Cascale

HIGG ASSESSMENT MODEL

Using the `model.json.gz` file

Version next · 11 September 2026

This document explains the structure of the model.json.gz file produced by the stl-local-release crate and how to use jq to query it.

Contents

This document explains the structure of the `model.json.gz` file produced by the `stl-local-release` crate and how to use `jq` to query it.

The `model.json.gz` file is a compressed JSON file that contains the compiled results of the STL files. After decompressing it, you will get a JSON file with the following structure:

```
{
  "schema": { ... },
  "blocks": { ... }
}
```

The `schema` attribute

The `schema` attribute contains a JSON schema that describes the structure of the blocks. The schema is generated from the `Type` definitions in the STL files. It defines the different types of blocks and their fields.

The schema is organized as follows:

- `$defs`: This section contains the definitions of all the block types. Each type definition includes:
  - `type`: "object"
  - `description`: A description of the block type, extracted from comments in the STL file.
  - `properties`: An object describing the fields of the block type. Each field is an object that can contain standard JSON schema keywords (`type`, `description`, `format`, etc.) and a special non-standard keyword `evaluatedType`.
  - `required`: An array of required field names.

The `properties` object and non-standard fields

Each key in the `properties` object corresponds to a field in a block type. The value is a JSON schema object describing that field.

For most fields, this will be a standard JSON schema object. For example:

```
"my_field": {
  "type": "string",
  "description": "This is a simple text field."
}
```

However, to fully describe the behavior of STL fields, several non-standard properties are added to the schema: `evaluatedType`, `keep`, and `unnest`.

`evaluatedType`

Some fields in the STL language are *interpreted*. This means their final value is the result of executing a script (e.g., a Javascript expression). In the `output_data.json` file, the value of such a field will be the script itself (as a string). The JSON schema for these fields includes a non-standard `evaluatedType` property to inform you of the expected type of the data *after* the script is executed.

Here is an example of a field with an `evaluatedType`:

```
"some_js_field": {
  "type": "string",
  "format": "javascript",
  "description": "A field that is a Javascript expression.",
  "evaluatedType": {
```

```
    "type": "number"
  }
}
```

In this example:

- `type: "string"` indicates that the value in the `blocks` data is a string.
- `format: "javascript"` provides a hint that the string is a Javascript expression.
- `evaluatedType: { "type": "number" }` tells you that if you were to execute this Javascript expression, the result should be a number.

This `evaluatedType` is crucial for understanding the *semantic* meaning of the data and for correctly processing the `blocks` if you intend to evaluate the embedded scripts.

`keep`
The `keep` field is a boolean that indicates that the block should be kept only if this field has a "truthy" value after evaluation. This is used to filter out blocks that are not relevant based on the evaluation context. See the "Keep Logic" section under "Block Evaluation" for more details.

`unnest`
The `unnest` field is a boolean that changes how a field's value is stored in the evaluation context. When `unnest` is `true`, the value of the field is placed directly into the context of the parent block, rather than being nested under the field's name. This is a mechanism for promoting a field's value to a more accessible scope.

You can use the main `schema` to validate the blocks or to understand the structure of the data.

The `blocks` attribute

The `blocks` attribute is a map where the keys are RFI PIDs (e.g., "fem2025") and the values are arrays of compiled blocks for that RFI PID.

Each block in the array is a JSON object. The keys of the object are the fields of the block, and the values are strings. Each block also contains two special fields:

- `__name__`: The name of the block.
- `__type__`: The type of the block, which corresponds to a type definition in the `schema`.

Here is an example of a block:

```
{
  "__name__": "some_block_name",
  "__type__": "SomeBlockType",
  "field1": "value1",
  "field2": "value2",
  ...
}
```

Querying the `blocks` with `jq`

The `model.json` file can be large, so using a command-line tool like `jq` is recommended for exploring and filtering the data.

First, you need to decompress the file:

```
gunzip model.json.gz
```

Here are some examples of how to use jq to query the blocks:

1. List all block types for a specific RFI PID

This command will list all unique block types for fem2025:

```
jq '.blocks.fem2025[].__type__' model.json | sort | uniq
```

2. Get all blocks of a specific type

This command will retrieve all blocks of type EnergyUse for fem2025:

```
jq '.blocks.fem2025[] | select(.__type__ == "EnergyUse")' model.json
```

3. Get specific fields from blocks of a certain type

This command will retrieve the __name__, value, and provenance for all EnergyUse blocks for fem2025:

```
jq '.blocks.fem2025[] | select(.__type__ == "EnergyUse") | {name: .__name__, value: .value,
provenance: .provenance}' model.json
```

4. Filter blocks based on a field's value

This command will retrieve all EnergyUse blocks where the provenance is "reported":

```
jq '.blocks.fem2025[] | select(.__type__ == "EnergyUse" and .provenance == "reported")'
model.json
```

5. Searching across all RFI PIDs

You can also search across all RFI PIDs. This command finds all ReportedEnergyUse blocks regardless of the RFI PID.

```
jq '.blocks[][][] | select(.__type__ == "ReportedEnergyUse")' model.json
```

These examples should give you a good starting point for exploring the model.json file with jq. You can construct more complex queries to suit your needs. Refer to the jq manual for more information on its features.

Implementing a Block Evaluator

The model.json file provides all the necessary information to build a custom evaluator for the STL blocks. An evaluator's main purpose is to take the blocks and an optional external context, execute the interpreted fields (the Javascript expressions), and produce a final JSON object with the computed values.

Here is a guide to the high-level steps required to implement a successful evaluator:

1. **Parse Interpreted Fields for Dependencies**: The first step is to determine the dependencies between all the interpreted fields. An interpreted field depends on another field if it references it in its expression. For example, in a block named my_block, if a field a has the expression this.b * 2,

then `a` depends on `b` within the same block. If the expression is `other_block.c + 5`, then `a` depends on field `c` of `other_block`. Your parser needs to identify these dependencies to build a dependency graph.

2. **Topologically Sort the Fields**: Once you have the dependency graph, you must perform a topological sort on all the interpreted fields. This will give you a linear evaluation order, ensuring that no field is evaluated before its dependencies have been evaluated. This is a critical step to guarantee a correct evaluation.
3. **Set up an Execution Environment**: Most interpreted fields are Javascript expressions. Therefore, you will need to set up a Javascript execution environment or engine. You will also need to provide some helper functions that may be expected by the expressions (see `stl-eval/src/functions.js` for examples).
4. **Prepare the Evaluation Context**: Before starting the evaluation, you should prepare a context object. This object will hold the values of all the fields. It should be pre-populated with all the non-interpreted (static) values from the blocks. You should also merge any external context provided at runtime into this main context.
5. **Evaluate Expressions in Order**: Iterate through the topologically sorted list of fields and execute the Javascript expression for each one. The result of each evaluation should be stored back into the context object, making it available for subsequent expressions that depend on it. A common way to implement this in Javascript is to use a `with (context)` block, which allows expressions to reference other values in the context directly.
6. **Handle `keep` and `unnest` Logic**:
   - **`keep`**: After evaluating all fields, you need to handle the `keep` logic. For each block, check if any of its fields have the `keep` property set to `true` in the schema. If so, check if the evaluated value of that field is "truthy". If any of the `keep` fields in a block have a "falsy" value (e.g., `false`, `0`, `""`, `null`, `undefined`), the entire block should be removed from the final result.
   - **`unnest`**: During evaluation, if a field has `unnest: true` in the schema, its value should be placed directly in the parent block's context, rather than being nested under its own field name. This can affect how dependencies are resolved.
7. **Produce the Final Result**: The final result of the evaluation should be a JSON object representing the evaluated blocks, with all the interpreted fields now containing their computed values.

For a concrete example of how such an evaluator is implemented, you can refer to the `stl-eval` crate in this repository, which follows these principles. The Rust code in `stl-eval/src/v8.rs` orchestrates the process, and the core Javascript logic can be found in `stl-eval/src/evaluate.js`.