Framework Configuration

discussion toward requirements

Brett Viren

December 11, 2023

Overview

This presentation:

- poses a few **problems** in **configuring** the offline "**framework**"
- and suggests some **solutions** for **discussion**.

Some assumptions / realities

Consider here "the framework" broadly as:

> *Software structures to **compose** and **execute** a **process** built of **components**.*

Users wish to **configure** the **process behavior**.

- Includes both the individual **components** and their **composition**.

Composition is **hierarchical**:

- **...** of **campaigns** of **stages** of **jobs** of **algorithms** of **tools** of **...**

This spans multiple software systems/libraries.

- That presents a **variety of configuration** mechanisms to deal with.

Reducing that complexity is one of the goals.

Problem: configuring components

A configurable component must **consume a valid config object**.

Assuring config syntax is **valid under the language model** is **not** a problem.

Any reasonable parser will produce a valid **dynamic object representation** of the syntax.

- …or it will raise a syntax error.

Assuring the object is **valid under an object schema** is a bigger problem.

Every developer invents their own object schema.
The component may determine validity by carefully **interpreting** the configuration object.

- Requires tedious, error prone and often omitted coding.

How can "the framework" help the developer?

Define system for **configuration schema**.

- Two strategies, next slide ⟶

Configuration schema encoding strategies

Define schema in the C++ "next to" component implementation (eg, a'la *art*)

Pro: Configuration schema and its consuming code can be in same source file.
Con: Validation must execute "the framework", hard to use schema in other contexts. Where optional, usually omitted. Developer implements a variety of patterns, bugs.

Define schema outside of the C++ component

Pro: enables many powerful features

- Adds out-process validation and thus more general across the **composition hiearchy**.
- Multiple consumers of schema: valid-by-construction config authoring, safe config object transformations, documentation generation, …
- Cornerstone of a **comprehensive configuration management system** spanning the hierarchy.

Con: Must still provide some C++ boilerplate

- But, be automated away with **schema-driven codegen**.

Problem: configuring composition

No problem?

On one hand, some component does the composition. That component gets some configuration.

But, composition requires knowing “meta” information

- A graph is formed between producer and consumer components.
- Cardinality of input and output “ports” of each graph node must be known.
- The data type(s) that may pass through each port must be known.

Similar solution strategies

Run time validation or allow external schema/validation/codegen.

What is **codegen**?

Apply an **object** to a **template** to generate a **document** (code file).

Example usage of **codegen**

- Generate code implementing a pattern parameterized by data types.
- Generated code is **perfect** (ie, fix a bug in the template, fix all code).
- Enables **valid-by-construction** configuration authoring pattern.
- Config syntax converters (*eg*, between JSON/FHiCL/INI/Geant4/etc).
- Derive JSON Schema schema to exploit existing data validation libraries.

Codegen systems

- moo provides a generalized schema syntax based on Jsonnet, convertible to JSON Schema and supports arbitrary code generation from models expressed in Jsonnet, JSON and many other data languages using Jinja2 templates via simple CLI and modules in Python. Also supports state machines, build systems and software project skeleton generation.
- ZeroMQ zproject and zproto projects, heavily influenced **moo** and has most of the same features while using a custom template language (GSL) and supports one data language (XML).
- Serialization (ProtoBuf, Cap'N'Proto, Avro).
- Cookiecutter for software project skeleton generation.

Problem: a DUNE FD “job” is really a “campaign of jobs”

The problem:

The size of DUNE FD data requires multiple jobs to process each trigger record.

- Especially true for first-stage data processing.
- Must configure each job and the ensemble **consistently**.

A solution:

Express overall configuration in a unified form and use **codegen** to produce job-level and campaign-level configuration files.

- Factor that overall configuration into high-level configuration and generators
- Codify schema of high-level config, (re)use documentation generators.
- Track and association high-level config and generator templates with campaigns for reproducible results.

Problem: detector (and beamline) geometry

Here, consider geometry = physical + logical connectivity (eg, “wire/channel maps”).

Some requirements

Support **multiple types of consumers**

- LArG4, Wire-Cell, event displays, human-oriented documentation

Architecture

- Factor out a “high level” parameter set describing a detector,
- Develop a generator to apply detector symmetry patterns to produce an “inflated” form
- Convert and pass that form to the various different consumers.

Currently, DUNE has two methods

A “Perl script” or **GeGeDe** (next slide)

General geometry description (GeGeDe)

Composes a hierarchy of "builders" components.

- Composition itself and the components are configurable (and extensible).
- Each builder constructs some portion of the geometry.
  - A small Python class configured via Python INI files.
- In concert, they construct an in-memory object following well defined schema.
  - The schema intentionally mimics that of Geant4 volumes and solids.
- Object exported to JSON, GDML, ROOT, AGDD and other file formats.

Repos: gegede, duneggd (fd), dunendggd (nd)

- The `dune*ggd` packages[a] provide high-level config and DUNE-specific builders.

[a] `https://github.com/dune/duneggd` `https://github.com/dune/dunendggd`

Geometry work

Basic work

- Port remaining “Perl script” to a GeGeDe description, deprecate the “Perl script”.
- Develop new GeGeDe export forms (eg for WCT “wires file”)
- Implement beam simulation geometry in GeGeDe.

Advanced work

- Codify the schema describing the high-level config inputs to `dune*ggd`.
  - Use for validation and to develop templates to generate documentation.
- Centrally manage `dune*ggd` config and builder code.
  - Perhaps unified git repo with `git submodule` links.
  - Associate git tag with offline releases, install with release.
  - Add to build/CI: validate high-level config, generate required products.

Summary

- DUNE has a large, heterogeneous ecosystem to configure.
  - Large enough that **Offline Configuration Group** may be needed?
- Individual components and their composition require configuration.
- Abstraction will simplify code and enable consolidation.
  - Models (schema) applied to templates can generate perfect code that otherwise humans must reinvent again and again.