name` key for groups and dataset to allow the specification of default names for objects that can have user-defined names (in addition to fixed names via `name`). Attributes can only have a fixed name since attributes can not have a neurodata_type and can, hence, only be identified via their fixed name.
- **Updated specification of fixed and default values for attributes to make the behavior of keys explicit:**
  - **Specifying attribute values:**
    - Added `default_value` key for attributes to specify a default value for attributes
    - Removed `const` key for attributes which was used to control the behavior of the `value` key, i.e., depending on the value of `const` the `value` key would either act as a fixed or default value. By adding the `default_value` key this behavior now becomes explicit and the behavior of the `value` key no longer depends on the value of another key (i.e., the `const` key)
- **Improved governance and reuse of specifications:**
  - The core specification documents are no longer stored as .py files as part of the original Python API but are released as separate YAML (or optionally JSON) documents in a seperate repository
  - All documentation has been ported to use reStructuredText (RST) markup that can be easily translated to PDF, HTML, text, and many other forms.

  - Documentation for source codes and the specification are auto-generated from source to ensure consistency between sources and the documentation

- **Avoid mixing of format specification and computations:**
  - Removed key `autogen` (without replacement). The autogen key was used to describe how to compute certain derived datasets from the file. This feature was problematic with respect to the guiding principles of NWB for a couple of reasons. E.g., the resulting datasets were often not interpretable without the provenance of the autogeneration procedure and autogeneration itself and often described the generation of derived data structures to ease follow-on computations. Describing computations as part of a format specification is problematic as it creates strong dependencies and often unnecessary restrictions for use and analysis of data stored in the format. Also, the reorganization of metadata has eliminated the need for autogen in many cases. A autogen features is arguably the role of a data API or intermediary derived-quantity API (or specification), rather than a format specification.
- **Enhanced specification of data types via `dtype`:**
  - Enhanced the syntax for `dtype` to allow the specification of flat compound data types via lists of types
  - Enhanced the syntax for `dtype` to allow the specification of i) object references and ii) region references
  - Removed "!" syntax (e.g., "float32!") previously used to specify a minimum precision. All types are interpreted as minimum specs.
  - Specified list of available data types and their names
  - Added `isodatetime` dtype for specification of ISO8061 datetime string (e.g., `2018-09-28T14:43:54.123+02:00`) as data type
  - Added `bool` dtype for specification fo boolean type fields (see PR691 (PyNWB) and I658 (PyNWB).
- **Others:**
  - Removed key `\_\_custom` (without replacement). This feature was used only in one location to provide user hints where custom data could be placed, however, since the NWB specification approach explicitly allows users to add custom data in any location, this information was not binding.

2.4.2 Currently unsupported features:

- `_required` : The current API does not yet support specification and verification of constraints previously expressed via `_required`.
- Relationships are currently available only through implicit concepts, i.e., by sharing dimension names and through implicit references as part of datasets. The goal is to provide explicit mechanisms for describing these as well as more advanced relationships.
- `dimensions_specification`: This will be implemented in later version likely through the use of relationships.

2.4.3 YAML support

To improve human readability of the specification language, Version 1.2a now allows specifications to be defined in YAML as well as JSON (Version 1.1c allowed only JSON).

2.4.4 `quantity`

Version 1.1c of the specification language used a `quantity_flag` as part of the name key of groups and datasets to the quantity

- *!* - Required (this is the default)
- *?*- Optional
- ^ - Recommended
- + - One or more instances of variable-named identifier required
- * - Zero or more instances of variable-named identifier allowed

Version 1.2a replaces the `quantity_flag` with a new key `quantity` with the following values:

value	required	number of instances
`zero_or_more` or `*`	optional	unlimited
`one_or_more` or `+`	required	unlimited but at least 1
`zero_or_one` or `?`	optional	0 or 1
`1`, `2`, `3`, ...	required	Fixed number of instances as indicated by the value

2.4.5 `merge` and `include`

To simplify the concept `include` and `merge`, version 1.2a introduced a new key `neurodata_type_def` which describes the creation of a new neurodata_type. The combination `neurodata_type_def` and `neurodata_type_inc` simplifies the concepts of merge (i.e., inheritance/extension) and inclusion and allows us to express the same concepts in an easier-to-use fashion. Accordingly, the keys `include`, `merge` and `merge+` have been removed in version 1.2a. Here a summary of the basic cases:

neuro-data_type_inc	neuro-data_type_def	Description
not set	not set	define standard dataset or group without a type
not set	set	create a new neurodata_type from scratch
set	not set	include (reuse) neurodata_type without creating a new one (include)
set	set	merge/extend neurodata_type and create a new type (merge)

2.4.6 `structured_dimensions`

The definition of structured dimensions has been removed in version 1.2a. The concept of structs as dimensions is problematic for several reasons: 1) it implies support for defining general tables with mixed units and data types which are currently not supported, 2) they easily allow for colliding specification where mixed units are assigned to the same value, 3) they are hard to use and unsupported by HDF5. Currently structured dimensions, however, have been used only to encode information about "columns" of a dataset (e.g., to indicate that a dimension stores x,y,z values). This information was translated to the `dims` and `shape` keys and `unit` attributes. The more general concept of structured dimensions will be implemented in future versions of the specification language and format likely via support for modeling of relationships or support for table data structures (stay tuned)

2.4.7 `autogen`

The `autogen` key has been removed without replacement.

Reason: The autogen specification was originally used to specify that the attribute or dataset contents (values) can be derived from the contents of the HDF5 file and, hence, generated and validated automatically. As such, autogen crossed a broad range of different functionalities, including:

1. Specification of the structure of format datasets/attributes
2. Description of data constraints (e.g., the shape of the generated dataset directly depends on the structure of the input data consumed by autogen),
3. Specification of the content (i.e., value) of datasets and attributes,
4. Description of computations to create derived data, and
5. Validation of the structure and content of datasets/attributes.

This mixing of functionality in turn led to several concerns:

- autogen exhibited a fairly complex syntax, which made it hard to interpret and use
- autogen is specifically used to create derived data from information that is already in the NWB file. Attributes/datasets generated via autogen: i) are redundant, ii) often require bookkeeping to ensure data consistency, iii) generate dependencies across data and types, iv) have limited utility as the information can be derived through other means, and v) interpretation of data values may require the provenance of autogen.
- Description of computations as part of a format specification was seen as problematic.
- There was potential for collisions between autogen and the specification of the dataset/attribute itself.

Usage in NWB autogen was used in NWB V.1.0.6 to generate 17 datasets/attributes primarily to: i) store the path of links in separate datasets/attributes or ii) generate lists of datasets/groups of a given type/property. The datasets were reviewed at a hackathon and determined to be non-essential and as such removed from the format as well.

2.5 Version 1.1c (Oct. 7, 2016)

- Original version of the specification language generated as part of the NWB pilot project

CHAPTER 3

Credits

3.1 Acknowledgments

The HDMF schema language was ported from the schema language used by the Neurodata Without Borders (NWB) neurophysiology data standardization project. NWB now uses the HDMF schema language with some different key names.

3.2 Authors

- Ben Dichter
- Ryan Ly
- Oliver Ruebel
- Andrew Tritt

CHAPTER 4

Legal

4.1 Copyright

4.2 License

OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

You are under no obligation whatsoever to provide any bug fixes, patches, or upgrades to the features, functionality or performance of the source code ("Enhancements") to anyone; however, if you choose to make your Enhancements available either publicly, or directly to Lawrence Berkeley National Laboratory, without imposing a separate written license agreement for such Enhancements, then you hereby grant the following license: a non-exclusive, royalty-free perpetual license to install, use, modify, prepare derivative works, incorporate into other computer software, distribute, and sublicense such enhancements or derivative works thereof, in binary and source code form.