Versioning NWB Specification Namespaces

- Version: 0.3.2
- Authors:
    - Oliver Ruebel
    - Ryan Ly
- Last update: December 7, 2021

Overview

The purpose of this document is to define the requirements and strategy for versioning namespaces for NWB extensions.

Definitions

- The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.
- **“Neurodata Extensions”** (NDX) refer to extensions to the NWB data standard. NDX MUST be described by a formal format specification using the NWB specification language.
- **“Namespace”** is a declarative region for format specifications that provides a scope to the identifiers (the neurodata_types, groups, datasets, links, attributes, etc.) inside it. Namespaces are used to organize format specifications into logical groups and to prevent name collisions that can occur especially when using multiple extensions to the core data specification.
- **“Sub-namespace”** refers to a namespace that is being used exclusively as a sub-component of an extension “Namespace.” I.e., sub-namespaces are always included in the main namespace and released in conjunction with the main namespace. Sub-namespaces can be useful if separate versioning of sub-components of a namespace is desirable. Generally we RECOMMEND to organize extensions into a single “Namespace” with possible multiple extension source yaml files, rather than using “Sub-namespaces”.

Summary

NWB uses semantic versioning for format specifications (schema). Versions are assigned on a namespace level and are stored in the “version” key of the corresponding namespace YAML file. A version number consists of the following components MAJOR.MINOR.PATCH. Version numbers SHOULD be incremented as follows:

1. MAJOR version is incremented when adding incompatible schema changes,
2. MINOR version is incremented when functionality is added in a backwards-compatible manner (e.g., addition of new neurodata types),
3. PATCH version is incremented when backwards compatible bug fixes or other changes that do not affect the actual specification (e.g., correction of documentation). For public releases, NWB does not allow custom extensions (e.g., adding the suffixes “-beta”, “-rc”) to the semantic versioning. Custom extensions may be used internally on development branches but should be removed for public release.

Further details on schema compatibility and what kinds of changes are allowed in major, minor, and patch versions are described in the NWB and HDMF Versioning and Compatibility Policy.

Versioning rules

1. Extensions to NWB MUST contain a valid and complete specification of the namespace and all types via YAML files compliant with the NWB specification language. It is further RECOMMENDED that sources for generating the extensions via the PyNWB/HDMF specification tools also be released in conjunction with the extension.
2. A version number MUST take the form X.Y.Z. X, Y, and Z MUST be non-negative integers. X, Y, and Z MUST NOT contain leading zeroes. X identifies the major version, Y identifies the minor version, and Z identifies the patch version. Each element MUST increase numerically. For instance: 1.9.0 -> 1.10.0 -> 1.11.0.
3. Any modifications to a specification MUST be released as a new version. That is, once a specific version of a specification has been released, the contents of that version MUST NOT be modified.
4. Initial version numbers SHOULD be created as follows:

    1. For initial development, version numbers with a MAJOR version zero (i.e., 0.y.z) SHOULD be used indicating that anything may change at any time and that the extensions SHOULD NOT be considered stable.
    2. Version 1.0.0 defines the public API. Incrementation of version numbers thereafter MUST follow the semantic versioning rules outlined here.
5. Version numbers MUST be incremented as follows:
    1. PATCH version Z (x.y.Z where x > 0) MUST be incremented if only backwards compatible bug fixes or other changes that do not affect the actual specification (e.g., correction of documentation etc.) are introduced. Bug fix defines an internal change that does not affect the actual data format specification. The PATCH version MUST be reset to 0 when the MAJOR or MINOR version is incremented.
    2. MINOR version Y (x.Y.z where x > 0) MUST be incremented if new, backwards compatible functionality is introduced to the public specification. It MUST be incremented if any public API functionality is marked as deprecated and if new functionality or improvements are introduced. It MAY include patch level changes. The PATCH version MUST be reset to 0 when MINOR version is incremented. The MINOR version MUST be reset to 0 when the MAJOR version is incremented.
    3. MAJOR version X (X.y.z where X > 0) MUST be incremented if any backwards incompatible changes are introduced to the public format specification. It MAY include minor and patch level changes. The PATCH and MINOR version MUST be reset to 0 when the major version is incremented.
6. Public releases of the format specification MUST NOT contain custom extensions to the semantic versioning. Custom extensions (e.g., adding the suffixes “-beta”, “-rc”) MAY be used internally on development branches but MUST be removed for full public release. For internal releases, addition of a hyphen followed by lowercase alphabetic identifiers (a-z) SHOULD be used. Internal release additions SHOULD be comparable via standard alphabet ordering (e.g,. 1.0.0a < 1.0.0b). Typically, not the full spectrum of available alphabet characters are used for internal pre-releases, e.g., in the case of the following SUGGESTED internal versioning scheme:
    1. The postfix “-alpha” (e.g., 2.0.1-alpha) may be used to indicate internal alpha releases. Internal alpha releases are considered not stable and under development.

    2. The postfix “-beta” (e.g., 2.0.1-beta) may be used to indicate internal beta releases. Internal beta releases are considered usable but still under development.
    3. The postfix “-rc” (e.g., 2.0.1-rc) may be used to indicate internal release candidates. Internal release candidates are considered usable and stable, expecting only minor changes for full public release.
7. To support versioning of subcomponents of an extension, an extension MAY contain sub-namespaces that are included in the main extension namespace. Sub-namespaces MUST follow the same versioning rules outlined above. In addition the version of the main namespace (and any sub-namespaces that include a corresponding sub-namespace) MUST update their version numbers accordingly when the version of a sub-namespace is incremented.

Determining version precedence

Version precedence refers to how versions are compared to each other when ordered. Precedence MUST be calculated by separating the version into MAJOR, MINOR, and PATCH (and for internal pre-release, additional alphabetic identifiers) in that order. Precedence is determined by the first difference when comparing each of these identifiers from left to right as follows: MAJOR, MINOR, and PATCH versions are always compared numerically. For example: 1.0.1 < 2.1.0 < 2.2.0 < 2.3.1. Versions with identical MAJOR and MINOR versions are considered backwards compatible, i.e., a file generated with version 2.0.x MUST be able to be read using all versions 2.0.y (y >= x). Internal release additions to the versioning schema are considered to have lower precedence than regular versions (e.g., 2.0.0-alpha < 2.0.0, i.e., pre-release versions are considered lower than the same regular release version). Precedence for two pre-release versions with the same MAJOR, MINOR, and PATCH version SHOULD be determined by lexical comparison in ASCII sort order (i.e., comparison left-to-right in alphabetic order). E.g. 1.0.0-a < 1.0.0-b and for internal pre-release additions consisting of multiple letters 1.0.0-a < 1.0.0c < 1.0.0-ca < 1.0.0-cb < 1.0.0-d.

References

The rules outlined in this document have been derived from Semantic Versioning 2.0.0 (https://semver.org).