UJO Schemes Documentation

The Titan Project

Dec 20, 2019

Contents

API Documentation:

CHAPTER 1

ujs2md cli tool

CHAPTER 2

ujsParser package

2.1 Submodules

2.2 ujsParser.parse2md module

2.3 ujsParser.parser module

UJO Schemes:

CHAPTER 3

UJO Schemes

UJO Schemes is an easy to read and easy to write language to define UJO data structures. The definition is translated into a documentation and/or compiled into a binary form for fast an reliable checks on data sets.

3.1 Convert UJO Schemes to markdown documentation

UJO Schemes files can be converted into a markdown documentation `ujs2md.py`.

Usage:

```
usage: ujs2md.py [-o <output>] <source>

positional arguments:
  source                path to ujo scheme file or folder
                        if providing a folder by default all files will be processed
                        you might need to provide an extension by using the "-ext/--
↪extension" option

optional arguments:
  -h, --help            show this help message and exit
  -o <output>, --output <output>
                        output path for markdown
  -ext <extension>, --extension <extension>
                        filter files by extension if providing a folder as <ujs_file>
                        example: ".ujs"
                        defaults to ".ujs"
```

Example:

```
python .\ujs2md.py -o testoutput -ext .ujs .\examples\ujs2md
```

3.2 Module Name

UJO Schemes can be divided into multiple modules. Each module is described in one file. At the beginning of a file the module name is defined. Additionally a documentation section for the particular module can be added.

```
module myModule;
```

Adding documentation is done by using the doc keyword.

```
module myModule
    : doc """This text is a description
of my module""";
```

3.3 Types

3.3.1 Atomic Types

Atomic types define the basic data fields in UJO. All data structures are build upon atomic types.

3.3.2 Container Types

Container types are used to organize values. A value in a container can be a container. This nesting allows to define complex comprehensive data structures.

3.3.3 Variant Type

The variant can hold values of any atomic and container type. The only constraint possible for variant type definitions is to exclude null as a possible value.

3.4 Defining Constraint Types

Based on Atomic and Container Types new types can be defined by applying constraint rules on them.

Creating a new type based on an existing atomic type without constraints. The new type can contain the same values as the original type.

```
new_type = int64;
```

The new type can be documented using `doc`.

```
new_type = int64 : doc "This is my new type"
```

Multiple lines can be used for better readability.

```
new_type = int64
    : doc "This is my new type";
```

3.5 Constraint Rules

Constraint Rules are used to define constraints on an atomic type.

3.5.1 Defining specific values

Storypoints are an agile metric containing only specific numbers.

```
StoryPoints = uint16
    : in (1, 2, 3 ,5, 8, 13, 20, 40, 100 );

SciConst = float32
    : in (3.14, 9.81, 343,2);
```

The `in` keyword can also be used to define specific words for a string.

```
CardColor = string
    : in ("Heart", "Spade", "Diamond", "Club");
```

3.5.2 Defining value ranges

A range includes all values from a lowest value to highest value. If the lowest or highest value is omitted, the minimum or maximum possible value of the chosen atomic type is used. This rule can only be applied to numeric types.

```
# all values from 0 to 10
lowRange = uint32
    : in ( .. 10 );

# all values from 10 to 4.294.967.295
HiRange = uint32
    : in ( 10 .. );
```

3.5.3 Documenting values

Values and ranges can be documented using `doc`.

```
CardColor = string
    : in (
        "Heart"     : doc "the red heart symbol",
        "Spade"     : doc "this is black",
        "Diamond"   : doc "a red symbol",
        "Club"      : doc "looks like a little tree");
```

3.5.4 Make a value mandatory

Values is UJO can be null by default. If null is not allowed in a dataset the `not null` rule is applied.

```
new_type = int64
    : not null
    : doc "This is my new type with no null values allowed";
```

If a value is mandatory, a default value can be applied.

```
new_type = int64
    : not null default 5
    : doc "This is my new type with no null values allowed, but with an automatic␣
↪default value of 5";
```

3.6 List Type

A list is a container type that can contain values of all valid ujo types by default.

3.6.1 A list for a specific type

A list can created from any valid type including container types. Here is an example how to create a list of int64 values. Only int64 values and null can be stored.

```
intList = list of int64;
```

If I want to exclude null values from the list I can apply the relating type rule.

```
intList = list of int64
    : not null;
```

A range can be applied as well.

```
intList = list of int64
    : not null
    : in ( 100 .. 200 );
```

A constraint type can be defined first and used in the list definition.

```
# a constraint type
MyType = int64
    : in ( 100 ..200)
    : not null;

# a list of this type
intList = list of MyType;
```

The `doc` rule can be applied to document the new list type. The following example applies the `doc`rule with a text with line feeds.

```
# a constraint type
MyType = int64
    : in ( 100 ..200)
    : not null
: doc "a new list type";

# a list of this type
intList = list of MyType
: doc """A list type
documentation in multiple lines""";
```

3.6.2 Defining a Record

A record is a constraint on a list container. It defines a limited, fixed sequence of values with specific and fixed types.

```
header = list [
    timestamp,
    int64,
    int16,
    string,
    list
];
```

For reference and probably for later conversions into JSON or XML data a name can be applied to the data fields in the record.

```
header = list [
    timestamp   : name CreationTime,
    int64       : name SequenceNumber,
    int16       : name Status,
    string      : name Message,
    list        : name Values
];
```

Constraint rules can be applied on each value and the field can be documented.

```
header = list [
    timestamp
        : name CreationTime
        : doc "Creation time of the message",
    int64
        : name SequenceNumber
        : doc "sequence number to order the messages",
    int16
        : name Status
        : in (
                0 : doc "Ok",
                1 : doc "Warning",
                2 : doc "Error",
                3 : doc "Critical"
            )
        : not null
        : doc "Processing status",
    string
        : name Message
        : doc "An error message",
    list
        : name Values
        : doc "a list with some values"
] : doc "This is a record";
```

3.6.3 Extending a record

An already define record can be extended to contain more fields. The resulting records appends the new fields to the previously defined record part.

```
aMessage = list extends header [
    float32
        : name temperature
        : doc "value read from a sensor",
    boolean
        : name FanStatus
        : doc "True = On, False = Off"
];
```

3.7 Associative array (map)

Constraints on Assoziative arrays apply to keys and values. All atomic types are allowed to be used as keys. Values can be any type including containers.

3.7.1 Define a static array with fixed keys

Define keys of a map with a type definition of each key/value pair.

```
 <identifier> = map {
    <key>[ as <type>] | <value type>,
 };
```

The `as <type>` definition is a cast of the default literal type to a specific atomic type. The cast is optional.

The following example shows how to define a static map.

```
mapType = map { 3.14 as float32 | list,
                "temperature"   | cstring : doc "another doc string" }
: doc "map defintion";
```

3.7.2 Use a custom datatype as map key

A custom datatype can be used as key for a map. All scalar types can be used.

Example:

```
new_str = cstring
    : in (
        "Test",
        "Fact"
    );

mapType = map { 3.14 as float32 | list,
                "temperature" | float32
                    : doc "another doc AA:XX string",
                "items" | list of int32
                    : length(2 .. 6)
                    : doc "a list with a specific length",
                new_str | int64
                    : doc "a data type as key",
                float32 | int64
                    : doc "a data type as key"
```

(continues on next page)

(continued from previous page)

```
                }
: doc "map defintion";
```

3.7.3 Extending a static map

A static map defintion can be extended using the `extend` keyword.

```
extMapType = map extends mapType {
                5       | list,
                "test"  | cstring : doc "another doc string" }
: doc "map defintion";
```

3.7.4 Define typed maps

The type of keys and values in a map can be defined. The following example demonstrates the definition of a map with int64 keys but variant values of all types.

```
keyTypeMap = map of int64 | variant : doc "my type defintion";
```

3.8 Defining variant types

The type `variant` is a wildcard for any types available, not matter if atomic, custom or container. Sometimes a data definition requires the flexibility of a variant, but still needs to be limited to a subset of types.

```
numeric = variant of [ int64, int32, int16, float64, float32, float16 ]
    : doc "a constraint variant type";
```

A documentation for each type can be added.

```
numeric = variant of [
        int64 : doc "large int numbers",
        int32 : doc "save memory for smaller numbers",
        int16 : doc "save more memory for smalle numbers",
        float64 : doc "allow large float",
        float32 : doc "save mem for smaller float",
        float16 : doc "you might want to save even more memory"]
    : doc "a constraint variant type";
```

Other constraints can be added likewise.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search