remaining

- Can workflow / step / process inputs be optional in each language?
- If so, how is this implemented?

Overview

This document aims to gather information to answer the following questions about our software design:

1. What should our internal model look like?
2. Where will the business logic appear?
3. How will translate units work with our model?
4. How would we extend our code base to support a new feature or language?

TLDR

Our code should consist of multiple parts that form a system.
The main components are the:

- ***Model*** - Represents a workflow in a generic fashion.
- ***Modules*** - Provide functionality related to ingest, model manipulation, or translation.
- ***Translation Units*** - Map workflow information from one specification to another.

Internal Model
Our internal model should be a generalisation. It should be as ***simple*** as possible and built from the most primative entities. It should segregated so that core entities do not need to change when a new language is supported as possible.

Business Logic
Business logic will appear within the translate units and modules. The model should not contain any business logic. Each translate unit will use its own process and functionality provided by the modules to perform a translation.

Translation Units <-> Model
The model should allow any information to be requested or accessed at any time. For an ingestion unit, we should be able to add any new information about the workflow at any time. This implies that our model does not guarantee dependent entities exist when referenced. For example, we could add a connection from a step called 'fastqc' to a step called 'BWA' even if the 'fastqc' step does not yet exist. The Model should provide methods to add new information, where some functionality may occur to guarantee the promises it ***does*** make.

New Feature / Language
When adding a new language, the model should not need alteration (ideally). New translate units will be written which map the data of that workflow language in and out of our Model. The functionality of some Modules may be extended, and new modules may need to be created.

To avoid model alteration, model entities should be as primitive as possible. Our model should have the fewest number of entities that can describe the features of our supported languages. Entities will combine to build the higher-level features of other languages (eg nextflow channels).

If our Model can cover Nextflow, Galaxy, Janis, CWL using a minimal set of primative entities, it will be uncommon to identify a new primative that needs to be added when supporting a new language.

This is fine in theory, but will likely not hold in practise. Keeping the model ***segregated*** should be a large focus. Segregation would mean moving operations / manipulations away from the data, and reducing the coupling between entities (minimal dependence on each other). This will allow us to change the model more easily in future, as reduces the pollution caused when an entity must be altered or added

Contents

Translation Units

Translate units reformat a source workflow into a destination output.

- **Ingestion** - supported lang -> internal
- **Translation** - internal -> supported lang

Ingestion Questions:

1. When and how can we add information from the source workflow to our internal model?
2. What promises does the system make?
3. What do we do when encountering something currently unsupported?
4. High vs low level implementations of features (ie builtin functions vs scripted) - general high (source) -> low (dest) but this goes against best practises.

Interacting with the Model

- System should provide an interface
    - Should be defined methods to add a step, add a connection etc
    - When these are added, system will perform internal actions so advertised behaviour is correct
- System should allow new information to be incorporated from anywhere at any time
    - For example: adding a connection between step1 and step2
    - Even if step1 has not yet been added to internal workflow, should still be able to register this connection
- System should provide promises
    - For example: tags
    - Each entity is guaranteed to have a unique tag
    - If the supplied tag is unique, then this should be preserved.
    - If the supplied tag is duplicated somewhere else, the system should provide new tags for these entities.

Modules

Notes Want to minimise unused or deprecated modules. Should leverage common and reusable modules as opposed to one-off solutions.

Ingestion - Holds each ingestion translation unit (high-level)

Translation - Holds each output translation unit (high-level)

Mapping - Provides functions which transform entities between specifications (factory)

Expressions - Holds regex patterns and search utils used throughout ingestion

Datatypes - Methods to supply entities with best-guess datatypes

Tags - Formats and sanitises entity tags. Promises each entity has unique tag.

Logs - Provides methods to log information & retreive that specific info later.

Messages - Pulls information from logs and embeds comments in output files for user.

FileIO - Reads & writes workflow files

Rendering - Defines how output files appear and are rendered from in-memory representations.

Containers

- Caches information about known containers (software requirement encapsulated, version, download url)
- Allows searching for a suitable container using the local cache
- Allows searching for a suitable container using online API requests

Model

Our internal model should be a generalisation. It should be as simple as possible and built from the most primative entities. It should segregated so that core entities do not need to change when a new language is supported as possible. It should be general enough that it can act as a lego kit which can build the higher-level features of other languages.

An example

Workflow specs often have similar concepts with some differing features. Rather than having a `Connection` entity as base class, then a `DataLinkConnection` to support CWL `data links` and a `NextflowChannelConnection` to support Nextflow `Channels`, we should have a single concept which generalises the two.

Here are the differences:

- A CWL `data link` allows multiple inputs and outputs, but does not permit operations.
- A Nextflow `Channel` supports a single input and output, as well as operations on the channel.

To generalise these concepts, we could make a `Connection` entity. One possible approach is the following:

```
class Connection:
    inputs: list[str]           # some ids which references each source
    outputs: list[str]          # some ids which references each
destination
    operations: list[Operation] # operations to perform on the data
```

The above is ok, but it can be simpler.

```
class Connection:
    source: str     # id which references another model entity (the source)
    dest: str       # id which references another model entity (the dest)
```

A CWL `data link` can then be stored as:

- 1+ connections

A Nextflow `Channel` can then be stored as:

- an input connection
- a step (using a 'groovy tool') or equiv
- an output connection

Properties of our model entities

Having a more ***simple*** model forces us to generalise. We should aim for the fewest entities possible, with the least number of attributes. This avoids doubling up and makes extension easier. This would also be more approachable for new contributors.

Notes:

- Entities should combine to create higher-order features of supported languages
- Entities will have a set of core (mandatory) attributes common to all specifications
- Entities will have a set of optional attributes which cover legitimately optional attrs, as well as attrs present in some specs but not others.

Extending Model for future languages

Keeping the model ***segregated*** allows us to move operations / manipulations away from the data. This reduces coupling, allowing us to change the model more easily in future.

- when adding a new language, will the model change, or will only the translate units change?
- What to do when incoming language has new feature?
- How to add a new feature across supported languages

Evolution of Janis pre/post WDL

```
class InputNode:
    identifier
    wf              <-- added
    label           <-- removed
    doc
    value
    default         <-- added
    datatype
```

Draft Model

- TASK
- WORKFLOW
- WORKFLOW INPUT
- WORKFLOW OUTPUT
- STEP
- PROCESS TODO TODAY HERE
- PROCESS INPUT
- PROCESS OUTPUT
- CONNECTION
- REQUIREMENT
- DATATYPE
- OPERATOR
- EXPRESSION
- CONFIG
- NAMESPACE

TASK

NOTE - This is an abstract entity.
Only used for Typing.
Not really part of the Model.

Generalisation

Core unit of computation.

- Accepts **inputs**, does something, produces **outputs**.
- Has a `namespace`

- Example: Workflow, Process

WORKFLOW

Generalisation

A Task which can invoke other Tasks.
Does not execute programs on a command line.

NOTE - subworkflows can be invoked as a task in each language.
No special entity required.

Attributes

```
tag: str                # identifier
inputs: list[WorkflowInput]
outputs: list[WorkflowOutput]
steps: list[Step]
metadata: WorkflowMetadata
```

WorkflowMetadata

- requirements: list[Requirement]
- friendly_name [optional]
- label [optional]
- doc [optional]
- version [optional]
- contributors [optional]
- dateCreated [optional]
- dateUpdated [optional]
- institution [optional]
- doi [optional]
- citation [optional]
- keywords [optional]
- documentationUrl [optional]
- documentation [optional]
- short_documentation [optional]
- sample_input_overrides [optional]

Specification Nuances

- CWL: steps are mandatory in workflow. unsure if this enforces 1+, or just a list.
- Nextflow: the main workflow is denoted as the entry point by omitting a name for the workflow
- Nextflow: subworkflow with inputs / outputs which are not from global variable requires `take main emit` syntax for subworkflow. main workflow can also use `take main emit` structure.
- Nextflow: A process component can be invoked only once in the same workflow context. WHY

WORKFLOW INPUT

Specification	Nomenclature
Galaxy	step['type'] == 'input_data'
Janis	InputNode
CWL	WorkflowInputParameter
Nextflow	NA

Generalisation

A variable which is available throught a workflow.

- Value may be static or resolved at runtime
- Either way, must be resolved before it is used

Attributes

```
tag: str                                    # identifier
datatype: Datatype
datatype_format: Optional[dict[str, Any]]   # 'File' type subsets ??
default: Optional[Any]
value: Optional[Any]
label: Optional[str]
doc: Optional[str]
```

Specification Nuances

- CWL / Janis: datatype attribute is mandatory
- CWL / Janis: has value attribute
- Nextflow: no dedicated WorkflowInput. A workflow component or process can access any variable and parameter defined in the outer scope.

WORKFLOW OUTPUT

Generalisation

A piece of data available to the user upon workflow completion.

Attributes

```
tag: str            # identifier
datatype: Datatype
datatype_format: Optional[dict[str, Any]]   # 'File' type subsets ??
label: Optional[str]
doc: Optional[str]
```

Specification Nuances

- Nextflow:
  - no dedicated WorkflowOutput.
  - outputs are simply the outputs of a workflow component.
  - no datatype for a workflow.out. Probably safe to assume they have `File` type
  - files may be written during runtime and are present in `work` directory. users may fish them out
  - for a subworkflow, subworkflow.outs are often used by other constructs
  - for the main workflow, often there will be a final 'action':

```
NfcoreTemplate.email()
NfcoreTemplate.summary()
workflow.onComplete {
    log.info ( workflow.success ? "\nDone! Open the following report
in your browser --> $params.outdir/multiqc_report.html\n" : "Oops ..
something went wrong" )
}
```

STEP

Specification	Nomenclature
Galaxy	step
Janis	step
CWL	step
Nextflow	NA

Generalisation

An invokation of a Task where input values are supplied.

- The Task to run must be supplied.
- Step Input values may not be exhaustive of Task inputs.
- Step Ouptut values may not be exhaustive of Task outputs.
- Step Input values may be superset of Task inputs

Attributes

```
tag: str                        # identifier
task_id: str                    # task to execute (a workflow or process)
inputs: list[Tuple[str, Any]]   # [input_name, value].  task_id +
input_name can be used to get ProcessInput
outputs: list[str]              # [output_name]
```

```
condition: str                  # a bool expression to determine whether
step should be run (CWL)
scatter: list[str]              # process inputs to scatter
scatter_method: Optional[str]   # method of scattering
label: Optional[str]
doc: Optional[str]
```

Specification Nuances

Janis

- Step inputs == task inputs

Janis / Nextflow / Galaxy

- All task outputs are automatically collected & available on the 'out' attribute of the step

CWL

- Scattering "scatter: [str|array]" defined on step input(s)
- Required - run: [str], in: [array], out: [array]
- ^must specify which task outputs to collect

PROCESS

Specification	Nomenclature
Galaxy	tool
Janis	CommandTool
CWL	CommandLineTool
Nextflow	process
WDL	task

Generalisation

A Task which cannot invoke other Tasks. Executes programs on the command line.

Attributes

```
tag: str                        # identifier
inputs: list[ProcessInput]
outputs: list[ProcessOutput]
params: list[Param]             # non inputs/outputs available to
process.shell
condition: str                  # a bool expression to determine whether
step should be run (nextflow)
shell: str                      # main code to execute in environment, or
```

```
  CWL style base_command + inputs with positions
  metadata: ProcessMetadata       # includes list[Requirement]
```

Specification Nuances

Nextflow

- Requirements (cpus 8, executor 'sge' etc)
- conditional execution of process: `when`
- publishDir '/data/chunks'

PROCESS INPUT

Generalisation

Attributes

```

```

Specification Nuances

PROCESS OUTPUT

Generalisation

Attributes

```

```

Specification Nuances

- Nextflow: outputs available as process.out, process.out[0], process.out.named_output
- Nextflow: When a process defines 2+ output channels they can be accessed using their index or their name if was specified (using `emit`)

CONNECTION

Specification	Nomenclature
Galaxy	inputs
Janis	ConnectionSource?
CWL	data link

Specification	Nomenclature
Nextflow	Channel

Generalisation

Method to dynamically supply data to task inputs once available

Attributes

```
source: str     # identifier
dest: str       # identifier
```

Specification Nuances

CWL
Dependencies between parameters are expressed using the source field on workflow step input parameters and outputSource field on workflow output parameters.

The source field on each workflow step input parameter expresses the data links that contribute to the value of the step input parameter (the "sink"). A workflow step can only begin execution when every data link connected to a step has been fulfilled.

The outputSource field on each workflow step input parameter expresses the data links that contribute to the value of the workflow output parameter (the "sink"). Workflow execution cannot complete successfully until every data link connected to an output parameter has been fulfilled.

OPERATOR

Generalisation

Attributes

```

```

Specification Nuances

DATATYPE

Generalisation

Attributes

```

```

Specification Nuances

REQUIREMENT

Generalisation

A piece of metadata. Some action must be taken regarding the metadata before it is needed. The requirement may or may not be needed to execute a task.

Examples:

- check / configure execution environment (eg container to run task)
- check / configure hardware (eg 8gb mem)
- check / configure software (eg ensure samtools is installed)
- create an #ENV var before running task command

NOTE - Good example here of entity properties. CWL needs `ScatterFeatureRequirement` / `SubworkflowFeatureRequirement`. There is no reason to include these requirements. Reasons:

1. CWL is the only spec with this requirement
    - When ingesting CWL, storing this is redundant because other specs do not have this requirement.
    - We will not be writing it out in the target language, so no need to store.
    - When translating to CWL, this requirement will not be stored as other specs do not have it.
2. The information can be derived
    - If our model has a step with a scatter, we know that `ScatterFeatureRequirement` would be required in the CWL workflow.
    - We know to include `ScatterFeatureRequirement` when writing in CWL, but we never stored this information.

Attributes

```
class Requirement:
    permitted_types = [
        'software',
        'container',
        'environment_variable',
        'cpu',
        'mem',
        ...
    ]

    def __init__(self, type: str, data: dict[str, Any]):
        if type not in self.permitted_types:
            raise ValueError(f'cannot add requirement of type {type}')
        self.type = type
        self.data = data
```

eg software requirement

```
Requirement (
    type='software',
    data={
        'name': 'bwa',
        'version': '1.5.2'
    }
)
```

eg cpu requirement

```
Requirement (
    type='cpu',
    data={ 'value': 8 }
)
```

Specification Nuances

CONFIG

Generalisation

Attributes

```

```

Specification Nuances

Graveyard

CWL Notes

- Seems like the main players are Michael R. Crusoe (project lead), John Chilton & Peter Amstutz
- All 3 are on the CWL leadership team alongside others
- John Chilton (Galaxy project lead) has been involved since inception
- No wonder that Galaxy & CWL have a similar mindset when it comes to workflows / tools
- Unsurprising they want CWL & Galaxy to essentially merge over time
- Michael Franklin was a contributor to CWL v1.2