yMMSL Python bindings Documentation

Release 0.13.0

Lourens Veen

Jan 17, 2023

Contents:

Welcome to the documentation pages for yMMSL, the YAML version of the Multiscale Modeling and Simulation Language. At the moment, yMMSL is mainly the configuration language for the MUSCLE3 multiscale coupling library.

This library provides Python bindings for yMMSL. With it, you can read and write yMMSL files, and manipulate them using a Python object representation of their contents. This documentation gives an overview of the format, and a description of the Python API.

CHAPTER 1

Overview

A yMMSL file is a YAML file that looks approximately like this:

Listing 1: `docs/example.ymmsl`

```
ymmsl_version: v0.1

model:
  name: macro_micro_model
  components:
    macro: my.macro_model
    micro: my.micro_model
  conduits:
    macro.state_out: micro.init_in
    micro.final_out: macro.update_in

settings:
  # Scales
  domain._muscle_grain: 0.01
  domain._muscle_extent: 1.0
  macro._muscle_timestep: 10.0
  macro._muscle_total_time: 1000.0
  micro._muscle_timestep: 0.01
  micro._muscle_total_time: 1.0

  # Global settings
  k: 1.0
  interpolation_method: linear

  # Submodel-specific setting
  micro.d: 2.3

implementations:
  my.macro_model:
    executable: /home/user/model
```

(continues on next page)

(continued from previous page)

```
  my.micro_model:
    modules: gcc openmpi
    execution_model: openmpi
    executable: /home/user/model2

resources:
  macro:
    threads: 1
  micro:
    mpi_processes: 8

checkpoints:
  at_end: true
  simulation_time:
  - every: 50
```

This file describes a macro-micro coupled simulation model with time-scale separation and domain overlap. It describes both the model itself and an experiment to be run with this model, and contains the minimal information needed for MUSCLE 3 to be able to coordinate model execution. We'll go into more detail on this file in a moment.

The yMMSL YAML format is supported by the ymmsl-python library, whose documentation you are currently reading. This library lets you read and write yMMSL files, and manipulate their contents using an object-based Python API.

1.1 Installation

ymmsl-python is on PyPI, so you can install it using Pip:

```
pip install ymmsl
```

or you can add it to your dependencies as usual, e.g. in your `setup.py` or your `requirements.txt`, depending on how you've set up your project.

1.2 Reading yMMSL files

Here is an example of loading a yMMSL file:

```
from pathlib import Path
import ymmsl

config = ymmsl.load(Path('example.ymmsl'))
```

This makes `config` an object of type *`ymmsl.Configuration`*, which is the top-level class describing a yMMSL document. More on these objects in the next section. The *`ymmsl.load()`* function can also load from an open file or from a string containing YAML data.

If the file is valid YAML, but not recognized as a yMMSL file, the library will raise a `ymmsl.RecognitionError` with a message describing in detail what is wrong, so that you can easily fix the file.

Note that the *`ymmsl.load()`* function uses the safe loading functionality of the underlying YAML library, so you can safely load files from untrusted sources.

1.3 Writing yMMSL files

To write a yMMSL file with the contents of a *ymmsl.Configuration*, we use `ymmsl.save`:

```
from pathlib import Path
from ymmsl import Component, Configuration, Model, Settings

model = Model('test_model', [Component('macro')])
settings = Settings(OrderedDict([('test_parameter', 42)]))
config = Configuration(model, settings)

ymmsl.save(config, Path('out.ymmsl'))
```

Here, we create a model named `test_model`, containing a single component named `macro`, and no conduits. For the settings, we create a Settings object, which is a container for an ordered dictionary of settings. Note that normal Python dictionaries are unordered, which is why YAML documents saved from Python are often in a random order and hard to read. We avoid that problem in yMMSL by using an `OrderedDict` here. You have to pass it a list of tuples, because using dictionary syntax with curly brackets will lose the ordering.

Finally, we combine the model and the settings into a `yammsl.Configuration` object, which we then save to a file. If you want to have the YAML as a string, use *ymmsl.dump()* instead.

As the format may develop over time, files are required to carry a version, in this case v0.1, which is currently the only version of yMMSL.

When you read in a yMMSL file as described above, you get a collection of Python objects describing its contents. The next section explains how those work.

CHAPTER 2

Usage

As shown on the previous page, the `ymmsl-python` library converts yMMSL from YAML to Python objects and back. Here, we dive into this a bit deeper and see how those Python objects can be used.

Generally speaking, the object model used by the `ymmsl` library follows the structure of the YAML document, but there are a few places where some syntactic sugar has been added to make the files easier to read and write by hand. Let's have a look at the example again:

Listing 1: `docs/example.ymmsl`

```
ymmsl_version: v0.1

model:
  name: macro_micro_model
  components:
    macro: my.macro_model
    micro: my.micro_model
  conduits:
    macro.state_out: micro.init_in
    micro.final_out: macro.update_in

settings:
  # Scales
  domain._muscle_grain: 0.01
  domain._muscle_extent: 1.0
  macro._muscle_timestep: 10.0
  macro._muscle_total_time: 1000.0
  micro._muscle_timestep: 0.01
  micro._muscle_total_time: 1.0

  # Global settings
  k: 1.0
  interpolation_method: linear

  # Submodel-specific setting
```

(continues on next page)

(continued from previous page)

```
  micro.d: 2.3

implementations:
  my.macro_model:
    executable: /home/user/model
  my.micro_model:
    modules: gcc openmpi
    execution_model: openmpi
    executable: /home/user/model2

resources:
  macro:
    threads: 1
  micro:
    mpi_processes: 8

checkpoints:
  at_end: true
  simulation_time:
  - every: 50
```

If you read this into a variable named `config`, then `config` will contain an object of type *`ymmsl.Configuration`*. The yMMSL file above is a nested dictionary (or mapping, in YAML terms) with at the top level the keys `ymmsl_version`, `model` and `settings`. The `ymmsl_version` key is handled internally by the library, so it does not show up in the *`ymmsl.Configuration`* object. The others, *model* and *settings* are loaded into attributes of `config`.

Note that *settings* is optional: if it is not given in the YAML file, the corresponding attribute will be an empty *`ymmsl.Settings`* object. Likewise, when saving an empty *`ymmsl.Configuration`*, the *settings* section will be omitted.

As a result, `config.model` will give you an object representing the model part of the file, while `config.settings` contains an object with the settings in it. *`ymmsl.Configuration`* is just a simple record that holds the two parts together, so this is all it can do.

2.1 Models

The `model` section of the yMMSL document describes the simulation model. It has the model's name, a list of simulation components, and it describes the conduits between those components. (Simulation) components are submodels, scale bridges, mappers, proxies, and any other program that makes up the coupled simulation. Conduits are the wires between them that are used to exchange messages.

The `model` section is represented in Python by the *`ymmsl.Model`* class. It has attributes `name`, `components` and `conduits` corresponding to those sections in the file. Attribute *name* is an *`ymmsl.Identifier`* object.

Note that conduits are optional, you may have a model that consists of only one component and no conduits at all. In YAML, you can write this by omitting the conduits attribute. In Python, you can also omit the conduits argument when constructing a Model. In both cases, the `conduits` attribute will be an empty list.

Listing 2: Accessing the model

```
from pathlib import Path
import ymmsl

config = ymmsl.load(Path('example.ymmsl'))
```

(continues on next page)

(continued from previous page)

```
print(config.model.name)        # output: macro_micro_model
print(len(config.model.components))    # output: 2
```

An identifier contains the name of an object, like a simulation model, a component or a port (see below). It is a string containing letters, digits, and/or underscores which must start with a letter or underscore, and may not be empty. Identifiers starting with an underscore are reserved for use by the software (e.g. MUSCLE3), and may only be used as specified by the software you are using.

The *ymmsl.Identifier* Python class represents an identifier. It works almost the same as a normal Python `str`, but checks that the string it contains is actually a valid identifier.

2.1.1 Simulation Components

The `model` section contains a subsection `components`, in which the components making up the simulation are described. These are the submodels, and special components like scale bridges, data converters, load balancers, etc. yMMSL lets you describe components in two ways, a short one and a longer one:

Listing 3: Macro-meso-micro model components

```
components:
  macro: my.macro_model
  meso:
    ports:
      f_init: boundary_in
      o_i: state_out
      s: state_in
      o_f: boundary_out
    implementation: my.meso_model
    multiplicity: 5
  micro:
    implementation: my.micro_model
    multiplicity: [5, 10]
```

This fragment describes a macro-meso-micro model set-up with a single macro model instance, five instances of the meso model, and five sets of ten micro model instances each. If the simulation requires only a single instance of a component, the short form can be used, as above for the `macro` component. It simply maps the name of the component to an implementation (more on those in a moment).

The longer form maps the name of the component to a dictionary containing three attributes: the `ports`, the `implementation` and the `multiplicity`. Ports are the connectors on the component to which conduits attach to connect it to other components. These are organised by operator; we refer to the MUSCLE3 documentation for more on how they are used. Specifying ports here is optional, but doing so can improve efficiency.

The implementation is the name of the implementation as in the short form, while the multiplicity specifies how many instances of this component exist in the simulation. Multiplicity is a list of integers (as for `micro` in this example), but may be written as a single integer if it's a one-dimensional set (as for `meso`).

All this is a concise and easy to read and write a YAML file, but on the Python side, all this flexibility would make for complex code. To avoid that, the ymmsl-python library applies syntactic sugar when converting between YAML and Python. On the Python side, the `components` attribute of *ymmsl.Model* always contains a list of *ymmsl.Component* objects, regardless of how the YAML file was written. When this list is written to a YAML file, the most concise representation is automatically chosen to make the file easier to read by a human user.

Listing 4: Accessing the simulation components

```
from pathlib import Path
import ymmsl

config = ymmsl.load(Path('macro_meso_micro.ymmsl'))

cps = config.model.components
print(cps[0].name)               # output: macro
print(cps[0].implementation)     # output: my.macro_model
print(cps[0].multiplicity)       # output: []
print(cps[2].name)               # output: micro
print(cps[2].implementation)     # output: my.micro_model
print(cps[2].multplicity)        # output: [5, 10]
```

(Note that `macro_meso_micro.ymmsl` does not come with this documentation, go ahead and make it yourself using the above listing!)

The *`ymmsl.Component`* class has four attributes, unsurprisingly named `name`, `implementation`, `multiplicity` and `ports`. Attributes `name` and `implementation` are of type *`ymmsl.Reference`*. A reference is a string consisting of one or more identifiers (as described above), separated by periods.

Depending on the context, this may represent a name in a namespace (as it is here), or an attribute of an object (as we will see below with Conduits). The `multiplicity` attribute is always a list of ints, but may be omitted or given as a single int when creating a *`ymmsl.Component`* object, just like in the YAML file.

The `implementation` attribute of *`ymmsl.Component`* refers to an implementation definition. More on those below.

2.1.2 Conduits

The final subsection of the `model` section is labeled `conduits`. Conduits tie the components together by connecting ports on those components. Which ports a component has depends on the component, so you have to look at its documentation (or the source code, if there isn't any documentation) to see which ports are available and how they should be used.

As you can see, the conduits are written as a dictionary on the YAML side, which maps senders to receivers. A sender consists of the name of a component, followed by a period and the name of a port on that component; likewise for a receiver. In the YAML file, the sender is always on the left of the colon, the receiver on the right.

Just like the simulation components, the conduits get converted to a list in Python, in this case containing *`ymmsl.Conduit`* objects. The *`ymmsl.Conduit`* class has `sender` and `receiver` attributes, of type *`ymmsl.Reference`* (see above), and a number of helper functions to interpret these fields, e.g. to extract the component and port name parts. Note that the format allows specifying a slot here, but this is currently not supported and illegal in MUSCLE3.

Multicast conduits

In yMMSL you can specify that an output port is connected to multiple input ports. When a message is sent on the output port, it is copied and delivered to all connected input ports. This is called multicast and is expressed as follows:

Listing 5: Specifying multicast in yMMSL

```
conduits:
  sender.port:
```

(continues on next page)

(continued from previous page)

```
  - receiver1.port
  - receiver2.port
```

This multicast conduit is converted to a a list of conduits sharing the same sender:

Listing 6: Multicast conduits in python code

```
from pathlib import Path
import ymmsl

config = ymmsl.load(Path('multicast.ymmsl'))

conduits = config.model.conduits
print(len(conduits))     # output: 2
print(conduits[0])       # output: Conduit(sender.port -> receiver1.port)
print(conduits[1])       # output: Conduit(sender.port -> receiver2.port)
```

2.2 Settings

The settings section contains settings for the simulation to run with. In YAML, this is a dictionary that maps a setting name (a *ymmsl.Reference*) to its value. Parameter values may be strings, integers, floating point numbers, lists of floating point numbers (vectors), or lists of lists of floating point numbers (arrays).

Listing 7: Settings example

```
settings:
  domain.grain: 0.01
  domain.extent.x: 1.0
  domain.extent.y: 1.0
  macro.timestep: 10.0
  macro.total_time: 1000.0
  micro.timestep: 0.01
  micro.total_time: 1.0

  interpolate: true
  interpolation_method: linear
  kernel:
    - [0.8, 0.2]
    - [0.2, 0.8]
```

In this example, there is a macro-micro model in which the two models share a one-dimensional domain, which is named domain, has a length and width of 1.0, and a grid spacing of 0.01. The macro model has a time step of 10 and a total run time of 1000 (so it will run for 100 steps), while the micro model has a time step of 0.01 and a total run time of 1.0. Furthermore, there are some other model settings, a boolean switch that enables interpolation, a string to select the interpolation method, and a 2D array specifying a kernel of some kind.

On the Python side, this will be turned into a *ymmsl.Settings* object, which acts much like a Python dictionary. So for instance, if you have a *ymmsl.Configuration* object named `config` which was loaded from a file containing the above `settings` section, then you could write:

```
grid_dx = config.settings['domain.grain']
kernel = config.settings['kernel']
```

to obtain a floating point value of 0.1 in `grid_dx` and a list of lists `[[0.8, 0.2], [0.2, 0.8]]` in `kernel`.

2.3 Implementations

Components are abstract objects. For an actual simulation to run, we need computer programs that implement the components of the simulation. As we've seen above, components refer to implementations, and those implementations are defined in the `implementations` section of the yMMSL file:

Listing 8: Defining implementations

```
implementations:
  simplest:
    executable: /home/user/models/my_model

  python_script:
    virtual_env: /home/user/envs/my_env
    executable: python3
    args: /home/user/models/my_model.py

  with_env_and_args:
    env:
      LD_LIBRARY_PATH: /home/user/muscle3/lib
      ENABLE_AWESOME_SCIENCE: 1
    executable: /home/user/models/my_model
    args:
      - --some-lengthy-option
      - --some-other-lengthy-option=some-lengthy-value
```

As you can see, there are quite a few different ways of describing an implementation, but all implementations have a name, which is the key in the dictionary, by which a component can refer to it.

The `simplest` implementation only has an executable. This could be a (probably statically linked) executable, or a script that sets up an environment and starts the model.

If your model or other component is a Python script, then you may want to load a virtual environment before starting it, to make the dependencies available. This is done using the `virtual_env` attribute. If the script does not have a `#!/usr/bin/env python` line at the top (in which case you could set it as the executable) then you need to start the Python interpreter directly, and pass the location of the script as an argument.

Environment variables can be set through the `env` attribute, which contains a dictionary mapping variable names to values, as shown for the `with_env_and_args` example. This also shows that you can pass the arguments as a list, if that makes things easier to read.

Listing 9: MPI and HPC implementations

```
implementations:
  mpi_implementation:
    executable: /home/user/models/my_model
    execution_model: openmpi

  on_hpc_cluster:
    modules: cpp openmpi
    executable: /home/user/models/my_model
    execution_model: intelmpi

  with_script:
    script: |
      #!/bin/bash
```

(continues on next page)

(continued from previous page)

```
    . /home/user/muscle3/bin/muscle3.env
    export ENABLE_AWESOME_SCIENCE=1

    /home/user/models/my_model -v -x
```

MPI programs are a bit special, as they need to be started via `mpirun`. However, `mpirun` assumes that the program to start is going to use all of the available resources. For a coupled simulation with multiple components, that is usually not what you want. It is possible to tell `mpirun` to only use some of the resources, but of course we don't know which ones will be available while writing this file. Instead, you simply specify the path to the executable, and set the `execution_model` attribute to either `openmpi` or `intelmpi` as required. When executing with MUSCLE3, the MUSCLE Manager will then start the component on its designated subset of the resources as required.

The `on_hpc_cluster` implementation demonstrates loading environment modules, as commonly needed on HPC machines. They're all in one line here, but if the modules have long names, then like with the arguments you can make a list to keep things readable.

Finally, if you need to do something complicated, you can write an inline script to start the implementation. This currently only works for non-MPI programs however.

2.3.1 Keeps state for next use

Implementations may indicate if they carry state between reuses. This is currently only used for *checkpoints*, but might see further use in the future (e.g. for load balancers). There are three possible values an implementation may indicate.

Necessary This implementation remembers state between consecutive iterations of the reuse loop. That state is required for the proper execution of the implementation.

This is the default value when not specified.

Example: A micro model simulating an enclosed volume, where every reuse the boundary conditions are updated by the connected macro model. This micro model must keep track of the state inside the simulated volume between iterations of the reuse loop.

No This implementation has no state between consecutive iterations of the reuse loop.

Example: A data converter that receives on an `F_INIT` port, transforms the data and outputs it on an `O_F` port. The transformation is only dependent on the information of the `F_INIT` message.

Helpful This implementation remembers state between consecutive iterations of the reuse loop. However, this state is not required for proper execution.

Example: A simulation of a fluid in a pipe with obstacles. The simulation converges much faster when starting from the solution of the previous iteration. However, the same solution can still be found when starting from scratch.

2.4 Resources

Finally, yMMSL allows specifying the amount of resources needed to run an instance of an implementation. This information is used by MUSCLE3 when it starts each component, to ensure it has the resources needed to do its calculations. Currently, only the number of threads or processes can be specified; memory and GPUs are future work.

Resources are specified per component, and apply to each instance of that component. For single- or multithreaded components, or components that use multiple local processes (for example with Python's `multiprocessing`), you specify the number of threads:

Listing 10: Resources for threaded processes

```
resources:
  macro:
    threads: 1

  micro:
    threads: 8
```

On the Python side, this is represented by *ymmsl.ThreadedResReq* (short for ThreadedResourceRequirements), which holds the name of the component it specifies the resources for in attribute `name`, and the number of threads or processes (basically, cores) as `threads`.

For MPI-based implementations, there are two different ways of specifying the required resources: core-based and node-based. For core-based resource requirements (*ymmsl.MPICoresResReq* on the Python side), you specify the number of MPI processes, and optionally the number of threads per MPI process:

Listing 11: Core-based resources for MPI components

```
resources:
  macro:
    mpi_processes: 32
  micro:
    mpi_processes: 16
    threads_per_mpi_process: 8
```

On HPC, this allocates each MPI process individually.

Node-based MPI allocations are not yet supported by MUSCLE3, but you can already specify them as follows:

Listing 12: Node-based resources for MPI components

```
resources:
  macro:
    nodes: 8
    mpi_processes_per_node: 4
    threads_per_mpi_process: 8
  micro:
    nodes: 1
    mpi_processes_per_node: 16
```

Here, whole nodes are assigned to the implementation, with a specific number of MPI processes started on each node, and optionally (the default is one) a certain number of cores per process made available.

More information on how this is interpreted and how MUSCLE3 allocates resources based on this can be found in the High-Performance Computing section in the MUSCLE3 documentation.

2.5 Checkpoints

In yMMSL you can specify if you expect the workflow to create checkpoints. Note that all implementations in your workflow must support checkpointing, MUSCLE3 will generate an error for you otherwise. See the documentation for MUSCLE3 on checkpointing for details on enabling checkpointing for an implementation.

2.5.1 Checkpoint triggers

In yMMSL you have three possible checkpoint triggers:

at_end Create a checkpoint just before the instance shuts down. This can be a useful checkpoint if you intend to resume the workflow at some later point, e.g. when you wish to simulate a longer time span. This trigger is either on or off, specified with a boolean `true` or `false` (default) in the configuration.

simulation_time Create checkpoints based on the passed simulation time. This can only work properly if there is a shared concept of simulated time in the workflow.

wallclock_time Create checkpoints based on the passed wall clock time (also called elapsed real time). This method is not perfect and may result in missed checkpoints in certain coupling scenarios. See the MUSCLE3 documentation for a discussion of the limitations.

When you use any of the time-based triggers, you must also specify at what moments a checkpoint is expected. MUSCLE3 will then snapshot as soon as possible **after** reaching the specified times. You may indicate specific moments with `at`-rules, but can also create repetitive checkpoints.

Listing 13: Example checkpoint definition

```
checkpoints:
  at_end: true
  simulation_time:
  - at: [1.2, 1.4]
  - every: 1
  wallclock_time:
  - every: 60
    stop: 600
  - every: 600
    start: 600
    stop: 3600
  - every: 1800
    start: 3600
```

Above example demonstrates all possible checkpoint options. The workflow will create checkpoints:

- At the end: `at_end: true`.
- Every second of passed simulated time (`t=0,1,2,...`), and additionally at `t=1.2` and `t=1.4`.
- Every minute of real elapsed time, for the first 10 minutes; then every 10 minutes for the remainder of the first hour; then every 30 minutes until the end.

See the API documentation for *`CheckpointRangeRule`* for more details on the behaviour of the repetitive checkpoints.

2.6 Examples

All the classes mentioned here are normal Python classes. They have constructors which you can use to create instances, and their attributes can be changed as needed.

Here are a few examples:

Listing 14: Creating a Configuration and saving it

```
from pathlib import Path
import ymmsl
```

(continues on next page)

(continued from previous page)

```
components = [
    ymmsl.Component('macro', 'my.macro_model'),
    ymmsl.Component('micro', 'my.micro_model')]

conduits = [
    ymmsl.Conduit('macro.out', 'micro.in'),
    ymmsl.Conduit('micro.out', 'macro.in')]

model = ymmsl.Model('my_model', components, conduits)

implementations = [
    ymmsl.Implementation(
        'my.macro_model', executable='/home/user/model'),
    ymmsl.Implementation(
        'my.micro_model', modules='gcc openmpi',
        execution_model=ymmsl.ExecutionModel.OPENMPI)]

resources = [
    ymmsl.ThreadedResReq(1),
    ymmsl.MPICoresResReq(8)]

config = ymmsl.Configuration(model, implementations, resources)

ymmsl.save(config, Path('out.ymmsl'))

# Will produce:
# ymmsl_version: v0.1
# model:
#   name: my_model
#   components:
#     macro: my.macro_model
#     micro: my.micro_model
#   conduits:
#     macro.out: micro.in
#     micro.out: macro.in
# implementations:
#   my.macro_model:
#     executable: /home/user/model
#   my.micro_model:
#     modules: gcc openmpi
#     execution_model: openmpi
# resources:
#   macro:
#     threads: 1
#   micro:
#     mpi_processes: 8
```

Listing 15: Adding or changing a setting

```
from pathlib import Path
import ymmsl

config = ymmsl.load(Path('example.ymmsl'))

config.settings['d'] = 0.12
```

(continues on next page)

(continued from previous page)

```
ymmsl.save(config, Path('out.ymmsl'))
```

For more details about these classes and what you can do with them, we refer to the API documentation.

CHAPTER 3

API Reference

3.1 API documentation

Python bindings for yMMSL.

This package contains all the classes needed to represent a yMMSL file, as well as to read and write yMMSL files.

class CheckpointRule
Defines a checkpoint rule.

There are two flavors of rules: *CheckpointRangeRule* and *CheckpointAtRule*. Do not use this class directly.

class CheckpointRangeRule(*start: Union[float, int, None] = None*, *stop: Union[float, int, None] = None*, *every: Union[float, int] = 0*)
Defines a range of checkpoint moments.

If `start` is supplied, this rule specifies a checkpoint at: `start`, `start + every`, `start + 2*every`, etc., for as long as `start + n*every <= stop`, with `n` a whole number. If stop is not given, the range continues indefinitely.

Start may be omitted, in which case a checkpoint is defined for `0`, `every`, `2*every`, etc. Note that in this case the range also extends to negative numbers (`-every`, `-2*every`, etc.), as simulated time may be negative (e.g. in rocket launch, `t=0` is generally taken as lift-off time, but events already take place before that moment).

start
Start of the range.

stop
Stopping criterium of the range.

every
Step size of the range, must be positive.

class CheckpointAtRule(*at: Optional[List[Union[float, int]]]*)
Defines an "at" checkpoint rule.

An "at" checkpoint rule creates a snapshot at the specified moments.

at
List of checkpoints.

class Checkpoints(*at_end: bool = False*, *wallclock_time: Optional[List[CheckpointRule]] = None*, *simulation_time: Optional[List[CheckpointRule]] = None*)
Defines checkpoints in a configuration.

There are three checkpoint triggers: *at_end*, *wallclock_time* and *simulation_time*. The *at_end* trigger specifies that a checkpoint should be created just before the workflow finishes. The *wallclock_time* trigger is based on the elapsed real time since starting the muscle_manager in seconds. The *simulation_time* trigger is based on the time in the simulation as reported by the instances.

Note that the *simulation_time* trigger assumes a shared concept of time among all components of the model.

at_end
Whether a checkpoint should be created just before ending the workflow.

wallclock_time
Checkpoint rules for the wallclock_time trigger.

simulation_time
Checkpoint rules for the simulation_time trigger.

update(*overlay: Checkpoints*) → None
Update this checkpoints with the given overlay.

Sets *at_end* to True if it is set in the overlay, otherwise *at_end* remains as is. Updates the checkpoint rules for wallclock time and simulation time. See `CheckpointRules.update()`.

class Component(*name: str*, *implementation: Optional[str] = None*, *multiplicity: Union[None, int, List[int]] = None*, *ports: Optional[Ports] = None*)
An object declaring a simulation component.

Simulation components are things like submodels, scale bridges, proxies, and any other program that makes up a model. This class represents a declaration of a set of instances of a simulation component, and it's used to describe which instances are needed to perform a certain simulation.

name
The name of this component.

Type *ymmsl.Reference*

implementation
A reference to the implementation to use.

Type *ymmsl.Reference*

multiplicity
The shape of the array of instances that execute simultaneously.

Type List[int]

ports
The ports of this component, organised by operator. None if not specified.

Type Optional[*Ports*]

instances() → List[Reference]
Creates a list of instances needed.

Returns A list with one Reference for each instance of this component.

class Conduit(*sender: str*, *receiver: str*)
A conduit transports data between simulation components.

A conduit has two endpoints, which are references to a port on a simulation component. These references must be of one of the following forms:

- component.port
- namespace.component.port (or several namespace prefixes)

sender
The sending port that this conduit is connected to.

receiver
The receiving port that this conduit is connected to.

receiving_component() → Reference
Returns a reference to the receiving component.

receiving_port() → Identifier
Returns the identity of the receiving port.

receiving_slot() → List[int]
Returns the slot on the receiving port.

If no slot was given, an empty list is returned.

Returns A list of slot indexes.

sending_component() → Reference
Returns a reference to the sending component.

sending_port() → Identifier
Returns the identity of the sending port.

sending_slot() → List[int]
Returns the slot on the sending port.

If no slot was given, an empty list is returned.

Returns A list of slot indexes.

class Configuration(*model: Model, settings: Optional[Settings] = None, implementations: Union[List[Implementation], Dict[Reference, Implementation]] = [], resources: Union[Sequence[ResourceRequirements], MutableMapping[Reference, ResourceRequirements]] = [], description: Optional[str] = None, checkpoints: Optional[Checkpoints] = None, resume: Optional[Dict[Reference, pathlib.Path]] = None*)
Configuration that includes all information for a simulation.

PartialConfiguration has some optional attributes, because we want to allow configuration files which only contain some of the information needed to run a simulation. At some point however, you need all the bits, and this class requires them.

When loading a yMMSL file, you will automatically get an object of this class if all the required components are there; if the file is incomplete, you'll get a PartialConfiguration instead.

model
A model to run.

settings
Settings to run the model with.

implementations
Implementations to use to run the model. Dictionary mapping implementation names (as References) to Implementation objects.

resources
Resources to allocate for the model components. Dictionary mapping component names to Resources objects.

description
A human-readable description of the configuration.

checkpoints
Defines when each model component should create a snapshot

resume
Defines what snapshot each model component should resume from

check_consistent() → None
Checks that the configuration is internally consistent.

This checks whether all conduits are connected to existing components, that there's an implementation for every component, and that resources have been requested for each component.

If any of these requirements is false, this function will raise a RuntimeError with an explanation of the problem.

dump(*config: PartialConfiguration*) → str
Converts a yMMSL configuration to a string containing YAML.

Parameters **config** – The configuration to be saved to yMMSL.

Returns A yMMSL YAML description of the given document.

class ExecutionModel
Describes how to start a model component.

DIRECT = 1
Start directly on the allocated core(s), without MPI.

INTELMPI = 3
Start using Intel MPI's mpirun.

OPENMPI = 2
Start using OpenMPI's mpirun.

SRUNMPI = 4
Start MPI implementation using srun.

class Identifier(*seq: Any*)
A custom string type that represents an identifier.

An identifier may consist of upper- and lowercase characters, digits, and underscores.

class Implementation(*name: Reference, modules: Union[str, List[str], None] = None, virtual_env: Optional[pathlib.Path] = None, env: Optional[Dict[str, str]] = None, execution_model: ExecutionModel = <ExecutionModel.DIRECT: 1>, executable: Optional[pathlib.Path] = None, args: Union[str, List[str], None] = None, script: Union[str, List[str], None] = None, can_share_resources: bool = True, keeps_state_for_next_use: KeepsStateForNextUse = <KeepsStateForNextUse.NECESSARY: 1>*)
Describes an installed implementation.

An Implementation normally has an `executable` and any other needed attributes, with `script` set to None. You should specify a script only as a last resort, probably after getting some help from the authors of this library. If a script is specified, all other attributes except for the name must be `None`.

For `execution_model`, the following values are supported:

direct The program will be executed directly. Use this for non-MPI programs.

openmpi For MPI programs that should be started using OpenMPI's mpirun.

intelmpi For MPI programs that should be started using Intel MPI's mpirun.

The `can_share_resources` attribute describes whether this implementation can share resources (cores) with other components in a macro-micro coupling. Set this to `False` if the implementation does significant computing inside of its time update loop after having sent messages on its O_I port(s) but before receiving messages on its S port(s). In the unlikely case that it's doing significant computing before receiving for F_INIT or after sending its O_F messages, likewise set this to `False`.

Setting this to `False` unnecessarily will waste core hours, setting it to `True` incorrectly will slow down your simulation.

name
Name of the implementation

modules
HPC software modules to load

virtual_env
Path to a virtual env to activate

env
Environment variables to set

execution_model
How to start the executable

executable
Full path to executable to run

args
Arguments to pass to the executable

script
A script that starts the implementation

can_share_resources
Whether this implementation can share resources (cores) with other components or not

keeps_state_for_next_use
Does this implementation keep state for the next iteration of the reuse loop. See `ImplementationState`.

class KeepsStateForNextUse
Describes whether an implementation keeps internal state between iterations of the reuse loop.

See also *Keeps state for next use*.

HELPFUL = 3
The implementation has an internal state, though this could be regenerated. Doing so may be expensive.

NECESSARY = 1
The implementation has an internal state that is necessary for continuing the implementation.

NO = 2
The implementation has no internal state.

load(*source: Union[str, pathlib.Path, IO[Any]]*) → PartialConfiguration
Loads a yMMSL document from a string or a file.

Parameters **source** – A string containing yMMSL data, a pathlib Path to a file containing yMMSL data, or an open file-like object containing from which yMMSL data can be read.

Returns A PartialConfiguration object corresponding to the input data.

class Model(*name: str, components: List[Component], conduits: Optional[List[Union[Conduit, MulticastConduit]]] = None*)
Describes a simulation model.

A model consists of a number of components connected by conduits.

Note that there may be no conduits, if there is only a single component. In that case, the conduits argument may be omitted when constructing the object, and also from the YAML file; the *conduits* attribute will then be set to an empty list.

name
The name by which this simulation model is known to the system.

components
A list of components making up the model.

conduits
A list of conduits connecting the components.

check_consistent() → None
Checks that the model is internally consistent.

This checks whether all conduits are connected to existing components, and will raise a RuntimeError with an explanation if one is not.

update(*overlay: Model*) → None
Overlay another model definition on top of this one.

This updates the object with the name, components and conduits given in the argument. The name is overwritten, and components are overwritten if they have the same name as an existing argument or else added.

Conduits are added. If a receiving port was already connected, the old conduit is removed. If a sending port was already connected, the new conduit is added and the sending port acts as a multicast port.

Parameters **overlay** – A Model definition to overlay on top of this one.

class ModelReference(*name: str*)
Describes a reference (by name) to a model.

name
The name of the simulation model this refers to.

class MPICoresResReq(*name: Reference*, *mpi_processes: int*, *threads_per_mpi_process: int = 1*)
Describes resources for simple MPI implementations.

This allocates individual cores or sets of cores on the same node for a given number of MPI processes per instance.

name
Name of the component to configure.

mpi_processes
Number of MPI processes to start.

threads_per_mpi_process
Number of threads/cores per process.

class MPINodesResReq(*name: Reference, nodes: int, mpi_processes_per_node: int, threads_per_mpi_process: int = 1*)
Describes resources for node based MPI implementations.

This allocates resources for an MPI process in terms of nodes and cores, processes and threads on them.

name
Name of the component to configure.

nodes
Number of nodes to reserve.

mpi_processes_per_node
Number of MPI processes to start on each node.

threads_per_mpi_process
Number of threads/cores per process.

class Operator
An operator of a component.

This is a combination of the Submodel Execution Loop operators, and operators for other components such as mappers.

F_INIT = 1
Initialisation phase, before start of the SEL

NONE = 0
No operator

O_F = 5
Observation of final state, after the SEL

O_I = 2
State observation within the model's main loop

S = 3
State update in the model's main loop

allows_receiving() → bool
Whether ports on this operator can receive.

allows_sending() → bool
Whether ports on this operator can send.

class PartialConfiguration(*model: Optional[ModelReference] = None, settings: Optional[Settings] = None, implementations: Union[List[Implementation], Dict[Reference, Implementation], None] = None, resources: Union[Sequence[ResourceRequirements], MutableMapping[Reference, ResourceRequirements], None] = None, description: Optional[str] = None, checkpoints: Optional[Checkpoints] = None, resume: Optional[Dict[Reference, pathlib.Path]] = None*)
Top-level class for all information in a yMMSL file.

model
A model to run.

settings
Settings to run the model with.

implementations
Implementations to use to run the model. Dictionary mapping implementation names (as References) to Implementation objects.

resources
Resources to allocate for the model components. Dictionary mapping component names to ResourceRequirements objects.

description
A human-readable description of the configuration.

checkpoints
Defines when each model component should create a snapshot

resume
Defines what snapshot each model component should resume from

as_configuration() → Configuration
Converts to a full Configuration object.

This checks that this PartialConfiguration has all the pieces needed to run a simulation, and if so converts it to a Configuration object.

Note that this doesn't check references, just that there is a model, implementations and resources. For the more extensive check, see *Configuration.check_consistent()*.

Returns A corresponding Configuration.

Raises ValueError – If this configuration isn't complete.

update(*overlay: PartialConfiguration*) → None
Update this configuration with the given overlay.

This will update the model according to *Model.update()*, copy settings from overlay on top of the current settings, overwrite implementations with the same name and add implementations with a new name, and likewise for resources and resume. The description of the overlay is appended to the current description. Checkpoints are updated according to *Checkpoints.update()*.

Parameters **overlay** – A configuration to overlay onto this one.

class Port(*name: Identifier*, *operator: Operator*)
A port on a component.

Ports are used by component to send or receive messages on. They are connected by conduits to enable communication between components.

name
The name of the port.

operator
The MMSL operator in which this port is used.

class Ports(*f_init: Union[None, str, List[str]] = None*, *o_i: Union[None, str, List[str]] = None*, *s: Union[None, str, List[str]] = None*, *o_f: Union[None, str, List[str]] = None*)
Ports declaration for a component.

Ports objects compare for equality by value. The names may be specified as a list of strings, or separated by spaces in a single string. If a particular operator has no associated ports, it may be omitted. For example:

```
ports:
  f_init:   # list of names
  - a
  - b
```

(continues on next page)

(continued from previous page)

```
o_i: c d  # on one line, space-separated
s: e      # single port
          # o_f omitted as it has no ports
```

f_init
The ports associated with the F_INIT operator.

o_i
The ports associated with the O_I operator

s
The ports associated with the S operator.

o_f
The ports associated with the O_F operator

all_ports() → Iterable[Port]
Returns an iterable containing all ports.

operator(*port_name: Identifier*) → Operator
Looks up the operator for a given port.

Parameters **port_name** – Name of the port to look up.

Returns The operator for that port.

Raises KeyError – If no port with this name was found.

port_names() → Iterable[Identifier]
Returns an iterable containing the names of all ports.

class Reference(*parts: Union[str, List[Union[Identifier, int]]]*)
A reference to an object in the MMSL execution model.

References in string form are written as either:

- an Identifier,
- a Reference followed by a period and an Identifier, or
- a Reference followed by an integer enclosed in square brackets.

In object form, they consist of a list of Identifiers and ints. The first list item is always an Identifier. For the rest of the list, an Identifier represents a period operator with that argument, while an int represents the indexing operator with that argument.

Reference objects act like a list of Identifiers and ints, you can get their length using len(), iterate through the parts using a loop, and get sublists or individual items using []. Note that the sublist has to be a valid Reference, so it cannot start with an int.

References can be compared for equality to each other or to a plain string, and they can be used as dictionary keys. Reference objects are immutable (or they're supposed to be anyway), so do not try to change any of the elements. Instead, make a new Reference. Especially References that are used as dictionary keys must not be modified, this will get your dictionary in a very confused state.

without_trailing_ints() → Reference
Returns a copy of this Reference with trailing ints removed.

Examples

a.b.c[1][2] -> a.b.c a[1].b.c -> a[1].b.c a.b.c -> a.b.c a[1].b.c[2] -> a[1].b.c

class ResourceRequirements(*name: Reference*)
Describes resources to allocate for components.

name
Name of the component to configure.

save(*config: PartialConfiguration, target: Union[str, pathlib.Path, IO[Any]]*) → None
Saves a yMMSL configuration to a file.

Parameters

- **config** – The configuration to save to yMMSL.
- **target** – The file to save to, either as a string containing a path, as a pathlib Path object, or as an open file-like object.

class Settings(*settings: Optional[Dict[str, Union[str, int, float, bool, List[float], List[List[float]], yatiml.util.bool_union_fix]]] = None*)
Settings for doing an experiment.

An experiment is done by running a model with particular settings, for the submodel scales, model parameters and any other configuration.

as_ordered_dict() → collections.OrderedDict
Represent as a dictionary of plain built-in types.

Returns: A dictionary that uses only built-in types, containing the configuration.

copy() → Settings
Makes a shallow copy of these settings and returns it.

ordered_items() → List[Tuple[Reference, Union[str, int, float, bool, List[float], List[List[float]], yatiml.util.bool_union_fix]]]
Return settings as a list of tuples.

class ThreadedResReq(*name: Reference*, *threads: int*)
Describes resources for threaded implementations.

This includes singlethreaded and multithreaded implementations that do not support MPI. As many cores as specified will be allocated on a single node, for each instance.

name
Name of the component to configure.

threads
Number of threads/cores per instance.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

y

Index