

Using NVH as a Backbone Format in the Lexonomy Dictionary Editor

Miloš Jakubíček, Vojtěch Kovář, Michal Měchura, and Adam Rambousek

Natural Language Processing Centre
Faculty of Informatics, Masaryk University, Brno, Czechia
{jak, xkovar3, xrambous}@fi.muni.cz

Lexical Computing
Brno, Czechia
{milos.jakubicek, vojtech.kovar, adam.rambousek}@sketchengine.eu

Abstract. In this paper we present an ongoing development in the Lexonomy dictionary editor consisting of replacing the XML backbone of the editor with an NVH-based one. We describe the core properties of the recently introduced NVH format, implications for using it in Lexonomy as well as a self-contained Python implementation in the form of one script (`nvh.py`) that can be used for several standard processing operations such as parsing, serialization, search or schema validation. We also outline some planned future development related to the usage of NVH in Lexonomy.

Keywords: NVH, XML, Lexonomy, dictionary editor

1 Introduction

This paper focuses on the development of Lexonomy, a lightweight dictionary editing system [1,2]. Lexonomy is a web-based tool which allows users to upload, edit and publish their dictionaries. It is tightly bound to the Sketch Engine corpus management system [3]: users can easily import corpus content from Sketch Engine into Lexonomy, either manually (pull corpus examples, collocations or thesaurus items) or automatically (by using the OneClick Dictionary approach [4]) or in a post-editing fashion [5]) where the dictionary is initially drafted fully automatically and post-edited in isolated steps inside of Lexonomy.

From the beginning, dictionaries in Lexonomy were stored as XML data of arbitrary XML schemas, stored as plain text in an SQLite database [6] and edited using the browser-based Xonomy XML editor on the front-end [7]. The main motivation behind this decision was the emphasis on flexibility (so that users could upload dictionaries not restricted in their schemas) as well as reliance on a widely known data format (XML). Over the five years of Lexonomy development, we have however established that this (i.e. arbitrary

XML-based schemas) hinders any further Lexonomy development as a platform for automating dictionary production.

Particularly, we have established the following findings:

- **unrestricted dictionary schemas are difficult to handle by most users** For any automatic extraction of data from corpora, the user needs to manually tell what information should be included into what part of the dictionary entry, such as which entry part should contain dictionary examples, whether the samples are per-sense or per-headword etc. For users-lexicographers that are not skilled in data modelling (and we assume this is the vast majority), this is a very error-prone task, especially if they initially designed a complex dictionary schema.
- **unrestricted XML serialization is not suitable for dictionaries** While XML became standard exchange and data format in many applications, including dictionaries, it is not suitable as format that should be human-editable, it is (without any restrictions) rather difficult to process computationally in terms of data manipulation and database search and it also has been shown as not suitable for dictionary modelling [8].

While the first issue – data modelling in lexicography – is currently being addressed by the LEXIDMA consortium as a forthcoming OASIS standard [9]¹, in this paper we focus on the latter problem and describe an alternative plain text data format (NVH), its manipulation tool implemented in Python and used by Lexonomy.

2 NVH data format

NVH stands for name-value hierarchy.² It is a plain text data format significantly simpler than XML. A NVH file is a list of nodes, each node having a value and (optionally) a list of children nodes (see examples in Figures 1 and 2.)

```
node1: value of node1
  childnode1: value of childnode1
  childnode2: value of childnode2
    grandchildnode1: value of grandchildnode1
  childnode3: value of childnode3
node2: value of node2
node3: value of node3
```

Fig. 1: Structure of the NVH format in a nutshell.

¹ See <https://www.oasis-open.org/committees/lexidma/>

² See <https://namevaluehierarchy.org>

```
hw: at-c
  language: Tagalog
  lemma: at
  freq: 332418
  pos: c
  flag: ok
  sense:
    example: Kumakain siya ng prutas at gulay.
    quality: good
    example_english: She was eating fruits and vegetables.
    english: and
```

Fig. 2: A sample dictionary entry represented in NVH.

The expressiveness of the format follows from its simplicity. Each node must be placed on a separate line, it features a Python-style mandatory indentation of children nodes and each name-value pair must be separated by the colon-space character pair. No other strings bear particular semantics (value is defined as the string following the separator and ending by the newline character, it may be empty), leaving it up to the user for specification of higher-level constructs like namespaces, intra-file cross-references or external links. Obviously, NVH is much easier to parse by a program as well as much easier to read or write by a human user.

In this paper we present a processing tool for NVH that is implemented as self-contained Python script `nvh.py` and is available from the official project size of NVH.³ The script in its current version implements the following operations:

- parsing into a Python data structure
- serialization of the same structure into NVH
- search by a simple query language
- splitting by top-level node into multiple files
- merging two files in a patch-style fashion
- schema generation from an existing NVH file
- schema validation of an NVH file against a predefined schema
- export to XML
- export to JSON

The `nvh.py` script can be either imported into another Python script (which would be the typical scenario when using it for parsing and subsequent custom manipulation of the parsed content) or used from command-line where the first argument specifies the action to be taken, as we describe in detail in the following:

³ See <https://github.com/michmech/nvh/blob/master/python/python.md>

2.1 Parsing and search: `nvh.py get`

The full syntax of the command is:

```
nvh.py get file.nvh [ SELECT_FILTER [ PROJECT_FILTER ] ]
```

The selection and projection filter are optional, not using them means that the script would be parse the `file.nvh` and reprint its content. The notions of selection and projection follow the paradigm of relational algebra: a selection filter specifies which nodes should be retrieved, a projection filter specifies what parts of the selected nodes will be retrieved. Not using the projection filter implies printing the whole top-level node matching the selection criteria.

Filters use a dot-style language to identify node parts (sense translation into English given in Figure 2 would be identified as `hw.sense.english`). There may be multiple selection filters which are logically ANDed, and each node in the filter may feature one of the following operators:

- equality (`=STRING`)
- regular expression matching (`~=Python RE`)
- count (`#=, #> or #<`)

Each of the operators can be negated by prefixing it with an exclamation mark (`!`). A special selection filter taking the form of `##NUMBER` may be used only at the beginning of the list of selection filters, limiting the retrieval to the first `NUMBER` items only. An example of a search command using the data in Figure 2 would be

```
nvh.py get file.nvh 'hw.sense.example#>0.quality=good' hw.sense
```

This query would retrieve all `sense` (identified as `hw.sense`) from entries having at least one example marked as good quality. More examples are available in the project documentation online.⁴

2.2 Merging and patching: `nvh.py put`

The full syntax of the command is:

```
nvh.py put file.nvh patch.nvh [ REPLACE_FILTER ]
```

It merges the content of `patch.nvh` into `file.nvh` by finding shared nodes and appending any nodes and children nodes not present yet from the patch into main file. The optional dot-style replace filter may be used to select which portion of the patch (such as an entry part only) shall be merged.

2.3 Splitting: `nvh.py split`

Splitting is used to split the NVH file by top-level nodes, or to put it in dictionary terms, to generate one NVH file per dictionary entry. The command `nvh.py split file.nvh DIRECTORY` takes the `file.nvh` as input and generates individual files into `DIRECTORY`. This is useful e.g. to keep per-entry copies tracked and versioned by a file-based management system such as Git.

⁴ See <https://github.com/michmech/nvh>.

2.4 Schema generation and validation: `nvh.py genschema|checkschema`

The command `nvh.py genschema file.nvh` parses the input `file.nvh` and generates a corresponding NVH schema.⁵ An NVH schema describes valid node names and their allowed cardinality using (obligatory/optional, Kleene plus/Kleene star).

The counterpart is then represented by the `nvh.py checkschema file.nvh schema.nvh` command which validates `file.nvh` against a schema given in `schema.nvh`.

2.5 Generic exports: `nvh.py xmlexport|jsonexport`

These two commands perform generic exports to XML and JSON, respectively. The XML export transforms all nodes into XML elements bearing their value in an attribute and keeping child nodes as child XML elements. The example in Figure 2 would be transformed into an XML file as presented in Figure 3 and into JSON as presented in Figure 4.

```
<?xml version="1.0"?>
<dictionary>
  <hw v="at-v">
    <lemma v="at" />
    <language v="Tagalog" />
    <pos v="c" />
    <freq v="332418" />
    <sense>
      <example v="Kumakain siya ng prutas at gulay.">
        <quality v="good" />
        <example_english v="She was eating fruits and vegetables." />
      </example>
      <english v="and" />
    </sense>
  </hw>
</dictionary>
```

Fig. 3: Generic XML export from an NVH input of Figure 2.

3 Using NVH as a Lexonomy backbone

The flexibility of Lexonomy in terms of dictionary schemes has always been an important feature. Using NVH inside Lexonomy instead of XML enables us to

⁵ See <https://github.com/michmech/nvh/blob/master/docs/schema.md> for detailed description of the schema format.

maintain a simple text format usable for human reading and writing as well as very efficient machine processing. Data may be saved in the underlying SQLite database directly in the NVH format, individual entries but also small dictionaries can be directly searched using `nvh.py` with very low latency response (less than a second).

For large dictionaries, the JSON conversion is used for one-way encoding into JSON and using the built-in SQLite JSON indexing to store the JSON content. A simple conversion procedure has been developed to translate the query language of `nvh.py get` into SQLite SQL-JSON queries. Both NVH and JSON content is stored in the database for each entry, using JSON only for fast indexed search, but NVH for any editing. Upon every update, the JSON content is regenerated.

```
{
  "hw": [ {
    "value": "at-c",
    "children": {
      "lemma": [ {"value": "at", "children": {}} ],
      "freq": [ {"value": "332418", "children": {}} ],
      "pos": [ {"value": "c", "children": {}} ],
      "flag": [ {"value": "ok", "children": {}} ],
      "sense": [ {
        "value": "",
        "children": {
          "example": [ {
            "value": "Kumakain siya ng prutas at gulay.",
            "children": {
              "quality": [ {"value": "best", "children": {}} ],
              "example_english": [ {
                "value": "She was eating fruits and vegetables.",
                "children": {}
              } ]
            } ]
          } ]
        } ],
        "english": [ {"value": "and", "children": {}} ]
      } ]
    } ]
  } ]
}
```

Fig. 4: Generic JSON export from an NVH input of Figure 2.

4 Conclusions and future development

In this paper we have presented recent development of Lexonomy consisting of replacing the XML backbone with an NVH-based one. This development is in line with the spirit of Lexonomy being a lightweight dictionary editor that can handle very large dictionaries (hundreds of thousands of complex entries) efficiently and support modern lexicographic workflow that is tightly connected to corpus data.

For Lexonomy, this particularly means to support the post-editing approach to dictionary making and Lexonomy does that by making it easy to maintain multiple versions of the dictionary, edit them simultaneously by the editorial team, split them and merge them as needed into individual editing tasks with custom editing widgets, while having a comprehensive NVH version available at every stage of the process.

In the future, more functions related to the management of the post-editing workflow are going to be added to Lexonomy, in the first place an NVH-based front-end editor is going to replace the current Xonomy-based one.

Acknowledgements This work has been partly supported by the Ministry of Education of CR within the Lindat Clarin Center. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731015.

References

1. Měchura, M.B., et al.: Introducing Lexonomy: an open-source dictionary writing and publishing system. In: *Electronic Lexicography in the 21st Century: Lexicography from Scratch*. Proceedings of the eLex 2017 conference. (2017) 19–21
2. Rambousek, A., Jakubíček, M., Kosem, I.: New developments in lexonomy. *Electronic lexicography in the 21st century (eLex 2021) Post-editing lexicography (2021)* 86
3. Kilgarriff, A., Baisa, V., Bušta, J., Jakubíček, M., Kovář, V., Michelfeit, J., Rychlý, P., Suchomel, V.: The Sketch Engine: ten years on. *Lexicography* 1 (2014)
4. Jakubíček, M., Kovář, V., Rychlý, P.: Million-Click Dictionary: Tools and Methods for Automatic Dictionary Drafting and Post-Editing. *EURALEX XIX (2021)*
5. Blahuš, M., Cukr, M., Herman, O., Jakubíček, M., Kovář, V., Medveď, M.: Semi-automatic building of large-scale digital dictionaries. *Electronic lexicography in the 21st century (eLex 2021) Post-editing lexicography (2021)* 99
6. Hipp, R.D.: SQLite (2020)
7. Měchura, M.B.: Building XML Editing Applications with Xonomy (2018)
8. Měchura, M.B.: Better than XML: Towards a Lexicographic Markup Language. Available at SSRN <https://ssrn.com/abstract=4165854> (2022)
9. Tiberius, C., Krek, S., Depuydt, K., Gantar, P., Kallas, J., Kosem, I., Rundell, M.: Towards the elexis data model: defining a common vocabulary for lexicographic resources. *Electronic lexicography in the 21st century (eLex 2021) Post-editing lexicography (2021)* 91