YAML file handling in JEDI

Wojciech Śmigaj

April 9, 2020

Outline

Met Office

Parameter encapsulation

YAML validation using JSON Schema

Outline

Met Office

Parameter encapsulation

YAML validation using JSON Schema

Motivation

Met Office

The standard way of retrieving configuration options set in YAML files is to call `eckit::Configuration` methods in member functions of the class to be configured.

```
DifferenceCheck::DifferenceCheck(
  const eckit::Configuration &config, /*other params*/)
  : FilterBase(config, /*other params*/),
    ref_(config_.getString("reference")),
    val_(config_.getString("value"))
{}

void DifferenceCheck::applyFilter(/*parameters*/) const {
  float vmin = config_.getFloat("minvalue", missing);
  float vmax = config_.getFloat("maxvalue", missing);
  // ...
}
```

Disadvantages:

- Users need to read the code to find the list of available parameters.
- Documentation is separated from parameter definition.
- Unrecognised (e.g. mistyped) options are simply ignored.

Alternative

Met Office

`oops::Parameters`
: A collection of parameters.

`oops::RequiredParameter<T>`
: A parameter whose value must be specified explicitly.

`oops::Parameter<T>`
: A parameter with a default value.

`oops::OptionalParameter<T>`
: A optional parameter whose absence is handled specially.

Parameter definition

Met Office

```
/// Options controlling the operation of a thinning filter.
class ThinningParameters : public oops::Parameters {
 public:
  /// Minimum distance (in km) between two retained observations.
  oops::RequiredParameter<float> minDistance{
    "min_distance", this};

  /// If true, observations will be randomly shuffled before
  /// being inspected as candidates for retaining.
  oops::Parameter<bool> shuffle{
    "shuffle", true, this};

  /// Variable storing observation priorities. An observation
  /// won't be retained if it lies within the exclusion volume
  /// of an observation with a higher priority.
  ///
  /// If not set, all observations have equal priority.
  oops::OptionalParameter<Variable> priorityVariable{
    "priority_variable", this};
};
```

Parameter usage

Met Office

```
// In the filter class:
ThinningParameters options_;

// In the filter's constructor
// ('config' is an instance of eckit::Configuration):
options_.deserialize(config);

// In the filter's applyFilter() method:
oops::Log::debug() << "Requested minimum distance (km) = "
                   << options_.minDistance << std::endl;
// (Parameter<T> is often implicitly converted to T.
// In contexts where that doesn't happen, call value()).

// Accessing an optional parameter's value:
const boost::optional<Variable> &priorityVariable =
  options_.priorityVariable;
if (priorityVariable != boost::none)
  oops::Log::debug() << "Requested priority variable = "
                     << priorityVariable->variable() << "@"
                     << priorityVariable->group() << std::endl;
```

Parameter hierarchies: use case

Met Office

```
ObsFilters:
  - Filter: Met Office Buddy Check
    traced_boxes:
    - min_latitude:   10
      max_latitude:   20
      min_longitude:  30
      max_longitude:  40
    - min_latitude:  -80
      max_latitude:  -70
      min_longitude: -60
      max_longitude: -50
```

Parameter hierarchies: definition

Met Office

```
/// A box covering a specified (closed) interval of
/// latitudes and longitudes.
class LatLonBoxParameters : public oops::Parameters {
 public:
  oops::Parameter<float> minLatitude{"min_latitude", -90, this};
  oops::Parameter<float> maxLatitude{"max_latitude",  90, this};
  oops::Parameter<float> minLongitude{"min_longitude", -180, this};
  oops::Parameter<float> maxLongitude{"max_longitude",  180, this};

  bool contains(float latitude, float longitude) const;
};

/// Options controlling the MetOfficeBuddyCheck filter.
class MetOfficeBuddyCheckParameters : public oops::Parameters {
 public:
  /// Information about observations lying within any of
  /// the specified boxes will be output to the log.
  oops::Parameter<std::vector<LatLonBoxParameters>> tracedBoxes{
    "traced_boxes", {}, this};
};
```

Parameter hierarchies: usage

Met Office

```
// In the filter class:
MetOfficeBuddyCheckParameters options_;

// In the filter's constructor:
options_.deserialize(config); // all levels will be deserialized

// In the filter's applyFilter() method:
const std::vector<LatLonBoxParameters> &tracedBoxes =
  options_.tracedBoxes.value();
const float obsLatitude = ..., obsLongitude = ...;
if (std::any_of(
      tracedBoxes.begin(), tracedBoxes.end(),
      [&](const LatLonBoxParameters &box)
      { return box.contains(obsLatitude, obsLongitude); })) {
  // print observation to the log
}
```

How this works

Met Office

- `Parameters` stores a vector of pointers to its constituent `(Required/Optional)Parameter<T>`s.
- `Parameters::deserialize()` calls the `deserialize()` method of each constituent `(Required/Optional)Parameter<T>`.
- That method calls `ParameterTraits<T>::get()`, which defines how a value of type `T` should be extracted from an `eckit::Configuration` object.

Generic impl. of ParameterTraits

Met Office

```
template <typename T>
struct ParameterTraits {
  static boost::optional<T> get(
    const eckit::Configuration &config,
    const std::string &name)
  {
    T value;
    if (config.get(name, value))
      return value;
    else
      return boost::none;
  }
};
```

Specialisation of ParameterTraits

Met Office

```
template <>
struct ParameterTraits<util::DateTime> {
  static boost::optional<util::DateTime> get(
    const eckit::Configuration &config,
    const std::string &name)
  {
    std::string value;  // T value;
    if (config.get(name, value))
      return util::DateTime(value); // return value;
    else
      return boost::none;
  }
};
```

Existing specialisations

Met Office

- `util::DateTime`
- `util::Duration`
- `ufo::Variable`
- `std::vector<T>`
- `std::map<Key, Value>`
- `util::ScalarOrMap<Key, Value>`
- Any subclass of `oops::Parameters`

Outlook

Met Office

- Detection of unused/mistyped parameters (August Weinbren): `Parameter` objects notify their parent (`Parameters`) about the configuration entries they've used; `Parameters` displays a warning if any entries were left unused.
- Range checking:

```
oops::BoundedParameter<int> numSteps{
  "num_steps", 10, this,
  // minimum allowed value
  1,
  // maximum allowed value
  std::numeric_limits<int>::max()};
```

- More documentation:
  `https://jointcenterforsatellitedataassimilation-jedi-docs.readthedocs-hosted.com/en/latest/jedi-components/ufo/parameters.html`

Outline

Met Office

JSON Schema

Met Office

- A JSON schema is a JSON document that describes the expected structure of other JSON documents using certain keywords.
- These keywords are defined in drafts of the JSON Schema standard published on `http://json-schema.org`.
- JSON documents can be validated against a schema using a variety of libraries, command-line and GUI tools.
- YAML documents can be handled by converting them into JSON beforehand. Some tools do it automatically.

(With thanks to Stephen Oxley)

Example

Met Office

YAML document

```yaml
name: Paris
latitude:  48.9
longitude:  2.4
```

Schema

```
{
  "type": "object",
  "properties": {
    "name":      {"type": "string"},
    "latitude":  {"type": "number",
                  "minimum": -90,
                  "maximum":  90},
    "longitude": {"type": "number",
                  "minimum": -180,
                  "maximum":  180}
  },
  "required": ["latitude",
               "longitude"],
  "additionalProperties": false
}
```

Example implementations

Met Office

- JSON schema validator for JSON for modern C++ (`https://github.com/pboettch/json-schema-validator`)
- The `jsonschema` Python package (`https://python-jsonschema.readthedocs.io`)
- Visual Studio Code with the *YAML* plugin (developed by Red Hat)

Demo: Editing a JEDI YAML file in Visual Studio Code.

Challenges

Met Office

1. Each executable (e.g. `Variational`, `HofX`) expects the YAML configuration file to have a different structure.
   - We may not support syntax checking for each one.
   - A separate top-level JSON Schema file must exist for each supported type of configuration files.
   - Schemas describing parts of YAML files shared by multiple executables (e.g. the `Model` section) can be defined in separate JSON files and referenced from the top-level schema:

     ```
     "Model": {"$ref": "Model.json.schema#/Model"}
     ```

Challenges

Met Office

2. The list of allowed properties may depend on the values of certain properties. Example:

```
- Filter: Thinning
  random_seed: 123
  amount: 0.96
- Filter: Background Check
  threshold: 3.0
```

Challenges

Met Office

2. The list of allowed properties may depend on the values of certain properties. Example:

```
- Filter: Thinning
  random_seed: 123
  amount: 0.96
- Filter: Background Check
  threshold: 3.0
```

- JSON Schema supports conditional subschemas through the `allOf` and `if-then` keywords. *Simplified* example:

```
"allOf": [
  {
    "if": { "properties": {
        "Filter": { "const": "Thinning" }
        } },
    "then": { "properties": {
        "random_seed": { "type": "integer" },
        "amount": { "type": "number" }
    } }
  },
  {
    "if": { "properties": {
        "Filter": { "const": "Background Check" }
    } },
    "then": { "properties": {
        "threshold": { "type": "number" }
    } }
  }
]
```

Challenges

Met Office

3. Parts of the YAML structure may depend on the model.
   - QG:

```
model:
  name: QG
cost_function:
  cost_type: 3D-Var
    Jb:
      Background:
        state:
        - date: 2010-01-01T12:00:00Z
          filename: Data/fcast.fc.2009-12-31T00:00:00Z.P1DT12H
```

   - FV3:

```
model:
  name: FV3
cost_function:
  cost_type: 3D-Var
    Jb:
      Background:
        state:
        - filetype: geos
          datapath: Data/inputs/geos_c12
          filename_bkgd: geos.bkg.20180415_000000z.nc4
```

Challenges

Met Office

- We could use a solution based on `allOf/if/then`, but the top-level schema, defined in OOPS, cannot know about all possible model types. So…
- OOPS schemas import model-dependent subschemas from the (non-existing) `model` folder:

```
"state": {
  "type": "array",
  "items": {
    "$ref": "../../model/schemas/state.schema.json#/state"
  }
}
```

- Each model provides the missing files and uses CMake to copy them together with those from OOPS into the build and installation folders.

```
build
‘- fv3-jedi
   ‘- model
   |  ‘- schemas
   |     ‘- state.schema.json
   ‘- oops
      ‘- schemas
         ‘- variational.schema.json
```

- Users configure VS Code to pick the top-level schema corresponding to their model of choice in the build or installation folder.

Challenges

Met Office

4. JSON schemas become an independent source of truth about the structure of configuration files.
   - We could parse JSON schemas and generate C++ source files defining data structures storing settings loaded from YAML files.
   - Alternatively, we may be able to generate JSON schemas from C++.

Source code

Met Office

- JSON Schema files used during the demonstration are available on the `feature/json-schema-variational` branch in the `oops`, `ioda`, `ufo` and `fv3-jedi` repositories.
- To test the schema in VS Code, run `make install` to copy all parts of the schema to the installation directory. Then add the following line to the `yaml.schemas` property in VS Code's `settings.json` configuration file:

```
"your-installation-dir/share/fv3jedi/schemas/oops/
  schemas/variational.schema.json": "3dvar*.yaml"
```