

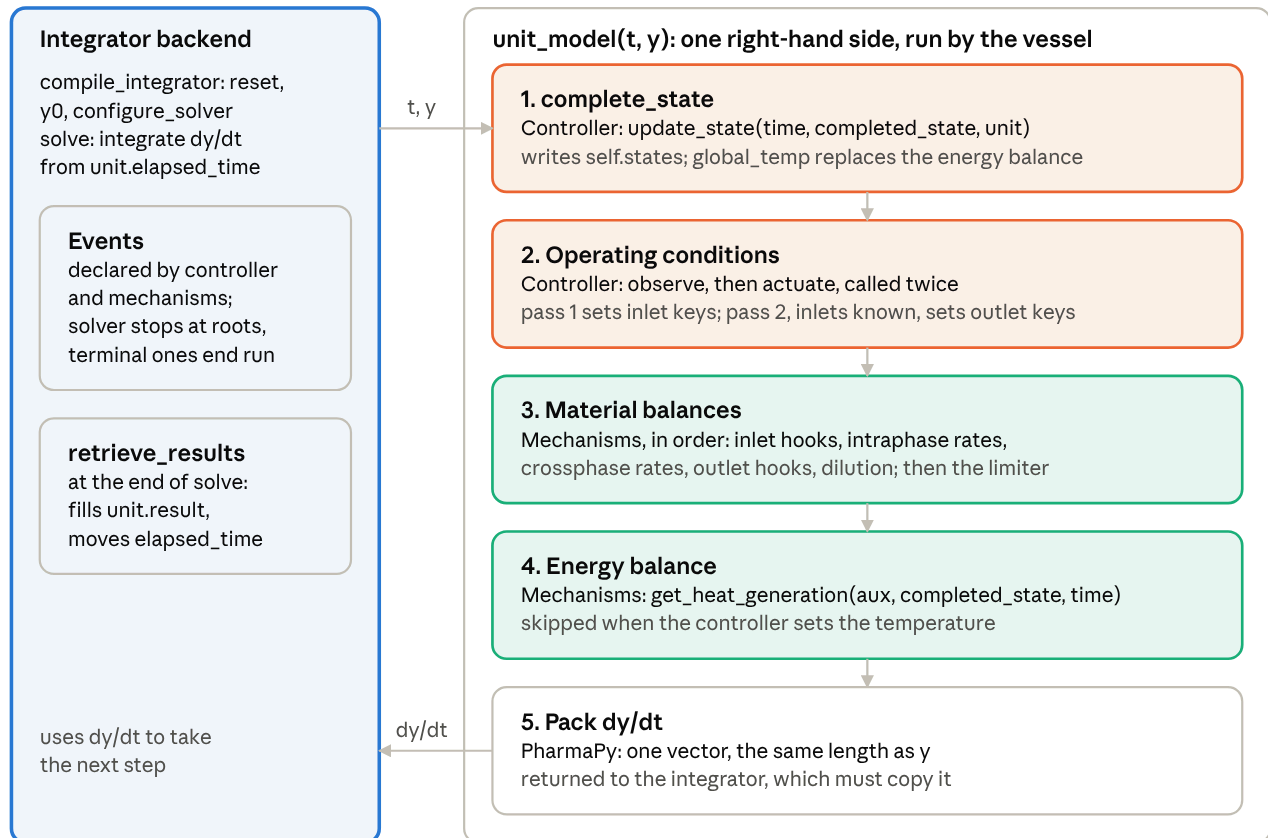
Extending PharmaPy: mechanisms, integrators and controllers

Oct 7, 2026 · @Nagy Group

At a glance

The three pieces never call each other. The integrator only ever calls `unit.unit_model(t, y)`. Inside that call the vessel asks your controller (steps 1 and 2), then your mechanisms (steps 3 and 4), and hands back dy/dt .

Every solver step: the controller acts first, then the mechanisms



Colour = who writes the code: ■ integrator backend ■ controller ■ mechanism ■ PharmaPy (given)

one right-hand-side evaluation · who is called, in what order

So a new mechanism only has to turn the current phases into rates, a new controller only has to set states and stream conditions, and a new integrator only has to drive `unit_model` through time and report the result.

Mechanism: the physics

A mechanism is a piece of physics that returns rates. It never integrates anything itself. On every right-hand-side evaluation the vessel brings its phases up to the current state and calls the mechanism with them. It then adds the returned rates into the balances. Base class: `PharmaPy.Mechanisms.Mechanism` (Mechanisms.py:17).

Pick the kind

Kind	Use it for	Plug it in	The vessel calls
Intraphase	Something inside one phase: a reaction, a profile through a filter cake	<code>unit.intraphase_processes = [IntraPhaseProcess(phaseref=PhaseRef('liquid', 0), mechanism=m)]</code>	<code>get_solver_state_rates(process, phase, time, completed_state)</code>
Crossphase	Material moving between two phases: crystallization, dissolution, mass transfer	<code>unit.phase_connections = [PhaseConnection(source_phaseref=..., sink_phaseref=..., kinetics=..., mechanism=m)]</code>	<code>get_solver_state_rates(source_phase, sink_phase, connection, completed_state, time)</code>
Carried by a phase	State that travels with a phase in and out of the vessel, such as a crystal size distribution	<code>phase.mechanisms = [m]</code>	The inlet, outlet and dilution hooks below

A crossphase mechanism that can run both ways (growth and dissolution) is wrapped once: `rev = ReversibleTransferMechanism(m)`. Then `rev.forward` goes on the liquid→solid connection and `rev.reverse` on the solid→liquid one. Each view zeroes the result whose `net_mass_rate` has the wrong sign (Crystallizers_Refactored.py:80).

What you must write

Member	Returns	Notes
<code>__init__</code> , calling <code>super().__init__()</code>		Sets up the exposed attributes and timers

Member	Returns	Notes
<code>get_solver_state_rates(...)</code>	<code>TransferResult(state_rates, aux, net_mass_rate)</code>	<code>state_rates</code> maps <code>StateKey(name, phaseref)</code> to an array of rates. For species masses use <code>StateKey('mass_j', phaseref)</code> with one rate per species, in kg/s. <code>aux</code> must contain 'process' (intrapphase) or 'connection' (crossphase), because the energy balance reads it back. <code>net_mass_rate</code> is the mass leaving the source in kg/s, and its sign drives <code>ReversibleTransferMechanism</code>

What you may add

Member	Signature or form	When the vessel uses it
<code>solver_states</code>	Tuple of <code>StateVariable(name, dim, units, state_type='diff', limit_negative_inventory=False)</code> ; call <code>self._update_exposed_attributes()</code> after it	At set-up, each becomes an extra ODE state (the FVM distribution is one). Return its rate in <code>state_rates</code> under <code>StateKey(name, process.phaseref)</code> . Its current value is <code>completed_state[StateKey(name, phaseref)]</code>
<code>output_states</code>	<code>StateVariable(..., state_type='post', compute_value=f)</code>	After the solve, to put derived results (supersaturation, moments) on <code>unit.result</code>
<code>get_heat_generation(aux, completed_state, time)</code>	Watts; positive heats the vessel	In the energy balance, with the <code>aux</code> you returned. Default 0
<code>get_overrides()</code>	<code>{phase attribute: callable}</code> , or <code>{}</code>	When the mechanism is assigned to <code>phase.mechanisms</code> . Required in that case (see the checklist)
<code>get_inlet_contributions(stream_phase, vessel_phase, amount, completed_state)</code>	<code>{StateKey: rate}</code> , added	Once per inlet, for mechanisms in <code>phase.mechanisms</code> . <code>amount</code> is the whole stream's volumetric flow
<code>get_outlet_contributions(stream_phase, vessel_phase, amount, completed_state)</code>	<code>{StateKey: rate}</code> , subtracted	Once per outlet. <code>amount</code> is this phase's outlet flow. The base class names the first two parameters the other way round; the vessel passes them in this order (MultiPhaseVessel.py:2663)
<code>get_dilution_contributions(net_vol_flow, completed_state)</code>	<code>{StateKey: rate}</code>	Only on vessels with <code>dilutes_intensive_states = True</code> , for states stored per unit volume

Member	Signature or form	When the vessel uses it
<code>get_events(unit)</code>	List of <code>StateEvent</code>	Once at compile, so the integrator stops at a regime change (Mechanisms.py:776 does this at zero supersaturation)
<code>get_solver_state_residuals(time, completed_state, unit)</code>	<code>{StateKey: residual}</code>	Only for <code>state_type='alg'</code> states. Needs a DAE backend (AssimuloDAEBackend, DiffeqpyBackend)
<code>apply_inlet_state(name, value), reset()</code>		A feed that carries one of your states, and a reset before each solve

`positivity_horizon` is set on every mechanism by the vessel (a look-ahead window in seconds). Read it if you guard against a species running out, as `ReactionMechanism` does.

Minimal example (tested)

This is a first-order isomerisation $A \rightarrow B$, attached to a batch reactor with no `RxnKinetics`. Over one hour the mass of A matches the analytic $m_{A0} \cdot e^{(-kt)}$ to $5e-5$ (the solver tolerance), and the total mass is conserved to $1e-16$ kg.

```
import numpy as np
from PharmaPy.Phases_Refactored import LiquidPhase
from PharmaPy.Reactors_Refactored import BatchReactor
from PharmaPy.IntegratorBackends import ScipyBackend
from PharmaPy.Mechanisms import Mechanism
from PharmaPy.DataClasses import IntraPhaseProcess, PhaseRef, StateKey,
TransferResult

class Isomerisation(Mechanism):
    """A --> B, first order in A, inside the phase it is attached to."""

    def __init__(self, k, reactant='A', product='B'):
        super().__init__()
        self.k = k # 1/s
        self.reactant, self.product = reactant, product

    def get_overrides(self):
        return {} # takes over no phase
attribute

    def get_solver_state_rates(self, process, phase, time, completed_state):
```

```

        i = phase.name_species.index(self.reactant)
        j = phase.name_species.index(self.product)

        moles_a = max(phase.mole_conc[i], 0.0) * phase.vol      # kmol
        rate = self.k * moles_a * phase.mw[i]                  # kg/s of A
converted
        mass_rates = np.zeros(phase.num_species)               # kg/s, one per
species
        mass_rates[i] = -rate
        mass_rates[j] = +rate                                  # A and B have
the same MW

        return TransferResult(
            state_rates={StateKey('mass_j', process.phaseref): mass_rates},
            aux={'phase': phase, 'process': process},            # the energy
balance reads 'process'
            net_mass_rate=0.0,                                   # nothing leaves
the phase
        )

reactor = BatchReactor(integrator=ScipyBackend(), isothermal=True)
reactor.Phases = LiquidPhase('data/compounds_workshop.json', temp=333.15,
vol=1e-3,
                                mole_conc=[1, 0, 0, 0, 0], name_solv='solvent')
reactor.intraphase_processes = [
    IntraPhaseProcess(phaseref=PhaseRef('liquid', 0),
mechanism=Isomerisation(k=1e-3)),
]
reactor.solve_unit(time_grid=np.linspace(0, 3600, 61), verbose=False)
mass_a = reactor.result.mass_j_liquid0[:, 0]

```

Phase attributes you can read inside a mechanism include `mole_conc` (kmol/m³, which equals mol/L), `mass_frac`, `mw`, `vol`, `temp`, `num_species` and `name_species`. Set the attachment after `Phases`, because assigning `Phases` rebuilds the vessel's state layout.

Worked examples in the source

- `ReactionMechanism` (Mechanisms.py:318): an intraphase mechanism with heat of reaction.

- `DeliquoringMechanism` (`SolidLiquidSep_Refactored.py:498`): intraphase, and owns its own profile states (`solver_states`).
- `PopulationBalanceMechanism` (`Mechanisms.py:440`) and `OneDFVMMechanism` (:1170): crossphase plus carried by a phase, with all of the hooks, events and overrides.

Integrator backend: the solver

An integrator backend takes a unit and its time horizon and drives the unit's right-hand side through time. It knows nothing about phases or chemistry. Its whole view of the model is one state vector `y`, one function `unit.unit_model(time, states)`, and a list of events. Base class: `PharmaPy.IntegratorBackends.IntegratorBackend` (`IntegratorBackends.py:45`). A unit takes one through its constructor: `BatchReactor(integrator=MyBackend())`. With none given, a unit uses `ScipyBackend()`.

What you must write

Method	Signature	Must do
<code>compile_integrator</code>	<code>(unit, eval_sens=False, jac_v_prod=False, options=None, verbose=True, any_event=True)</code>	Run the set-up sequence below once and return the initial state vector. Set <code>self._compiled = True</code>
<code>solve</code>	<code>(unit, runtime=None, time_grid=None, eval_sens=False, jac_v_prod=False, verbose=True, options=None, any_event=True)</code>	Compile if needed, integrate, call <code>unit.retrieve_results(time, states)</code> , and return <code>(time, states)</code>

`unit.solve_unit(runtime=..., time_grid=..., **kwargs)` calls `unit.compile_structure()` and then `integrator.solve(unit, ...)`, forwarding any extra keywords.

The contract with the unit

This is the call sequence `ScipyBackend` (IntegratorBackends.py:1574–1641) and `AssimuloBackend` (:173–305) both follow:

1. `unit.reset()`
2. `y0 = unit.create_solver_init_states()`, the initial vector built from the current phases.
3. `unit.save_initial_solver_state(y0, unit.elapsed_time)`
4. `self.check_algebraic_support(unit)`, which refuses a unit with algebraic states unless `supports_algebraic = True`.
5. `unit.configure_solver()`, where the unit may call `self.set_linear_solver('krylov' | 'dense' | 'sparse')`. That call is a hint; ignore it if your solver has no such choice.
6. Integrate `dy/dt = unit.unit_model(time=t, states=y)` from `unit.elapsed_time` to `unit.elapsed_time + runtime`, or to `time_grid[-1]`. **Copy what `unit_model` returns** (`np.array(..., dtype=float)`), because the vessel refills the same buffer on every call.
7. Events (optional, but without them the solver integrates straight through switching surfaces):
 - `unit.compiled_events` is a tuple of `StateEvent(name, function, direction, terminal, source)`.
 - `unit.evaluate_events(t, y)` returns all of their values at once, cached per point.
 - After a root, call `unit.handle_event(t, indices)`; it returns True when a terminal event fired and the run should stop.
8. `unit.retrieve_results(time, states)`, with `time` 1-D and `states` 2-D with **one row per time** ($n_t \times n_y$). It fills `unit.result` and moves `unit.elapsed_time` to the last time, so a second `solve` continues the run.

What you may add

Member	Purpose
<code>fast_solve(unit, runtime=None, time_grid=None, verbose=True)</code>	Re-solve without recompiling; the default just calls <code>solve</code>

Member	Purpose
<code>set_linear_solver(kind)</code>	Map the neutral name onto your solver's own linear solver
<code>supports_algebraic = True</code>	Accept <code>state_type='alg'</code> states. The residual rows are <code>unit.solver_state_collection.algebraic_mask</code>
Sensitivities	Raise <code>NotImplementedError</code> on <code>eval_sens=True</code> unless you compute them (only <code>AssimuloBackend</code> does)

Minimal example (tested)

This backend wraps scipy's explicit RK45 and handles terminal events. On the two-reaction batch reactor from the workshop it agrees with `ScipyBackend` to 1.7e-7 mol/L. A terminal `StateEvent` (stop when $C_A = 0.5$) ends the run at $C_A = 0.5000$, and a continuous reactor whose controller declares an event also runs on it.

```
import numpy as np
from scipy.integrate import solve_ivp
from PharmaPy.IntegratorBackends import IntegratorBackend

class RK45Backend(IntegratorBackend):
    """Explicit Runge-Kutta (scipy RK45). Terminal events only, no
    sensitivities, no DAEs."""

    def __init__(self, rtol=1e-6, atol=1e-10):
        super().__init__()
        self.rtol, self.atol = rtol, atol

    def compile_integrator(self, unit, eval_sens=False, jac_v_prod=False,
                          options=None, verbose=True, any_event=True):
        if eval_sens:
            raise NotImplementedError('RK45Backend computes no
sensitivities')
        self.check_algebraic_support(unit)          # refuses a unit with
algebraic states
        unit.reset()
        states_init = unit.create_solver_init_states()
```

```

        unit.save_initial_solver_state(states_init, unit.elapsed_time)
        unit.configure_solver()                # the unit may call
self.set_linear_solver(...)
        self._compiled = True
        return states_init

def make_events(self, unit):
    events = []
    for index, event in enumerate(unit.compiled_events):
        def g(t, y, index=index):
            return unit.evaluate_events(t, y)[index]
        g.terminal, g.direction = bool(event.terminal), event.direction
        events.append(g)
    return events

def solve(self, unit, runtime=None, time_grid=None, eval_sens=False,
          jac_v_prod=False, verbose=True, options=None, any_event=True):
    if not self._compiled:
        self.compile_integrator(unit, eval_sens=eval_sens)

    t0 = unit.elapsed_time                    # a second solve continues
from here
    t1 = t0 + runtime if runtime is not None else time_grid[-1]
    y0 = unit.create_solver_init_states()

    def rhs(t, y):                            # copy: the vessel reuses
its output buffer
        return np.array(unit.unit_model(time=t, states=y), dtype=float)

    sol = solve_ivp(rhs, (t0, t1), y0, method='RK45', t_eval=time_grid,
                    events=self.make_events(unit), rtol=self.rtol,
atol=self.atol)
    if sol.status == -1:
        raise RuntimeError(sol.message)

    time, states = sol.t, sol.y.T              # one row of states per
time
    unit.retrieve_results(time, states)
    return time, states

reactor = BatchReactor(integrator=RK45Backend(), isothermal=True)

```

scipy stops on a terminal event by itself, so this backend never needs `handle_event`. Non-terminal events are only recorded here. A backend that should stop and restart at them, which is what keeps a stiff solver's steps long across a regime change, has to re-launch from each root. `ScipyBackend.fast_solve` does that, with a dead band so it does not fire twice on the same root.

Worked examples in the source

- `ScipyBackend` (IntegratorBackends.py:1184): no install needed, restarts at events, and stops with a diagnosis if it stalls.
- `AssimuloBackend` (:143): CVode, sensitivities, Krylov solver.
- `AssimuloDAEBackend` (:392) and `DiffeqpyBackend` (:615): algebraic states, through a residual or a mass matrix.

Controller: the operating policy

A controller decides what the operator would decide: the temperature to hold, how much to feed, and how fast to draw off. A vessel has exactly one, passed as `controller=` in its constructor. Anything that drives several variables does all of it in that one class. Base class: `PharmaPy.ProcessControl_Refactored.Controller` (ProcessControl_Refactored.py:10). The default `Controller()` does nothing.

It has two outputs, and each has its own dictionary:

- **`self.states`**: values that replace a state. The common one is `StateKey('global_temp')`. A controlled temperature removes the energy balance, so `Utility` is not consulted.
- **`self.operating_conditions`**: values that set the streams, keyed by `OperatingKey(name, connection, phaseref, port)`.

What you write

Method	Signature	When the vessel calls it	What it does
<code>__init__</code>	Must call <code>super().__init__()</code>	Once	Creates <code>self.states</code> and <code>self.operating_conditions</code> . Put the <code>global_temp</code> key into <code>self.states</code> here if you control temperature: the vessel decides at set-up whether to solve an energy balance by checking for that key
<code>update_state</code>	<code>(time, completed_state, unit)</code>	Every right-hand side, first	Write <code>self.states[StateKey('global_temp')] = T</code>
<code>actuate</code>	<code>(time, completed_state, unit, resolved_inlets=None, resolved_outlets=None)</code>	Every right-hand side, twice	First pass, with <code>resolved_inlets</code> is <code>None</code> : set the inlet keys. Second pass, with the actual inlets known: set the outlet keys. The vessel merges the two passes (MultiPhaseVessel.py:1732)
<code>observe</code>	Same arguments as <code>actuate</code>	Before each <code>actuate</code>	Measurements and internal bookkeeping only; set no outputs here
<code>get_events</code>	<code>(unit)</code> returns a list of <code>StateEvent</code>	Once at compile	Declare every switch you make (<code>function(time, completed_state, unit)</code>), with a sign change at the switch)
<code>reset</code>	<code>()</code>	Before each solve	Restore anything <code>__init__</code> set up, as <code>TemperatureProfileMixin.reset</code> does

Everything is optional; override only what you need. The inherited `compute_states` and `compute_operating_conditions` clear the dictionaries and call your methods, so do not override those.

Operating keys

Key	Meaning
<code>OperatingKey('vol_flow', connection=0, port='outlet')</code>	Total outlet flow of outlet connection 0 (m ³ /s), split over its phases by volume

Key	Meaning
<code>OperatingKey('vol_flow', connection=0, phaseref=PhaseRef('liquid', 0), port='inlet')</code>	Feed rate of the liquid on inlet connection 0. It must name the phase: a connection-level inlet <code>vol_flow</code> is ignored (MultiPhaseVessel.py:2431)
<code>OperatingKey(name, connection=i, phaseref=..., port='inlet' or 'outlet')</code>	Any other keyword <code>updatePhase</code> accepts (<code>temp</code> , <code>pres</code> , a composition) for that phase of that stream. It is applied last, so it overrides the feed

Minimal example (tested)

This controller holds the temperature, doubles the feed of a CSTR at one hour, and holds the volume with the outlet. Its result:

- before the step the outlet runs at $0.975\times$ the original feed (the reaction shrinks the liquid);
- at the end it runs at $1.956\times$;
- the volume stays within 0.15 % of 1 L.

With `phaseref=None` on the inlet key, the feed never changes.

```
from PharmaPy.ProcessControl_Refactored import Controller
from PharmaPy.DataClasses import OperatingKey, PhaseRef, StateEvent, StateKey

class FeedStep(Controller):
    """Hold the temperature, double the feed at t_step, hold the volume with
    the outlet."""

    def __init__(self, temp, base_flow, t_step, tau=60.0):
        super().__init__() # creates
        self.states, self.operating_conditions =
            self.temp, self.base_flow, self.t_step, self.tau = temp, base_flow,
            t_step, tau
            self.target_volume = None
            self.states[StateKey('global_temp')] = temp # registered now:
            replaces the energy balance

    def update_state(self, time, completed_state, unit):
```

```

        self.states[StateKey('global_temp')] = self.temp

    def observe(self, time, completed_state, unit, resolved_inlets=None,
resolved_outlets=None):
        if self.target_volume is None:
            self.target_volume = unit.Phases.vol           # hold the
starting volume

    def actuate(self, time, completed_state, unit, resolved_inlets=None,
resolved_outlets=None):
        if resolved_inlets is None:                        # pass 1: inlet
keys
            flow = self.base_flow * (2.0 if time >= self.t_step else 1.0)
            key = OperatingKey('vol_flow', connection=0,
phaseref=PhaseRef('liquid', 0), port='inlet')
            self.operating_conditions[key] = flow
        else:                                              # pass 2: outlet
keys, actual inlet known
            inlet = sum(t.stream_phase.vol_flow for c in resolved_inlets for
t in c)
            outlet = inlet + (unit.Phases.vol - self.target_volume) /
self.tau
            self.operating_conditions[OperatingKey('vol_flow', connection=0,
port='outlet')] = max(outlet, 0.0)

    def get_events(self, unit):
        # the feed jumps at t_step: let the integrator stop and restart there
        return [StateEvent(name='feed_step', function=lambda time,
completed_state, unit: time - self.t_step,
                        direction=1, terminal=False, source=self)]

cstr = ContinuousReactor(integrator=ScipyBackend(),
                        controller=FeedStep(333.15, base_flow=1e-3 / 1800,
t_step=3600))

```

The `max(outlet, 0.0)` is a corner the integrator can still trip on when the outlet nears zero. `DefaultContinuousVesselVolume` replaces it with a smooth `soft_floor` and declares an event at the sign change. Copy that if your level can fall to zero.

Worked examples in the source

- `SimpleTemperatureController` and `TemperatureProfileMixin` (`ProcessControl_Refactored.py:101-149`): temperature from a function of time.
- `DefaultContinuousVesselVolume` (:152): level control with a gain set from a settling time, a smooth floor, and an event.
- `ContinuousVesselController` (:349): the two combined; the default for continuous units.
- `TankLevelController` (:397): the smallest outlet controller, an on/off switch paired with its event.
- `ComplexController` (:460): a template that splits the two passes into `actuate_inlets` and `actuate_outlets`.
- `JacketProfile` in the workshop's `SyntaxExamples.ipynb`: sets `unit.Utility.temp_in` in `update_state` and leaves the energy balance on.

Checklist and pitfalls

Go through this before handing a new piece to anyone. Each item is a mistake that fails silently or with an unhelpful error.

Mechanism

- ☐ `super().__init__()` is called.
- ☐ Every rate is keyed by a `StateKey` the vessel knows: `StateKey('mass_j', process.phaseref)`, or the `phaseref` of the connection's source or sink. **A key the vessel does not know is dropped without a warning**, and your physics then does nothing.
- ☐ Species-mass rates are in kg/s, with one entry per species (`phase.num_species`).
- ☐ `aux` contains `'process'` (intraphase) or `'connection'` (crossphase).
- ☐ If the mechanism goes in `phase.mechanisms`, it defines `get_overrides(self)` with no argument and returns `{}` or a dict. The base class has `get_overrides(self, name)`, so the assignment fails with `TypeError: ... missing 1 required positional argument: 'name'`.
- ☐ Outlet hook: the first argument is the outlet stream's phase, and the rate you return is positive, because the vessel subtracts it.

- ☐ Each solver state that is not an inventory (a distribution, a saturation, a profile) has `limit_negative_inventory=False`. Otherwise the positivity limiter throttles it as if it could run out.
- ☐ Nothing is remembered between calls. The right-hand side is evaluated at trial points the solver throws away. Anything that must persist has to be a solver state.
- ☐ Inputs that may dip slightly below zero are clipped (`max(c, 0.0)`): solvers overshoot.

Controller

- ☐ `super().__init__()` is called.
- ☐ If it controls temperature, `StateKey('global_temp')` is in `self.states` from `__init__` and is written on every `update_state`. `unit.reset()` runs before each solve and calls `controller.reset()`, which clears `self.states`. Alternatively, restore the key in `reset`, as `TemperatureProfileMixin` does.
- ☐ Inlet keys are set on the first `actuate` pass (`resolved_inlets is None`), and outlet keys on the second.
- ☐ An inlet `vol_flow` key names the phase (`phaseref=PhaseRef('liquid', 0)`). A connection-level one is ignored without a warning.
- ☐ Every hard switch (an `if` on time or a state, `max`, `min`) is smoothed or declared as a `StateEvent` whose function changes sign there. Otherwise a stiff solver shrinks its steps to find the corner, and `ScipyBackend` can stop with a "stalled" error. `direction=1` catches only a rising crossing, `-1` only a falling one, and `0` both.

Integrator backend

- ☐ The array `unit.unit_model` returns is copied.
- ☐ Integration starts at `unit.elapsed_time`, not at 0.
- ☐ `retrieve_results` gets `states` with one row per time.
- ☐ What it cannot do is refused: `self.check_algebraic_support(unit)`, and `NotImplementedError` for `eval_sens=True`.

Using any of them

- ☐ Build a new unit for each simulation. `solve_unit` on a solved unit continues from where it stopped.
- ☐ Assign in this order: `Phases`, then kinetics or `intraphase_processes` / `phase_connections`, then `Inlet`.

- ☐ Test against something you already know: an analytic solution, a mass balance, or the same model on a second backend. The three examples here were checked that way, on PharmaPy branch `workshop-crystallizer-fixes` (equivalent to `pharmapy-org 0.3.1`).