Vivarium: overview and demos

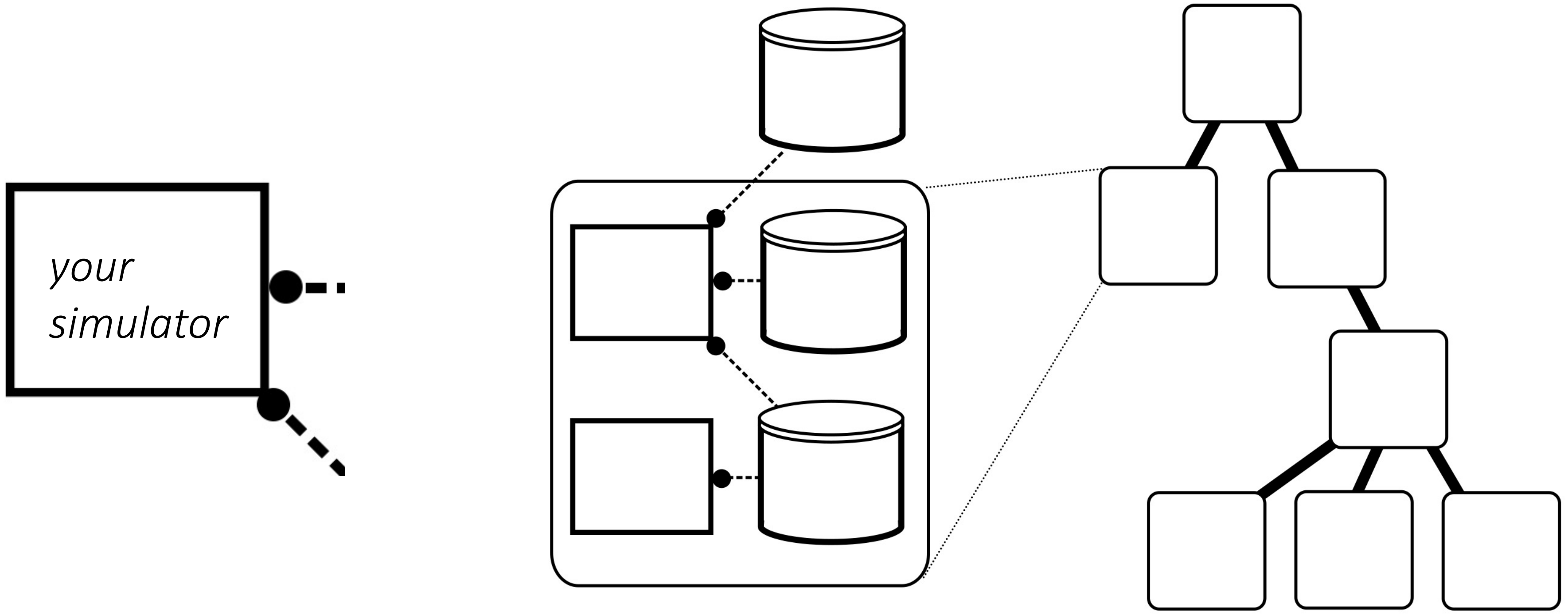
Eran Agmon, PhD

Department of Bioengineering | Stanford University

COMBINE 2021 | 10/13/2021

Overview

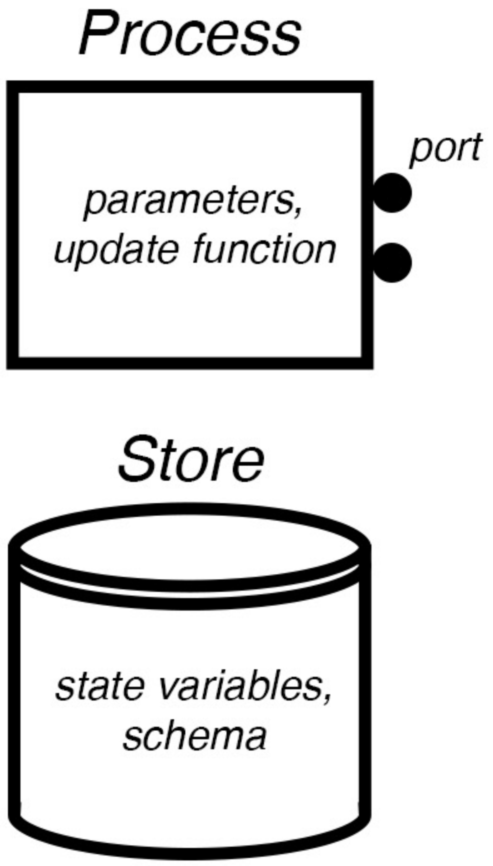
We need an "interface protocol" for connecting separate models, simulators, and data into a large, complex, and open-ended network that anyone can contribute to.

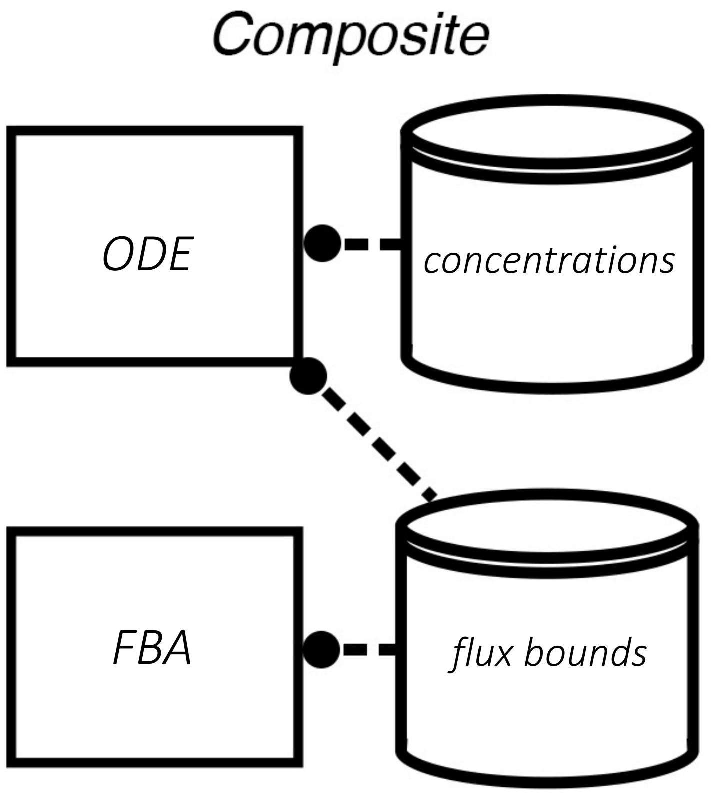


Agmon, E., et al. (2021). Vivarium: an interface and engine for integrative multi-scale modeling in computational biology. *bioRxiv.*

Basic Elements

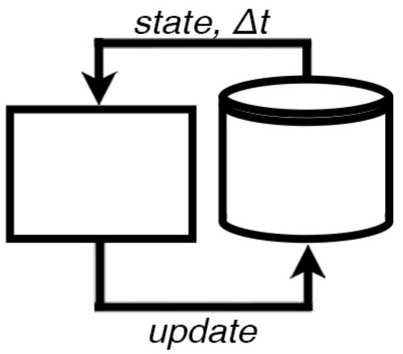
- **Processes:** consist of parameters, ports, and an update function (i.e. the simulator).
- **Stores:** hold the state variables, map the variable names to their values, and apply the process updates.
- **Composites:** bundles of processes and stores, wired together by their ports, and run together in time.



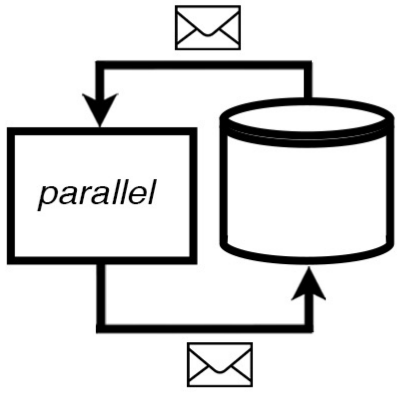


Co-Simulation Engine

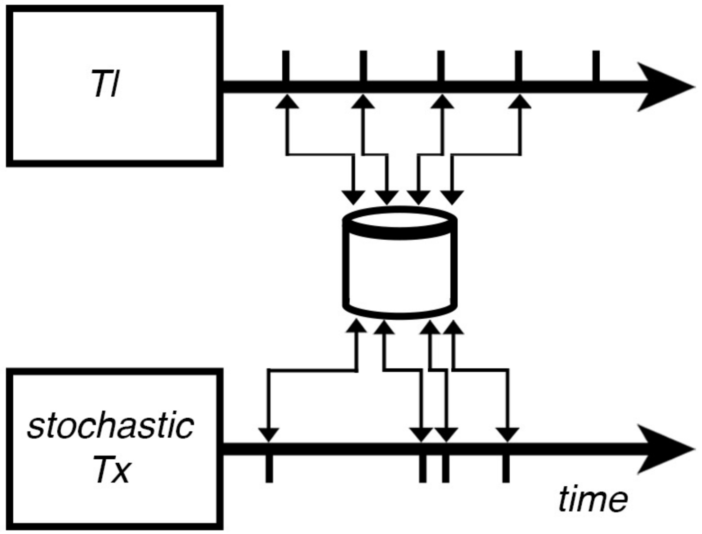
basic simulation cycle



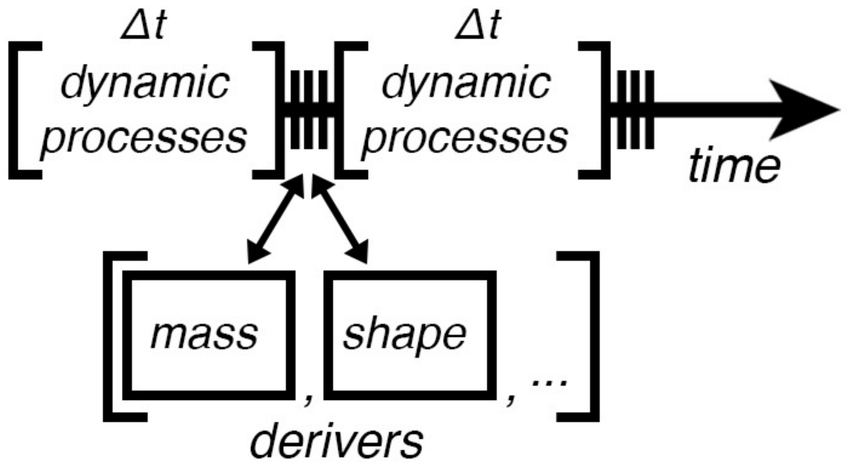
distributed processing



multi-timestepping

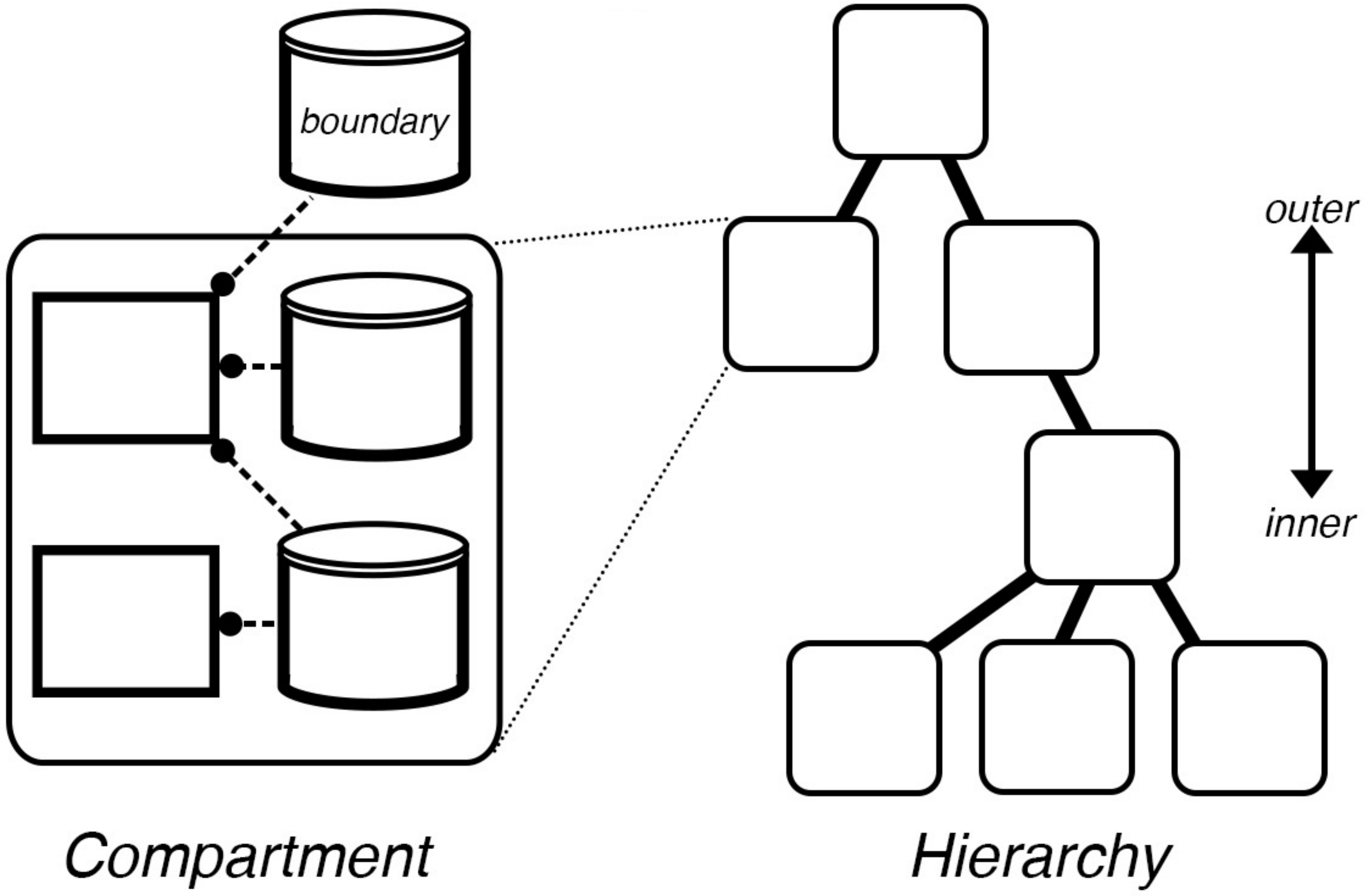


“derivers” run between
the time-dependent processes



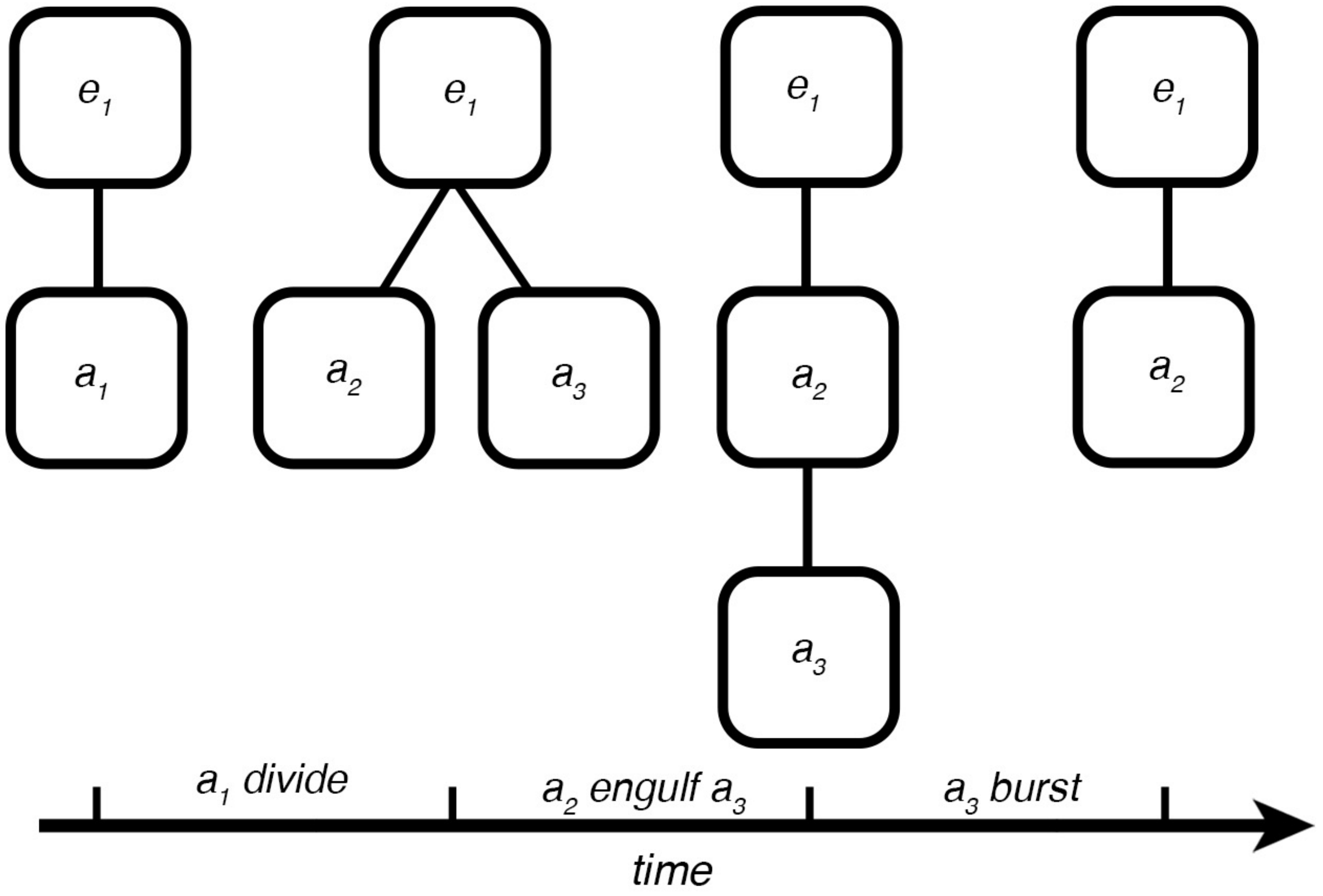
Hierarchical Embedding

- A bigraph is a graph with embeddable nodes that can be placed *within* other nodes.



Hierarchical Updates

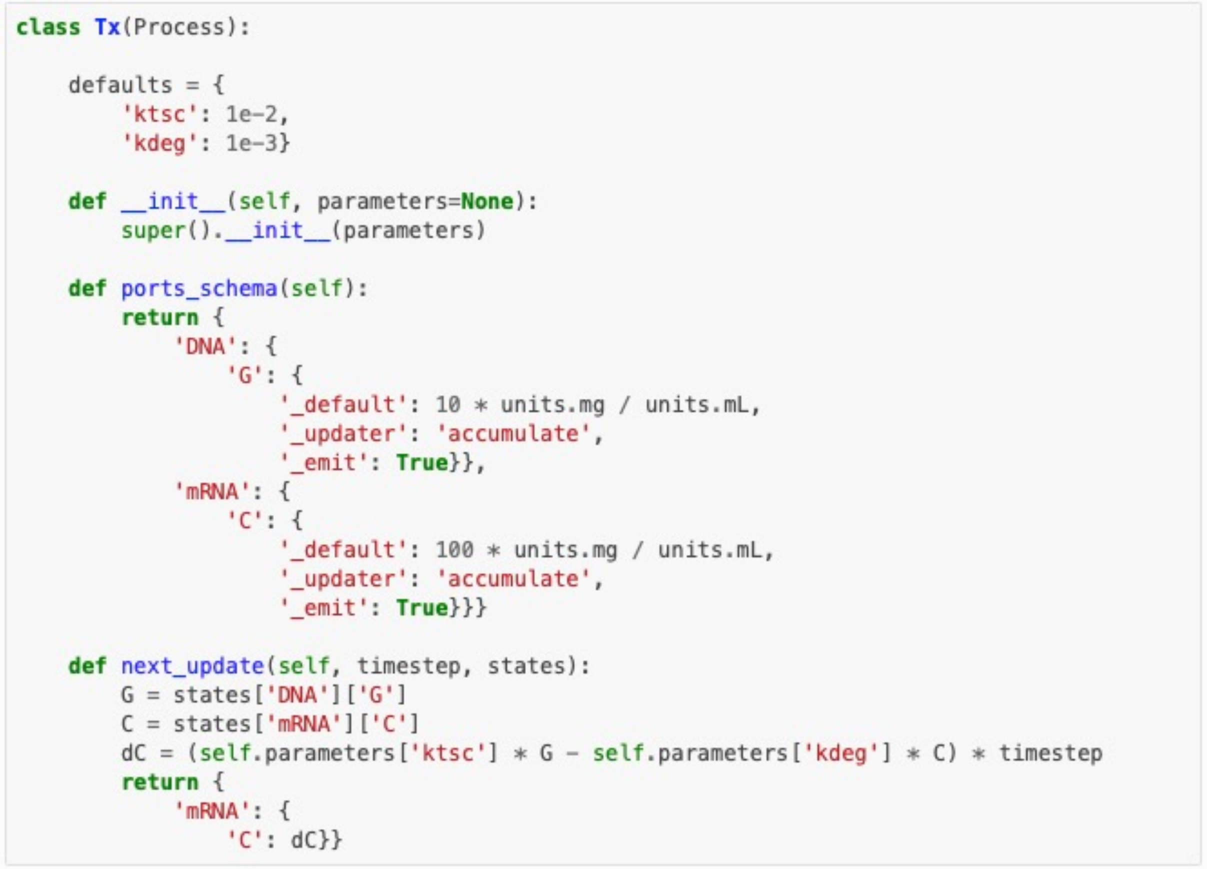
Processes, stores, and entire sub-graphs can be added/removed/moved during simulation runtime.



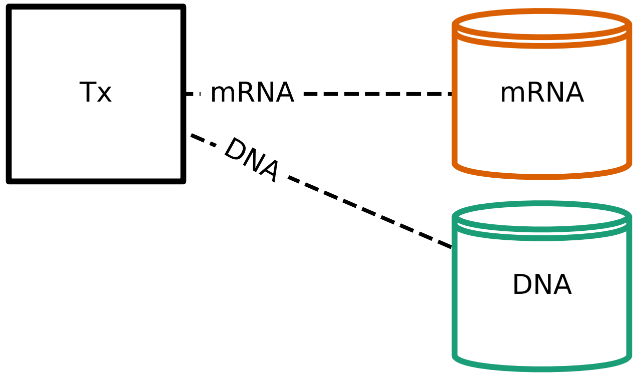
Minimal Example: Transcription ODE

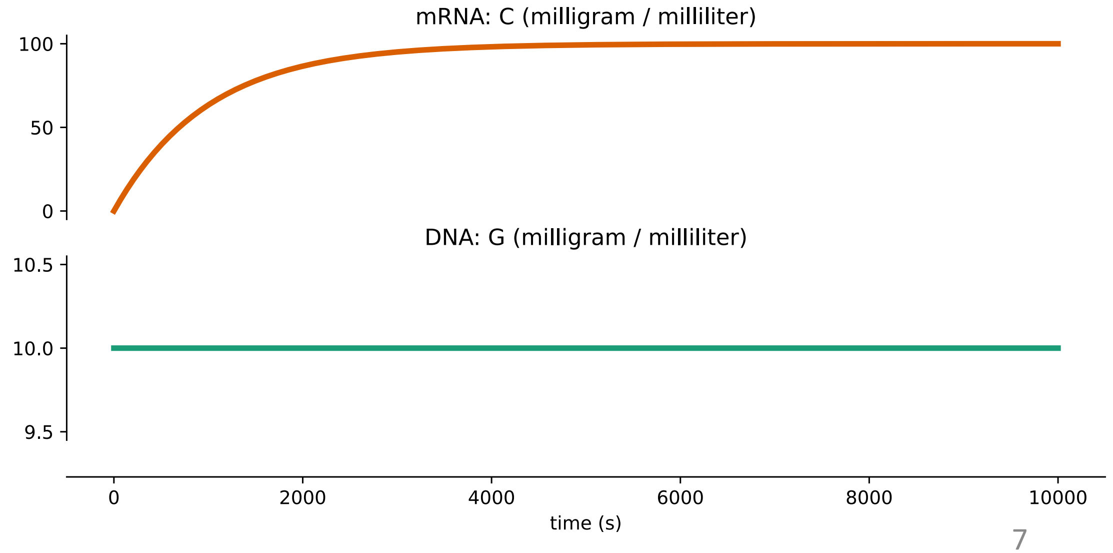
Process Interface Protocol:

1. **constructor***:* accepts parameters and configures the model.
2. **ports_schema***:* declares the ports and their schema.
3. **next_update***:* runs the model and returns an update.



```python
class Tx(Process):

    defaults = {
        'ktsc': 1e-2,
        'kdeg': 1e-3}

    def __init__(self, parameters=None):
        super().__init__(parameters)

    def ports_schema(self):
        return {
            'DNA': {
                'G': {
                    '_default': 10 * units.mg / units.mL,
                    '_updater': 'accumulate',
                    '_emit': True}},
            'mRNA': {
                'C': {
                    '_default': 100 * units.mg / units.mL,
                    '_updater': 'accumulate',
                    '_emit': True}}}

    def next_update(self, timestep, states):
        G = states['DNA']['G']
        C = states['mRNA']['C']
        dC = (self.parameters['ktsc'] * G - self.parameters['kdeg'] * C) * timestep
        return {
            'mRNA': {
                'C': dC}}
```

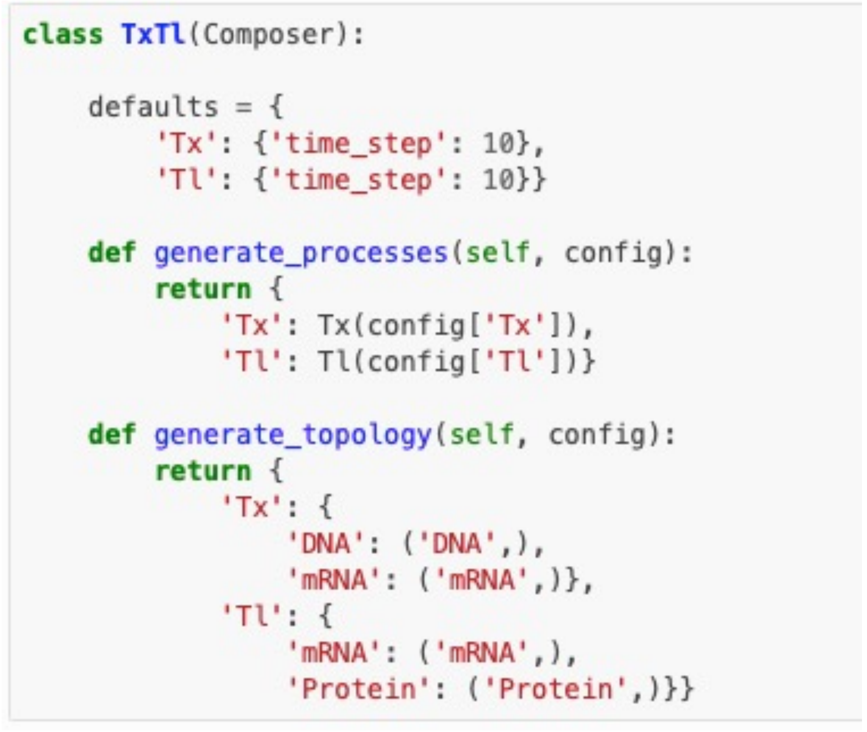




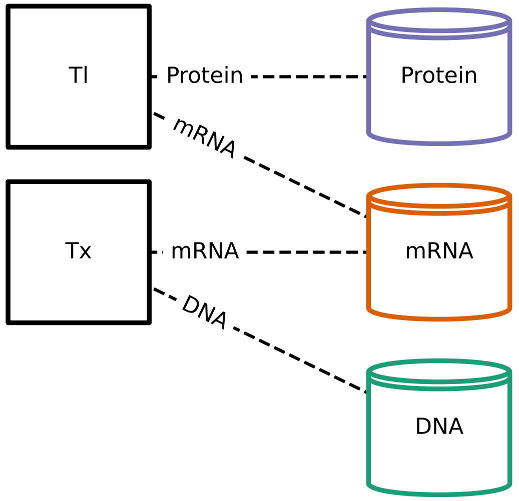
Minimal Composite: Transcription + Translation

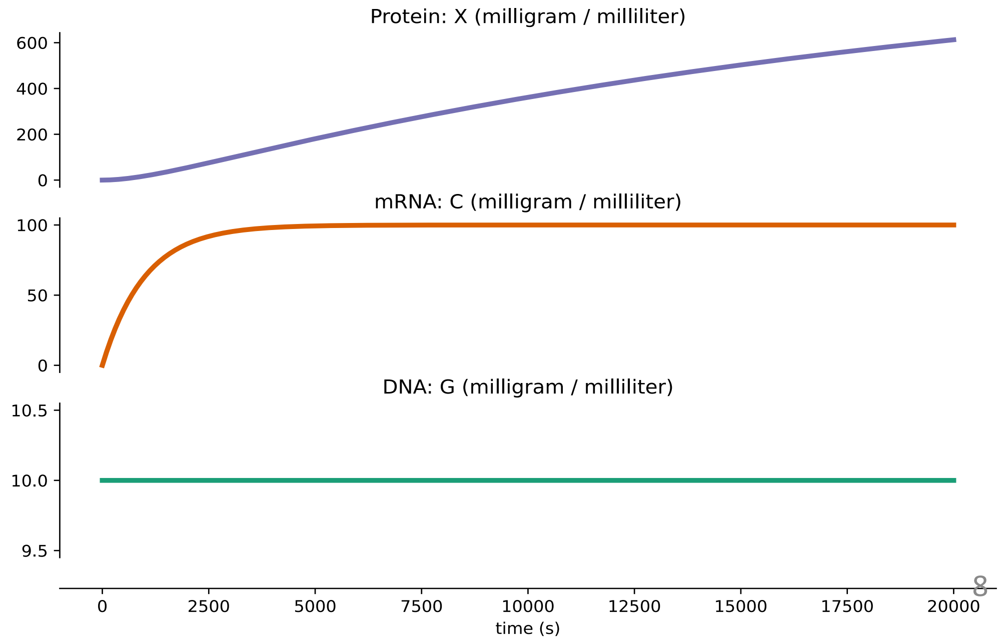
Composition Protocol:

1. **generate_processes***:* initialize processes in a dictionary.
2. **generate_topology***:* declare how process ports are wired together.

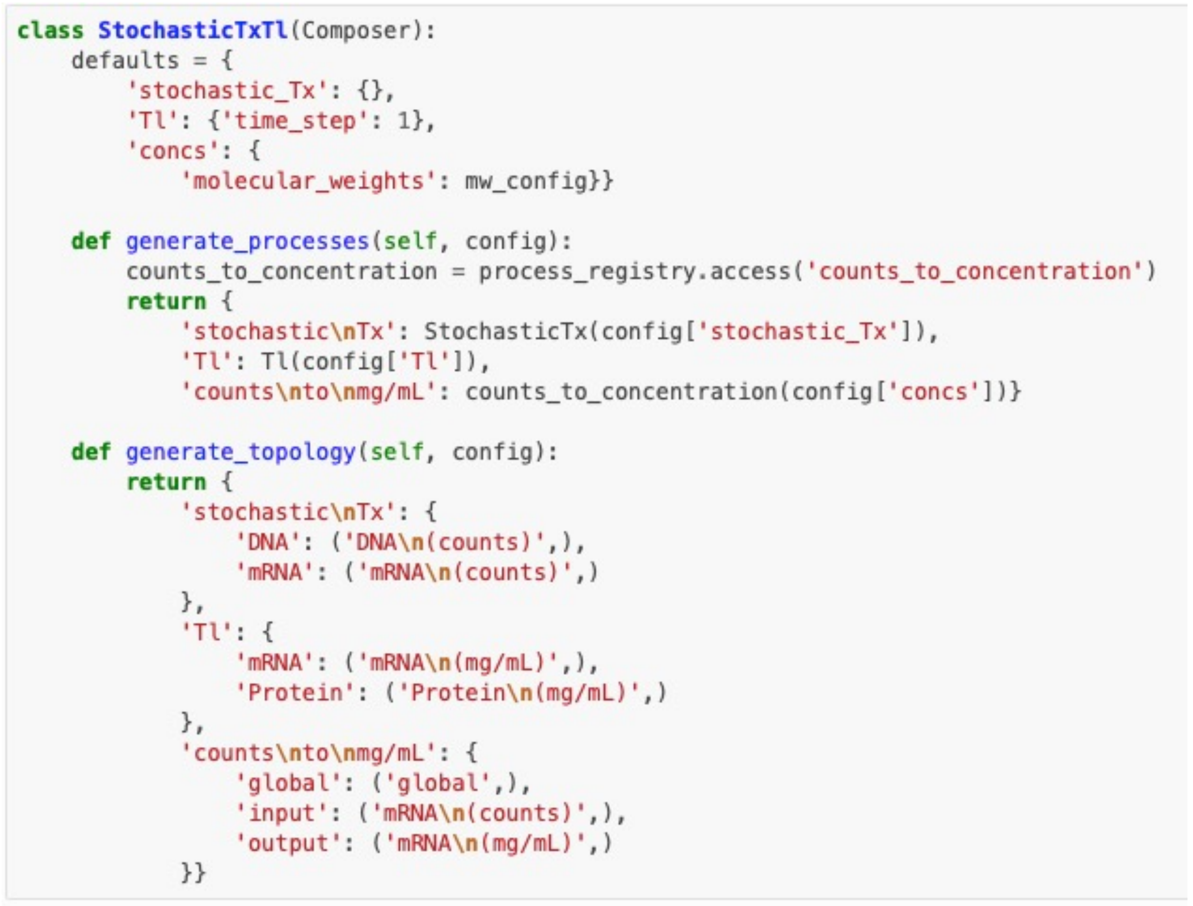


```
class TxTl(Composer):

    defaults = {
        'Tx': {'time_step': 10},
        'Tl': {'time_step': 10}}

    def generate_processes(self, config):
        return {
            'Tx': Tx(config['Tx']),
            'Tl': Tl(config['Tl'])}

    def generate_topology(self, config):
        return {
            'Tx': {
                'DNA': ('DNA',),
                'mRNA': ('mRNA',)},
            'Tl': {
                'mRNA': ('mRNA',),
                'Protein': ('Protein',)}}
```

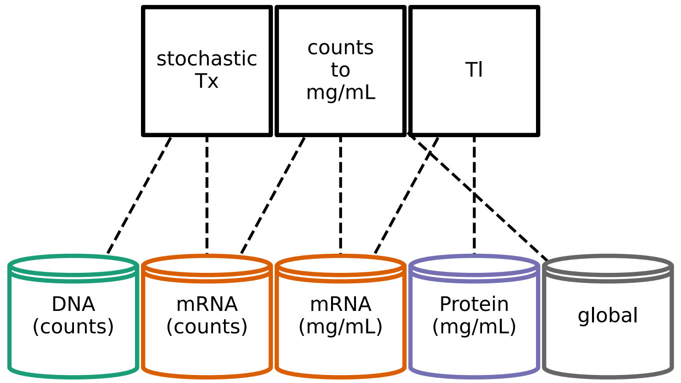


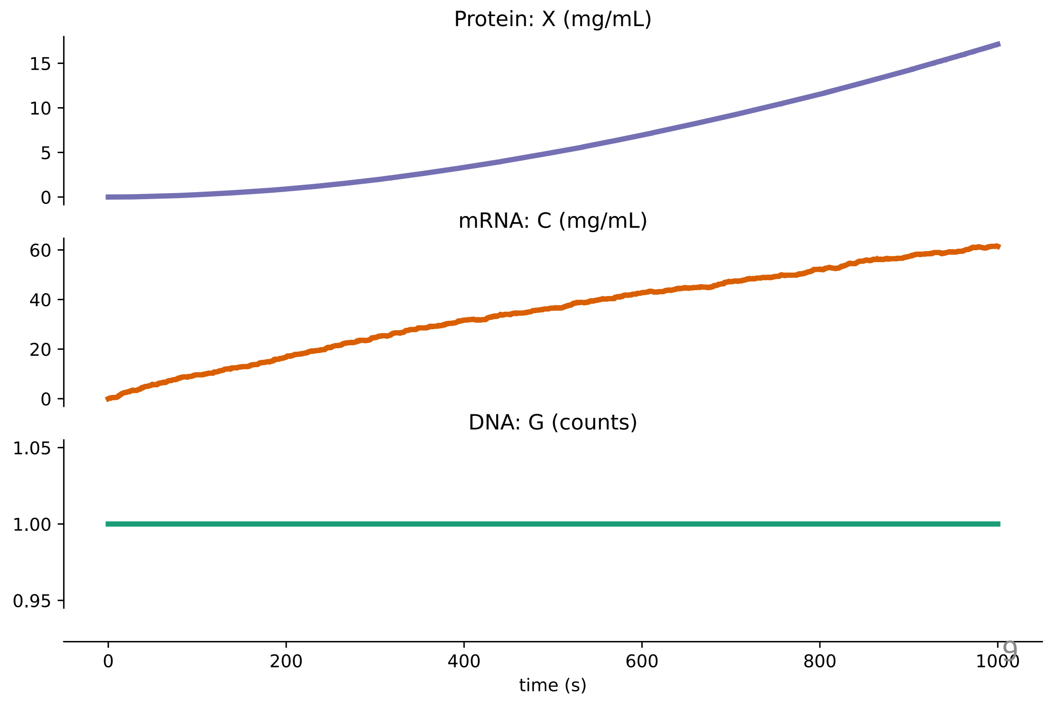


Swap out processes: Stochastic Transcription

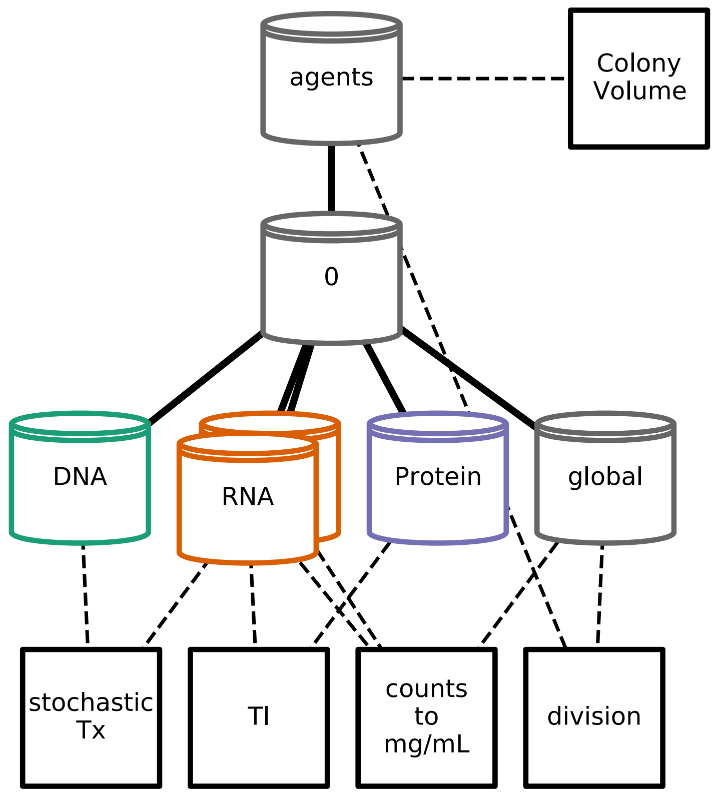


```
class StochasticTxTl(Composer):
    defaults = {
        'stochastic_Tx': {},
        'Tl': {'time_step': 1},
        'concs': {
            'molecular_weights': mw_config}}

    def generate_processes(self, config):
        counts_to_concentration = process_registry.access('counts_to_concentration')
        return {
            'stochastic\nTx': StochasticTx(config['stochastic_Tx']),
            'Tl': Tl(config['Tl']),
            'counts\nto\nmg/mL': counts_to_concentration(config['concs'])}

    def generate_topology(self, config):
        return {
            'stochastic\nTx': {
                'DNA': ('DNA\n(counts)',),
                'mRNA': ('mRNA\n(counts)',)
            },
            'Tl': {
                'mRNA': ('mRNA\n(mg/mL)',),
                'Protein': ('Protein\n(mg/mL)',)
            },
            'counts\nto\nmg/mL': {
                'global': ('global',),
                'input': ('mRNA\n(counts)',),
                'output': ('mRNA\n(mg/mL)',)
            }}
```

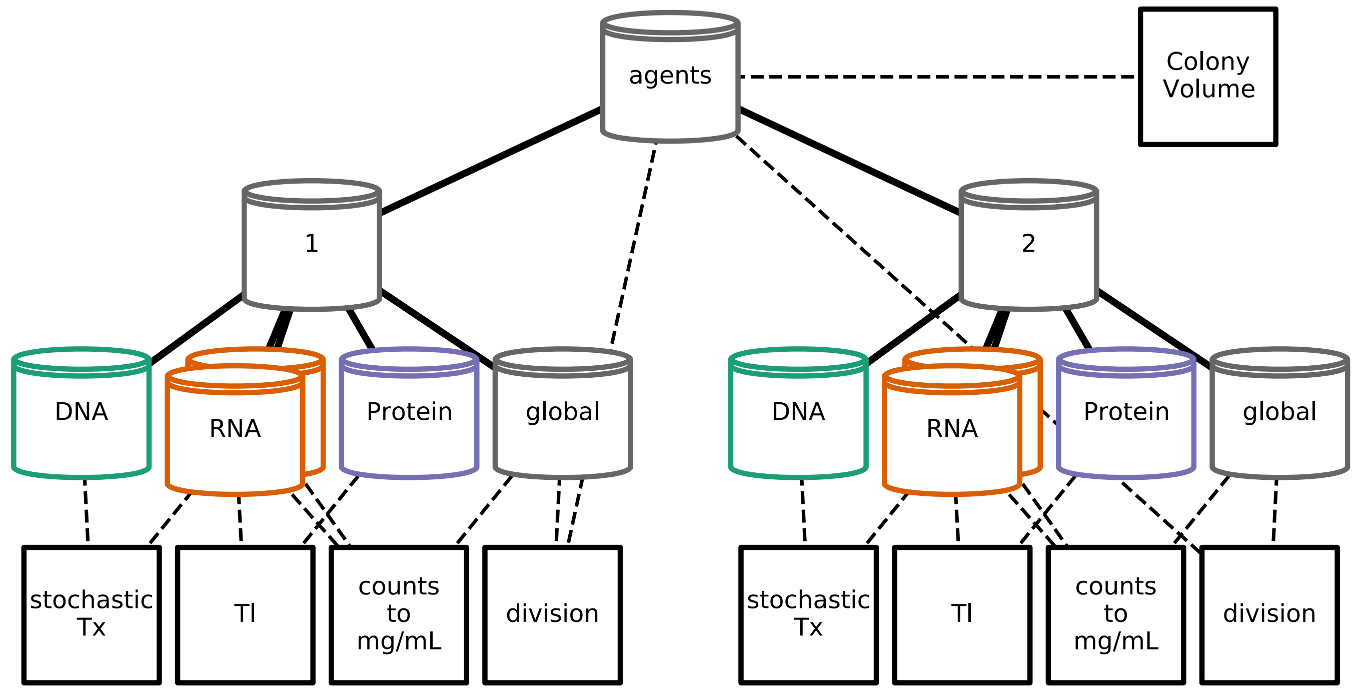




Hierarchical Updates

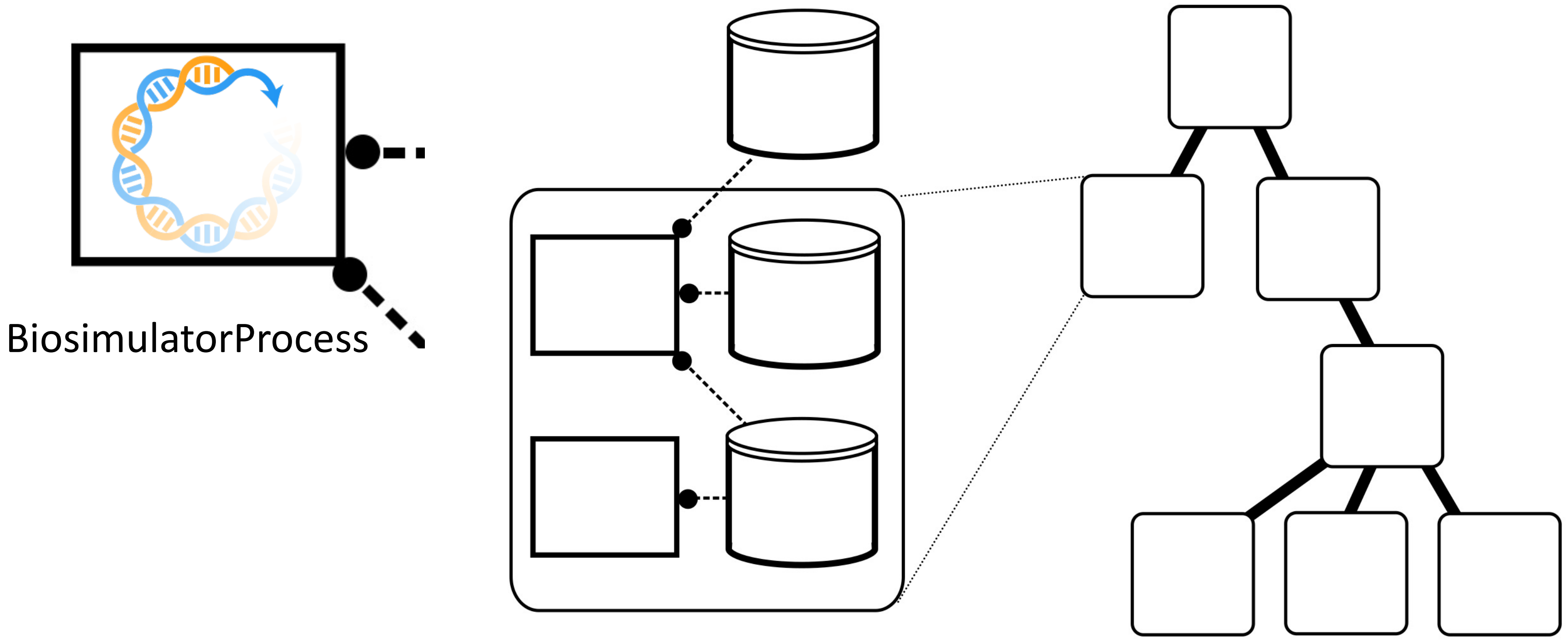


before division



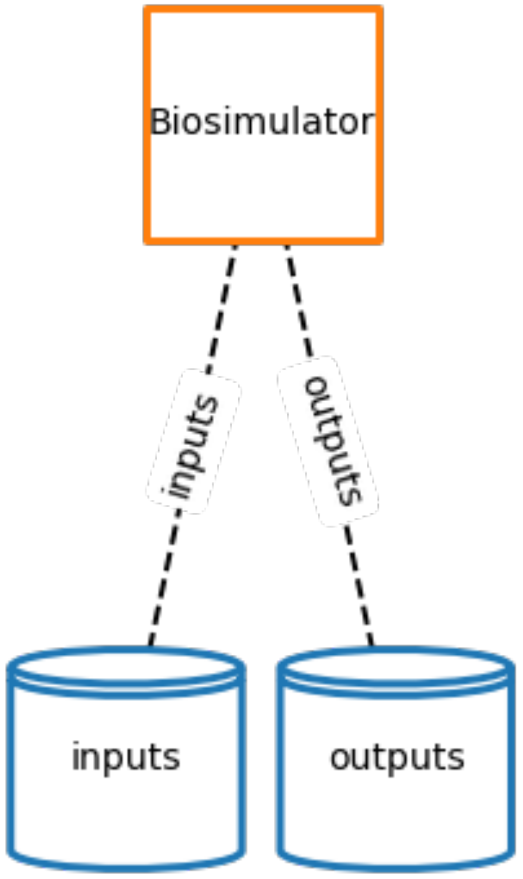
after division

Vivarium-BioSimulators



https://github.com/vivarium-collective/vivarium-biosimulators

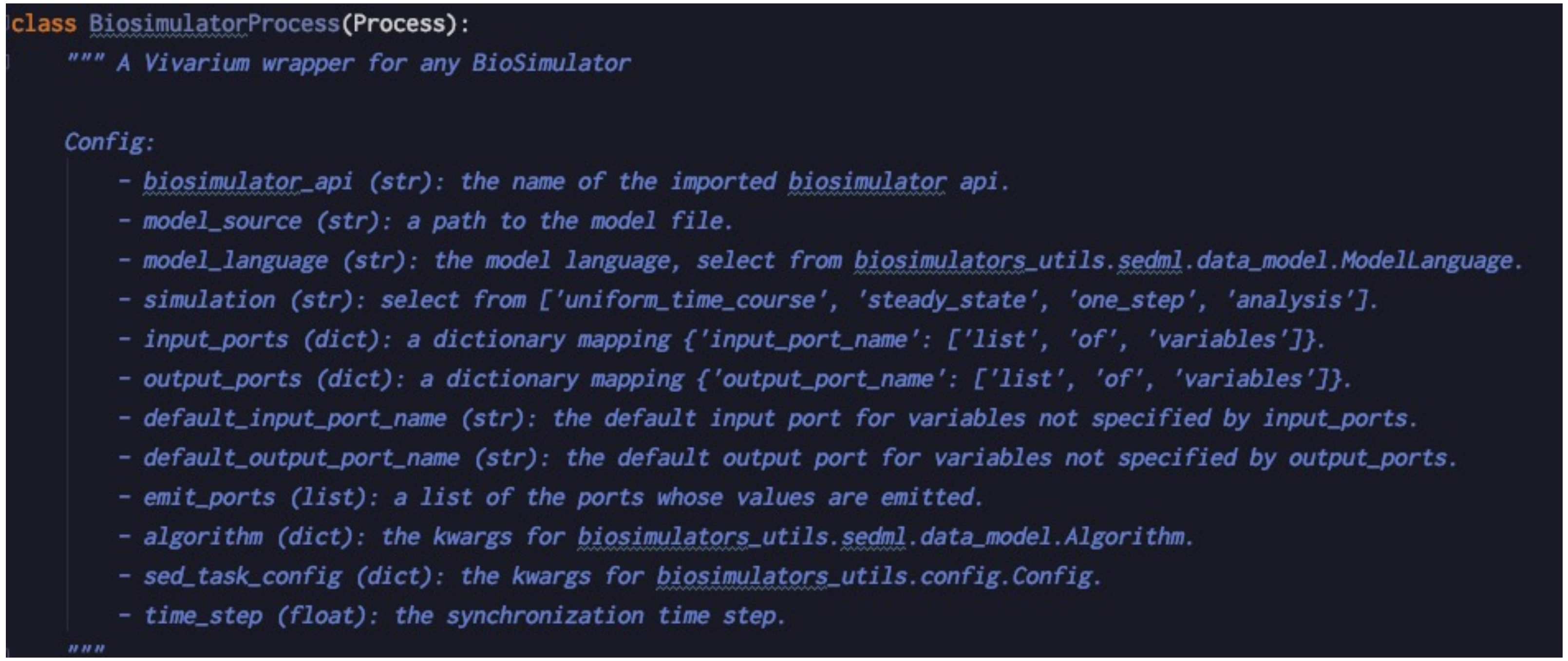
BiosimulatorProcess
1. Parameters
2. Ports
3. next_update



`vivarium_biosimulators.processes.biosimulator_process.BiosimulatorProcess`

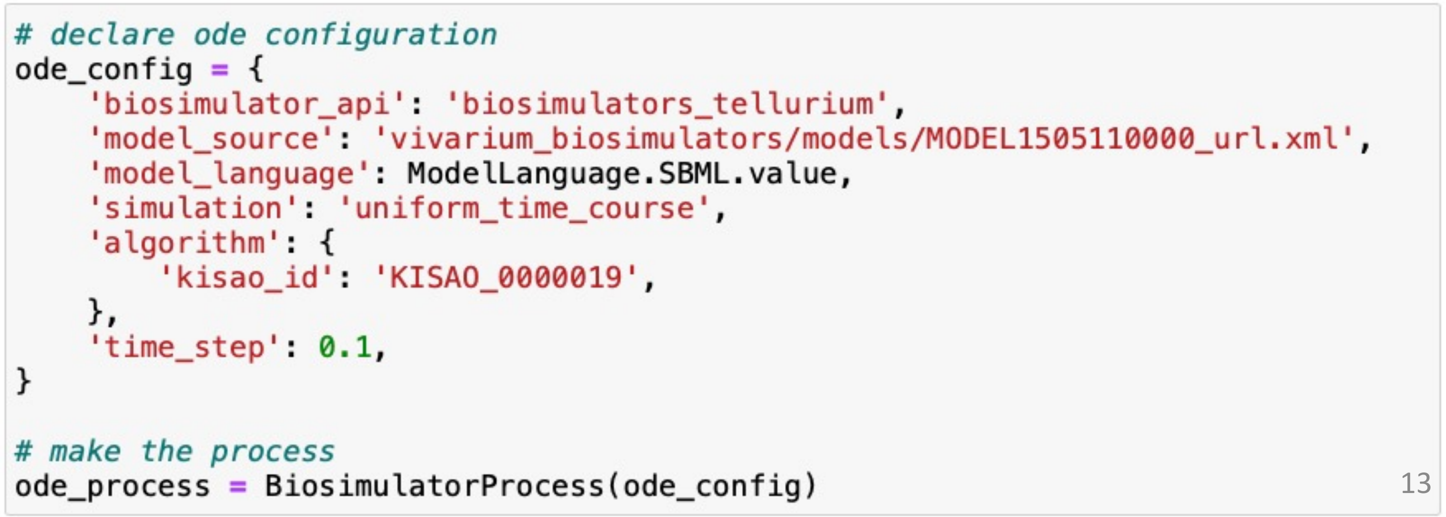
BiosimulatorProcess
> Parameters

Parameters for a BiosimulatorProcess:



```
class BiosimulatorProcess(Process):
    """ A Vivarium wrapper for any BioSimulator

    Config:
        - biosimulator_api (str): the name of the imported biosimulator api.
        - model_source (str): a path to the model file.
        - model_language (str): the model language, select from biosimulators_utils.sedml.data_model.ModelLanguage.
        - simulation (str): select from ['uniform_time_course', 'steady_state', 'one_step', 'analysis'].
        - input_ports (dict): a dictionary mapping {'input_port_name': ['list', 'of', 'variables']}.
        - output_ports (dict): a dictionary mapping {'output_port_name': ['list', 'of', 'variables']}.
        - default_input_port_name (str): the default input port for variables not specified by input_ports.
        - default_output_port_name (str): the default output port for variables not specified by output_ports.
        - emit_ports (list): a list of the ports whose values are emitted.
        - algorithm (dict): the kwargs for biosimulators_utils.sedml.data_model.Algorithm.
        - sed_task_config (dict): the kwargs for biosimulators_utils.config.Config.
        - time_step (float): the synchronization time step.
    """
```

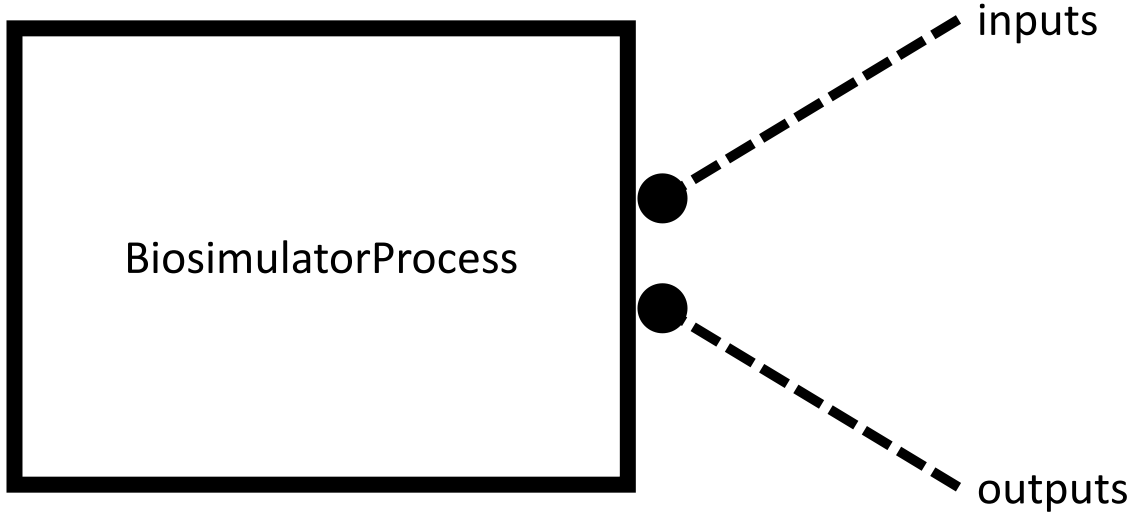
Example:



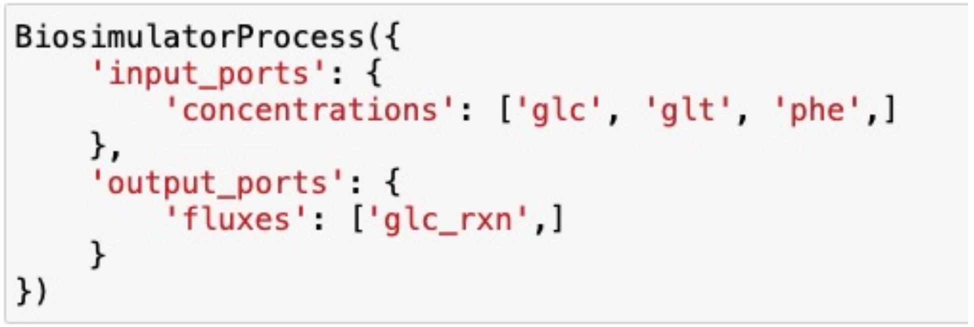
```
# declare ode configuration
ode_config = {
    'biosimulator_api': 'biosimulators_tellurium',
    'model_source': 'vivarium_biosimulators/models/MODEL1505110000_url.xml',
    'model_language': ModelLanguage.SBML.value,
    'simulation': 'uniform_time_course',
    'algorithm': {
        'kisao_id': 'KISAO_0000019',
    },
    'time_step': 0.1,
}

# make the process
ode_process = BiosimulatorProcess(ode_config)
```

BiosimulatorProcess
> ports

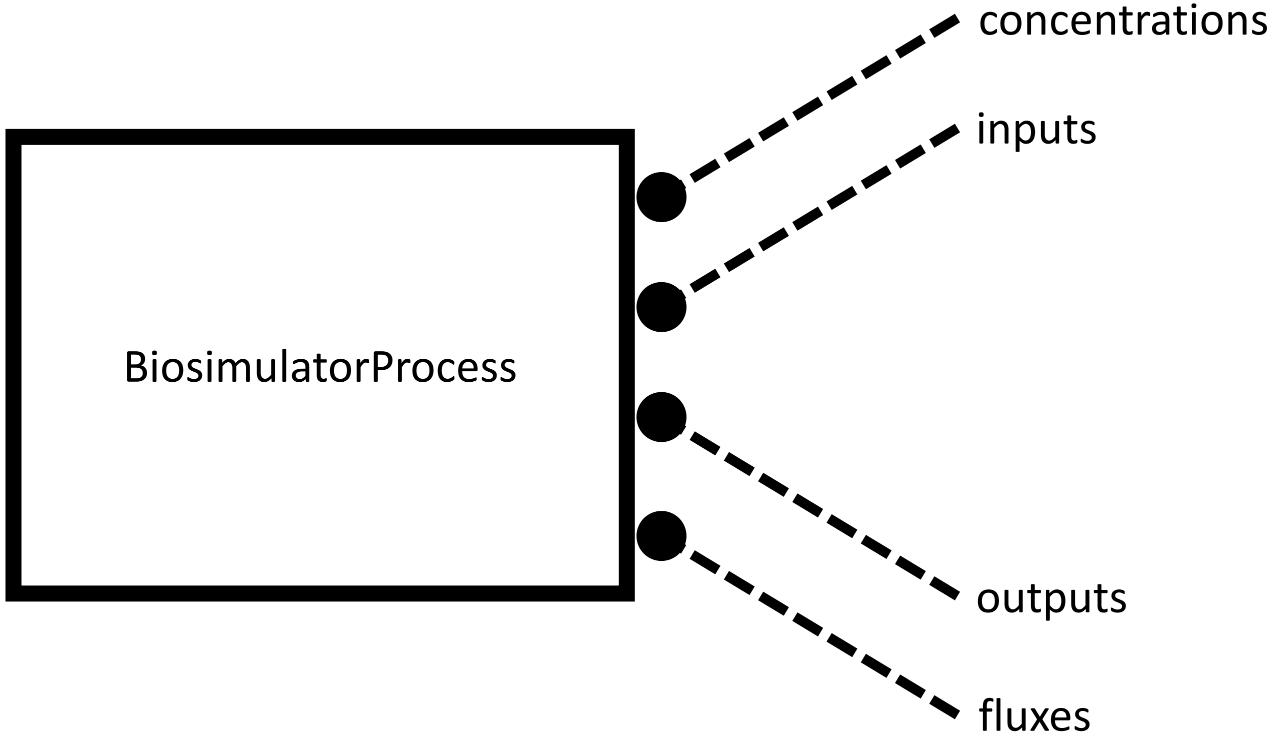
Two ports by default, with inputs and outputs automatically extracted from a model with BioSimulators inspection methods.



BiosimulatorProcess
> ports



```
BiosimulatorProcess({
    'input_ports': {
        'concentrations': ['glc', 'glt', 'phe',]
    },
    'output_ports': {
        'fluxes': ['glc_rxn',]
    }
})
```

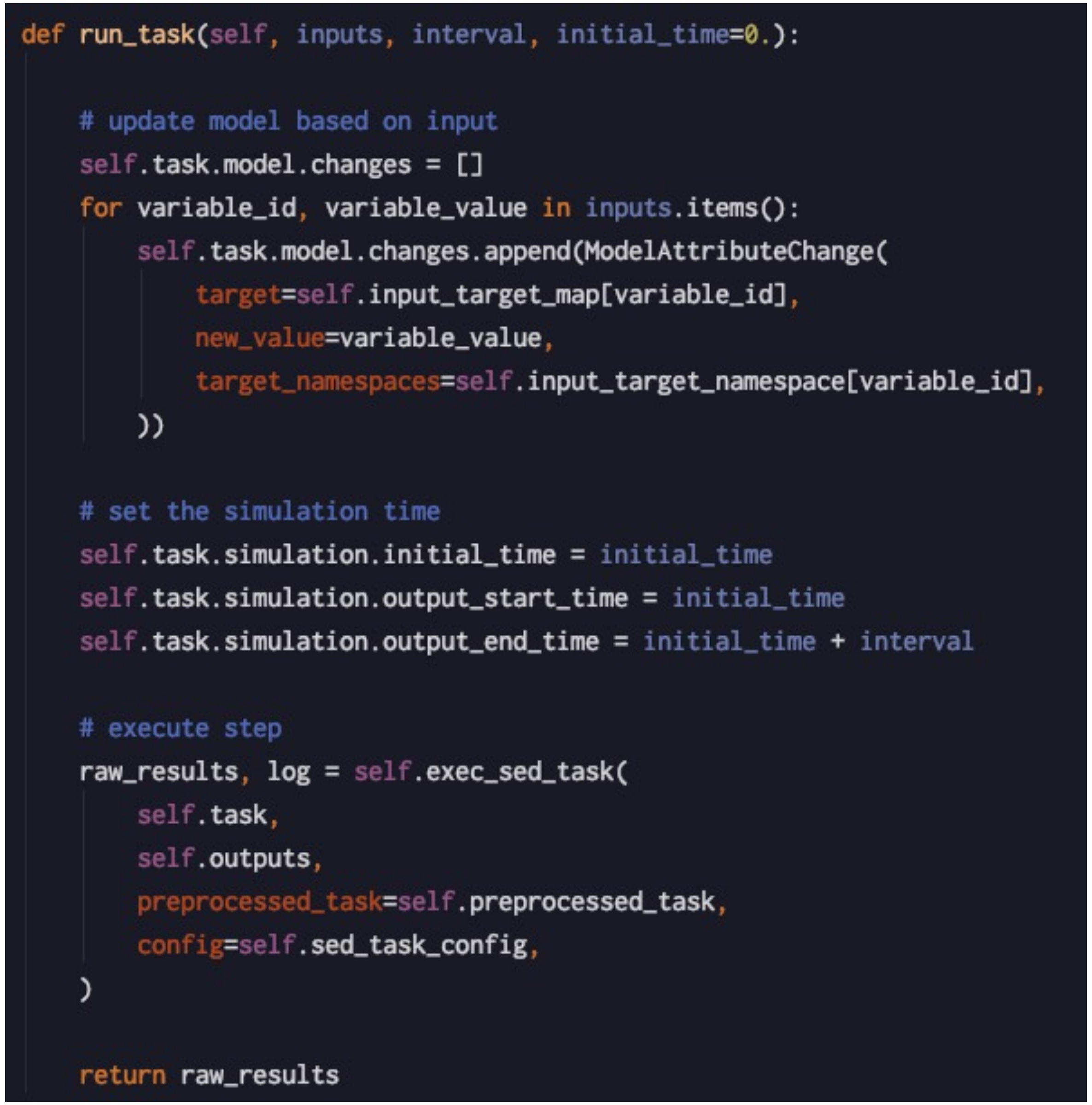


BiosimulatorProcess
> next_update

Three steps of a `next_update()`:

1) take the inputs through the ports and flattens them,
2) pass inputs to `run_task`, get results back.
3) transforms the results to updates and sends them back to the engine.

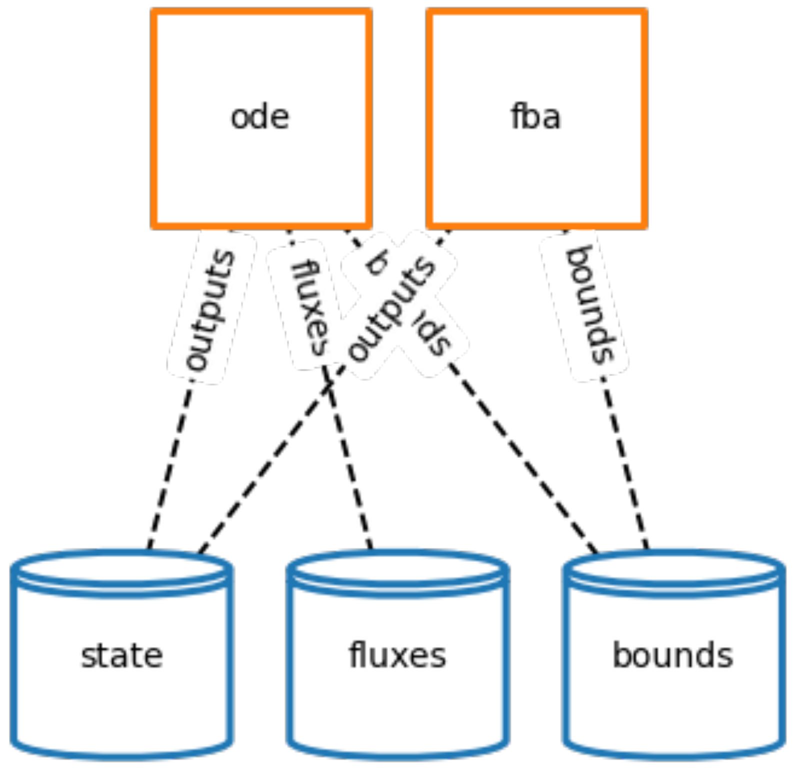
`run_task` can simulate: Tellurium, COBRApy, CMBpy, BioNetGen, COPASI, GillesPy2, LibSBMLSim, RBApy, XPP, and more (thanks BioSimulators)



```python
def run_task(self, inputs, interval, initial_time=0.):

    # update model based on input
    self.task.model.changes = []
    for variable_id, variable_value in inputs.items():
        self.task.model.changes.append(ModelAttributeChange(
            target=self.input_target_map[variable_id],
            new_value=variable_value,
            target_namespaces=self.input_target_namespace[variable_id],
        ))

    # set the simulation time
    self.task.simulation.initial_time = initial_time
    self.task.simulation.output_start_time = initial_time
    self.task.simulation.output_end_time = initial_time + interval

    # execute step
    raw_results, log = self.exec_sed_task(
        self.task,
        self.outputs,
        preprocessed_task=self.preprocessed_task,
        config=self.sed_task_config,
    )

    return raw_results
```

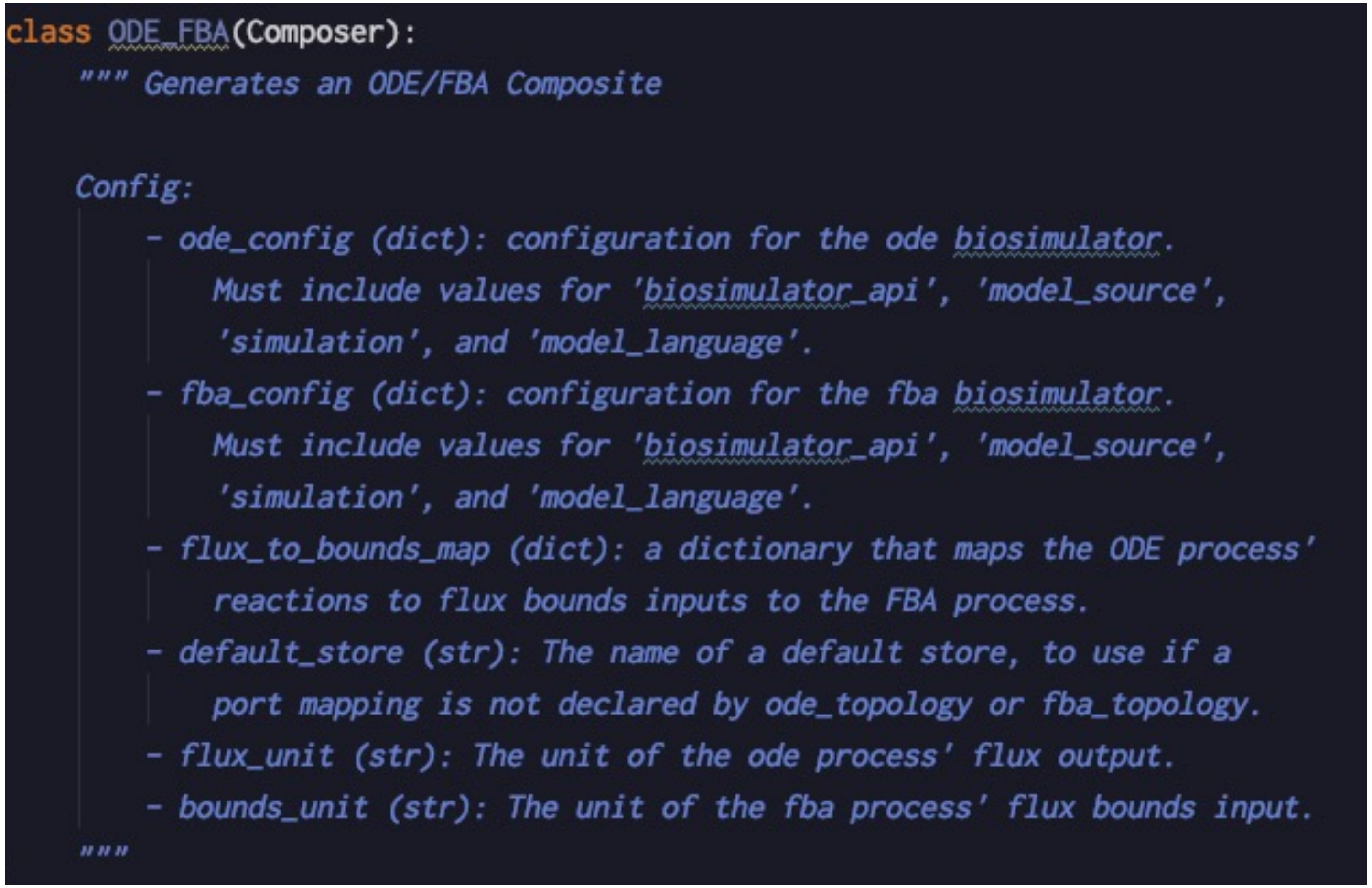
ODE_FBA

1. configuration
2. generate_processes
3. generate_topology



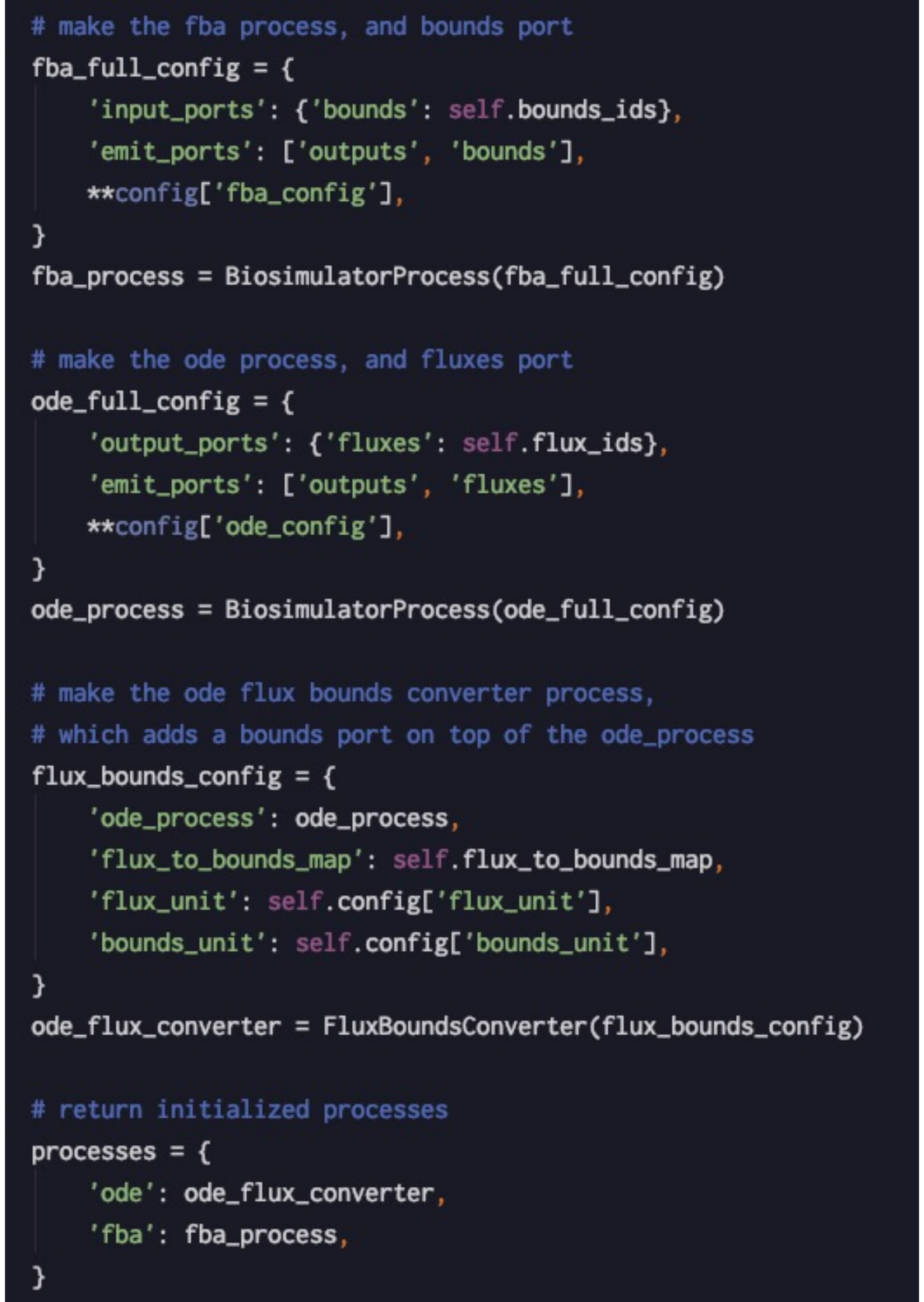
`vivarium_biosimulators.composites.ode_fba.ODE_FBA`

ODE_FBA
> configuration



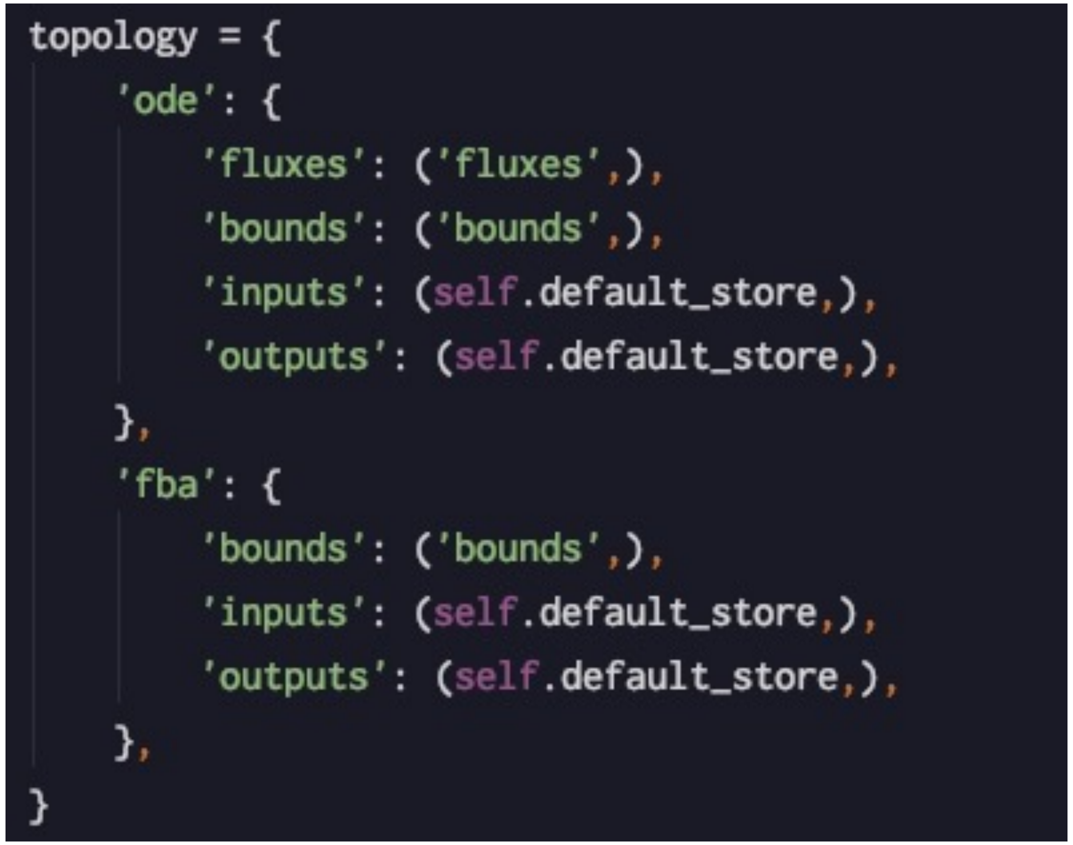
```
class ODE_FBA(Composer):
    """ Generates an ODE/FBA Composite

    Config:
        - ode_config (dict): configuration for the ode biosimulator.
            Must include values for 'biosimulator_api', 'model_source',
            'simulation', and 'model_language'.
        - fba_config (dict): configuration for the fba biosimulator.
            Must include values for 'biosimulator_api', 'model_source',
            'simulation', and 'model_language'.
        - flux_to_bounds_map (dict): a dictionary that maps the ODE process'
            reactions to flux bounds inputs to the FBA process.
        - default_store (str): The name of a default store, to use if a
            port mapping is not declared by ode_topology or fba_topology.
        - flux_unit (str): The unit of the ode process' flux output.
        - bounds_unit (str): The unit of the fba process' flux bounds input.
    """
```

ODE_FBA
> generate_processes



```python
# make the fba process, and bounds port
fba_full_config = {
    'input_ports': {'bounds': self.bounds_ids},
    'emit_ports': ['outputs', 'bounds'],
    **config['fba_config'],
}
fba_process = BiosimulatorProcess(fba_full_config)

# make the ode process, and fluxes port
ode_full_config = {
    'output_ports': {'fluxes': self.flux_ids},
    'emit_ports': ['outputs', 'fluxes'],
    **config['ode_config'],
}
ode_process = BiosimulatorProcess(ode_full_config)

# make the ode flux bounds converter process,
# which adds a bounds port on top of the ode_process
flux_bounds_config = {
    'ode_process': ode_process,
    'flux_to_bounds_map': self.flux_to_bounds_map,
    'flux_unit': self.config['flux_unit'],
    'bounds_unit': self.config['bounds_unit'],
}
ode_flux_converter = FluxBoundsConverter(flux_bounds_config)

# return initialized processes
processes = {
    'ode': ode_flux_converter,
    'fba': fba_process,
}
```

ODE_FBA
> generate_topology



```
topology = {
    'ode': {
        'fluxes': ('fluxes',),
        'bounds': ('bounds',),
        'inputs': (self.default_store,),
        'outputs': (self.default_store,),
    },
    'fba': {
        'bounds': ('bounds',),
        'inputs': (self.default_store,),
        'outputs': (self.default_store,),
    },
}
```

Next up: the demo

- https://github.com/vivarium-collective/vivarium-biosimulators/blob/master/tutorials/ode_fba.ipynb

Thank you!

Vivarium-core:

- Ryan Spangler (Allen Institute for Cell Science)
- Chris Skalnik (Stanford)
- William Poole (Caltech)
- Jerry Morrison (Stanford)
- Shayn Peirce-Cottler (U of Virginia)
- Markus Covert (Stanford)

National Institutes of Health

References:

- **Vivarium-Collective:** https://vivarium-collective.github.io
- **Vivarium Documentation:** https://vivarium-core.readthedocs.io
- **Demo:** https://vivarium-core.readthedocs.io/en/latest/notebooks/Vivarium_interface_basics.html
- **bioRxiv:** Agmon, E., Spangler, R. K., Skalnik, C. J., Poole, W., Peirce, S. M., Morrison, J. H., & Covert, M. W. (2021). Vivarium: an interface and engine for integrative multiscale modeling in computational biology.

email: eagmon@stanford.edu