



Configuration Object Generation System

Brett Viren

June 28, 2020

Some requirements for a configuration system

Configuration information:

- must be **valid**
 - ▶ well defined structure
 - ▶ constraints on values
 - ▶ valid-by-construction patterns
 - ▶ centralized validation methods
- is needed in **multiple contexts**
 - ▶ contract between producers and consumers
 - ▶ authoring, displaying, storing, serializing, native code types
- must support **varied and changing forms**
 - ▶ many types of applications and services
 - ▶ some common “base” for implementing “roles”
 - ▶ application- and instance-specific variety
 - ▶ evolution of structure and values over time

The cogs approach

schema

Define formal **schema** to describe structure and constraints.

codegen

Generate code to validate, produce, transport and consume configuration.

correctness

Enable the pattern “single source of truth” (SSOT).



automate

Minimize human effort and the chaos it brings.

a **schema** is a **data structure**
which may be **interpreted**
as **describing** the **structure of data**
(*including that of schema!*)

Categories of schema interpretation

- `translate(schema) → schema`
- `codegen(schema, template) → code`
- `validate(schema, data) → true | false`

These functions are largely provided to  the  tool.

Defining schema



⚙️cogs⚙️ supports authoring schema with functions of an **abstract base schema** in the **Jsonnet** data templating language¹.

```
function(schema) {  
  types: [schema.string(pattern="^[a-zA-Z][a-zA-Z0-9_]*$")],  
}
```

- The resulting abstract *application-level* schema defines a **string type** taking a **valid value** that must match the given pattern.
- When a concrete schema object from some **schema domain** is provided it will result in a concrete schema.

¹⚙️cogs⚙️ (via moo) also supports defining schema in other languages (JSON, YAML, INI, XML or languages that generate these) but these lack support for the abstract base schema.

Larger Schema Example

Describe the configuration for a “node” from the   demo.

```
function(schema) {  
  // ... other locals ...  
  
  local node = schema.record("Node", fields=[  
    schema.field("ident", ident,  
      doc="Identify the node instance"),  
    schema.field("portdefs", schema.sequence("Port"),  
      doc="Define ports used by components"),  
    schema.field("compdefs", schema.sequence("Comp"),  
      doc="Describe components needing ports"),  
  ], doc="A node configures ports and components"),  
  
  types: [ ltype, link, port, comp, node ],  
}
```

Abstract base schema

```
function(schema) {  
  types: [schema.string(pattern="^[a-zA-Z][a-zA-Z0-9_]*$")],  
}
```

The schema object is used as a OO “abstract base class” instance to define abstract *application-level* schema.

⚙️cogs⚙️ includes these concrete **domain schema**:

- `avro-schema.jsonnet` for codegen with **Avro CPP** and `moo` using serialization provided by **nlohmann::json**.
- `json-schema.jsonnet` for object validation via **JSON Schema** and `moo`.

Possible future domain schema:

- support for Protobuf / Cap’N Proto schema depending on RPC choices.
- Jsonnet functions for valid-by-construction configuration authoring.

moo

...provides a **Python3 CLI and module** for processing of schema defined in Jsonnet, JSON, XML, YAML, INI, etc, validation of objects in the same languages and template-based file generation using **Jinja2**.

```
$ moo --help
```

```
Usage: moo [OPTIONS] COMMAND [ARGS]...
```

moo command line interface

Options:

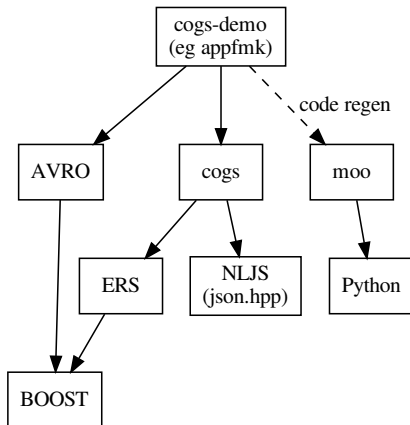
`--help` Show this message and exit.

Commands:

<code>compile</code>	Compile a model to JSON
<code>imports</code>	Emit a list of imports required by the model
<code>many</code>	Render many files
<code>render</code>	Render a template against a model.
<code>render-many</code>	Render many files for a project.
<code>validate</code>	Validate a model against a schema

moo essentially replaces a large set of other tools (jsonnet, jq, j2, grep, awk, etc) and the shell scripting that would be needed to

⚙️cogs⚙️ package dependencies



⚙️cogs⚙️ package features

configuration stream methods for **deserialization** of configuration objects from multiple sources and formats.

configurable base an abstract base mixin class for user code to receive **dynamically or statically typed** configuration objects.

tech opinions ERS for exceptions, `nlohmann::json` for dynamic typed intermediate data representation, Avro for C++ config struct types.

non-trivial demo component-based mocked framework and main application demonstrating ⚙️cogs⚙️ ([link to doc](#)).

Some current elements of demo are general and will be factored into ⚙️cogs⚙️ or moo.

Choice of Avro being revisited and may be influenced by choice of RPC.

cogs configuration stream

A configuration is delivered as an ordered sequence (stream) of objects.

```
std::string uri = "....";  
stream_p s = cogs::make_stream(uri);  
cogs::object o = s->pop();
```

- The `make_stream()` factory returns steam based on parsing URI.
- The returned `unique_ptr<cogs::Stream>` is abstract.
- `cogs::object` is a typedef for `nlohmann::json` and provides a dynamic typed intermediate data representation layer.
- Exceptions defined by ERS may be thrown if stream is corrupt or an attempt is made to `pop()` past its end.

URLs with built-in support:

`file://config.json` a JSON array of configuration objects

`file://config.jstream` A **JSON Stream** of configuration objects

Potential future stream types URLs:

- Files via `https://` addressing.
- RPC server address (eg, hardwired host/port)
- ZeroMQ/ZIO port spec (eg, direct or auto-discovered address)
- An upgraded factory would allow streams provided by plugins.

⚙️cogs⚙️ delivery of configuration to component

An application component receives configuration object by inheriting from a **virtual mixin** class and implementing the method:

A **dynamic typed** interface

The user code must interpret a **dynamic object**.

```
struct ConfigurableBase {  
    virtual void configure(cogs::object obj) = 0;  
};
```

A **static typed** interface

The user code receives C++ struct generated from **schema**, eg via Avro.

```
template<class CfgObj>  
struct Configurable : virtual public ConfigurableBase {  
    virtual void configure(CfgObj&& cfgobj) = 0;  
};
```

⚙️cogs⚙️ demo stream

The ⚙️cogs⚙️ **demo stream** assumes a pair-wise ordering:

component 1: democfg :: ConfigHeader
component 1: corresponding config object
...
component N: democfg :: ConfigHeader
component N: corresponding config object

Each pair:

header identifies a component **implementation** and **instance** name

payload provides config object for the identified component

⚙️cogs⚙️ supports other choices for stream-level interpretation.

Demo stream model and schema

```
model: [  
  head("demoSource", "mycomp_source1"),  
  source(42),  
  
  head("demoNode", "mynode_inst1"),  
  node("mynode1",  
    ports=[portdef("src",[  
      link("bind","tcp://127.0.0.1:5678")])],  
    comps=[compdef("mycomp_source1", "demoSource", ["src"])]  
  ],  
schema: [  
  schema.head,  
  schema.comp,  
  schema.head,  
  schema.node,  
],
```

Details on schema array next.

Demo stream schema (more)

JSON Schema requires types to be defined in a special location in the structure. The `compound()` function helps prepare that.

```
local jscm = import "json-schema.jsonnet";
local compound(types, top=null) = {
    ret : {
        definitions: {[t._name]:t for t in types}
    } + if std.type(top) == "null"
    then types[std.length(types)-1]
    else top,
}.ret;

local schema = {
    head: compound(head_schema(jscm).types),
    comp: compound(comp_schema(jscm).types),
    node: compound(node_schema(jscm).types),
};
```



Perform validation

The moo tool can dig into arbitrary Jsonnet for schema and model and perform validation on a single object (default) or on an array (`--sequence`):

```
$ moo validate --sequence \  
  -S schema -s demo/demo-config.jsonnet \  
  -D model demo/demo-config.jsonnet
```

Currently returns `null` for success and traceback into the data structure if showing where the model fails to validate.

Some work still needed for DUNE FD DAQ

- The concrete domain schema (JSON Schema and Avro schema) are general and should be moved into moo.
- Improve how to create application-level JSON Schema in Jsonnet.
- Integration with DUNE FD DAQ appfmk may include
 - ▶ use of appfmk factory to construct streams, locate components
 - ▶ provision of a configuration dispatch method
- A choice of RPC technology for larger CCM may influence replacement of Avro (eg with Protobuf, Cap'n Proto, etc).
- Understand if cogs and moo approach can help with connecting CCM RPC to appfmk.
- Understand larger configuration issues (authoring, version control, schema evolution, wholesale validation).