Chapter 5 – Just Enough Simulation: State, Time, Solvers, and Co-Simulation

5.0 Before you start

Where we are. Chapter 2 defined three words and taught one of them. A **model** is a description of behaviour that does not run. A **simulation** is one execution of a model over time, from a starting state, producing a trajectory. A **simulator** is the software that performs it. Chapter 4 took the model apart. This chapter takes the other two.

The gap this fills is specific. Chapter 4 left you able to say what a model assumes and where it is valid. It left you unable to say anything at all about what happens when someone presses run – and a startling amount of what goes wrong in a twin happens there, in a component whose input was correct.

What you are assumed to know. Everything Parts I and II so far. Specifically, from Chapter 4: the pot's water-balance model, its parameters `k` and `g`, and the fact that the linear form is the short-interval approximation of an exponential one. From Chapter 3: the simulation runner as a component, and the store's obligation to answer "what did the twin believe at time t?".

The maths budget. Chapter 4 spent the book's one derivative. This chapter spends none. Everything below is a difference equation, a table of numbers, or a ratio. Where a statement is really a theorem, the chapter demonstrates it with arithmetic and says it is not proving it.

The advantage you are bringing. Unlike Chapter 4, this chapter is about territory you already occupy. A step size is a loop increment. A real-time factor is a throughput ratio. Co-simulation is two services with a synchronisation interval and a stale-read problem. Your instincts about clocks, ordering, staleness and non-determinism are not analogies here. They are the subject.

What this chapter covers. A simulation traced by hand; two clocks and the real-time factor; step size, the error-versus-cost trade, and instability as a failure distinct from inaccuracy; initial state, warm start, snapshot and replay; determinism and reproducibility; and co-simulation with its three forced decisions.

What this chapter deliberately does not cover. The taxonomy of solver kinds – Chapter 6, which is why exactly one stepping rule appears here and is labelled as the simplest one. The Functional Mock-up Interface (FMI) and its interface detail – Chapter 6. State estimation – Chapter 6. Monte Carlo – Chapters 6 and 7. Calibration and validation – Chapter 7, and Sec. 5.4 is careful that the error it discusses is a *different quantity* from the gap between model and reality. Discrete-event simulation as a formalism – Chapters 6 and 11.

Learning objectives

By the end of this chapter, you will be able to:

1. **Trace** a simulation run by hand from an initial state, and **distinguish** the trajectory it produces from a record of measurements.

2. **Distinguish** simulated time from wall-clock time, and **compute** a real-time factor from a stated workload to decide whether a run fits inside a request.
3. **Predict** the effect of halving a step size on both numerical error and cost, and **recognise** numerical instability as a different failure from inaccuracy.
4. **Specify** what must be recorded for a simulation run to be reproducible, and **diagnose** why a given run is not.
5. **Explain** why coupling two models is harder than running either alone, and **identify** the three decisions a co-simulation forces on you.

5.1 Why execution is its own subject

The short version: **a correct model can produce a wrong run.**

That sentence is worth sitting with, because it does not fit the mental model most engineers arrive with. We are used to a chain where a correct specification plus a correct implementation gives a correct result. Simulation adds a step in the middle that has its own error, its own failure modes, and its own budget. The model can be perfectly derived, its assumptions all satisfied, its parameters freshly fitted – and the number that comes out can still be wrong, because of how it was computed.

Three concrete forms this takes, all of which appear later in the chapter:

- **The step was too big.** The solver advanced simulated time in chunks too coarse for the behaviour it was tracking, and accumulated error. Sec. 5.4.2 puts numbers on this.
- **The step was so big the answer diverged.** Not "a bit inaccurate" – the output grows without bound and reports a moisture reading of minus three hundred. Sec. 5.4.4.
- **The run cannot be reproduced.** Six weeks later nobody can recreate what the twin computed, which means Chapter 3's audit trail is decorative. Sec. 5.6.

Why this lands on you rather than on the modeller. The modeller chooses the equations. You choose, or inherit, the step size, the horizon, the scheduling, the machine it runs on, and whether the result is cached. Every item on that list changes the answer or its cost. In a twin the pressure is sharper than in offline analysis, because the run has a deadline: the state it is computing is wanted before the next decision. Work on twins for cyber-physical systems makes the deadline explicit: whatever a service spends handling one message has to fit inside the interval its inputs arrive on, or the service falls progressively further behind the thing it is tracking. The same work notes that buying that speed usually costs accuracy, and that the exchange rate is a decision to settle when the system is designed rather than after [1].

That last clause is the chapter in one line: **speed and accuracy are traded against each other at run time, by you, using knobs the modeller handed over.** Knowing which knobs exist is the deliverable.

5.2 A simulation, traced by hand

Nothing here is abstract if you run one. We will run the pot.

5.2.1 The model, restated for execution

Chapter 4 derived two forms. The linear one, valid over short intervals:

```
between waterings:   m falls by k units per hour
at a watering:       m rises by g * (d / 100)
```

Chapter 4 also derived the form the linear one approximates: drying slows as the soil dries, approaching a floor. Written as a rule you could code:

```
the reading falls toward m_floor, and the further above m_floor it is,
the faster it falls
```

We will need that second form from Sec. 5.4 onward, because a model whose answer a solver gets exactly right is useless for demonstrating what solvers get wrong. For now, the linear form is enough.

5.2.2 Three things a run needs before it can start

A run is not "the model". A run is the model **plus three more things**, and a request that omits any of them is under-specified:

1. **An initial state.** `m` at the start. Sec. 5.5.1 argues this deserves more respect than it usually gets.
2. **The inputs over the horizon.** When the doses happen and how large they are.
3. **A horizon and a step size.** How far to go, and in what increments.

Take: `m = 640` at 09:20, `k = 4` units per hour, `g = 48` units per 100 ml, a 120 ml dose at 17:05, horizon to 20:00, step size `h = 1` hour.

5.2.3 The loop

The solver's loop, for the linear form, is three lines:

```
t = start
m = m0
while t < end:
    if a dose lands in [t, t+h):  m = m + g * (dose / 100)
    m = m - k * h
    t = t + h
    record (t, m)
```

Run it. Simulated time is on the left; nothing about the wall clock appears anywhere:

t (simulated)	event	m at end of step
09:20	start	640.0
10:20		636.0
11:20		632.0
...		...

t (simulated)	event	m at end of step
16:20		612.0
17:20	120 ml dose	612.0 + 57.6 - 4.0 = 665.6
18:20		661.6
19:20		657.6
20:20		653.6

That table is a **trajectory**: the sequence of states the run produced.

5.2.4 What a trajectory is, and three things it is not

It is not a measurement. Every number in that column is computed. If a sensor also reported values over the same hours, comparing the two columns is the residual of Chapter 1 – and keeping them in separate columns, with different names, is a discipline worth enforcing in code. A twin that stores predicted and measured values in one table with a flag is a twin that will eventually plot one as the other.

It is not continuous. We have `m` at nine instants. What happened at 17:50 is not in the table. You can interpolate, and interpolating is an assumption. Notice also what the loop did with the dose. The 17:05 dose fell inside the step covering 16:20 to 17:20, and the loop adds it at the *start* of that step – so the run behaved as though the pot were watered at 16:20, forty-five minutes early. Nothing warned us. Adding it at the end of the step instead would have put it fifteen minutes late, and neither choice is more correct than the other at this step size. That is the step size showing through, and Sec. 5.7.3 returns to it when the controller becomes its own model.

It is not the only trajectory. Change the initial state, the inputs, the horizon or the step size and you get a different one, from the same model. Chapter 2 insisted you do not "have a simulation", you *ran* one; this is what that distinction protects. In a twin you will run thousands, and being able to say which run produced a number is the same problem as any other provenance problem you have solved.

5.3 Two clocks

Every simulation has two clocks, and confusing them causes more design errors than any other single thing in this chapter.

> **Simulated time** (`t`) is the clock inside the model. In Sec. 5.2.3 it ran from 09:20 to 20:20.
>
> **Wall-clock time** (`T`) is the clock on the wall. That loop took microseconds.

The nine hours never happened. They were *represented*. Figure 5.1 draws the two clocks against each other, which is the picture to hold when someone says "the simulation takes nine hours".

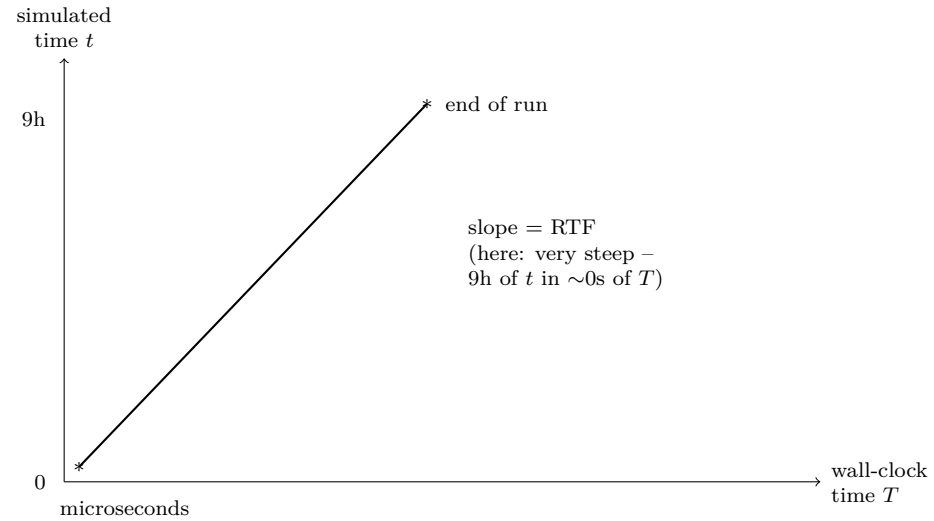

Figure 5.1 The two clocks. A run is a line on this plane, and the real-time factor is its slope – not a property of the model, but of model and machine together.

Say the two clock names out loud in design discussions – "eight hours of simulated time in two seconds of wall clock" – and a whole category of misunderstanding disappears. The slope is also the number that answers Sec. 5.3.4's question about fitting in a request.

5.3.1 Real-time factor

The ratio has a name, the **Real-Time Factor (RTF)**:

```
RTF = simulated time advanced / wall-clock time taken
```

- **RTF = 1** – keeping pace. One second of model per second of reality.
- **RTF > 1** – faster than real time. Nine simulated hours in one wall-clock second is RTF = 32,400.
- **RTF < 1** – slower than real time. Detailed models of complicated physics are routinely here, and it is not a defect; it is a fact you must design around.

Standards for distributed simulation take the diversity of time notions seriously enough to build time management around it, precisely because some participants have no notion of time at all, some must outrun real time, and some are discrete-event simulations with their own scheme [2]. If a standards committee needed to make this explicit, it is not a distinction you can safely leave implicit in your service.

5.3.2 Why a twin wants RTF greater than one

Three of Chapter 1's five value patterns need it, and the reason differs each time.

- ***Predict*** needs to know what happens over the next twelve hours before those twelve hours elapse. RTF must exceed 1 or the forecast arrives after the event.

- ***Decide*** needs many alternative futures compared before a decision is taken. Chapter 1 cited a documented requirement of hundreds of simulations within a few seconds [3]. That is RTF multiplied by the number of scenarios.
- ***Certify-and-train*** wants a year of operation compressed into an afternoon.

Monitor and *diagnose* are the exceptions, and this is a useful check on Chapter 3's component table: diagnosis compares a prediction over one interval that has already happened, so it needs the run to finish before the *next* decision, not before the interval it covers.

5.3.3 When the twin falls behind

A twin's simulation is chasing a physical twin that never pauses. If the model cannot keep up, simulated time falls behind wall-clock time and the gap grows. Work on time discrepancy between digital and physical twins addresses exactly this, including using variable step sizes aimed at reducing the difference between the twin's time and wall-clock time – and reports the honest limit, that during a persistently degraded scenario the delay keeps accumulating, and that the ability to run faster than real time is of no help when there is no new data to consume [4].

Read that limit carefully, because it is the useful part. Running faster than real time lets you catch up **only if the backlog is data you already have**. If the twin is behind because the *network* is slow, extra compute buys nothing. Chapter 2's *age of twin* is the quantity being managed, and Chapter 3's actuation guard is what refuses to act when it grows too large. This section is where those two chapters' abstractions acquire a mechanism.

5.3.4 Does it fit in a request?

Chapter 3 said the simulation runner forces one design question early – whether a run fits inside a request or needs a queue – and that the answer comes from timing a single run. Here is that arithmetic, on the pot.

Case A, the diagnose service. Simulate eight hours at `h` = 15 minutes: 32 steps. At a microsecond a step, 32 microseconds. It fits in a request several thousand times over. No queue. No caching. Do not build one.

Case B, a decide service. Compare 500 candidate watering schedules over 7 days at `h` = 15 minutes:

```
steps per scenario = 7 * 24 * 4 = 672
total steps        = 500 * 672 = 336,000
at 1 microsecond   = 0.34 seconds
```

Still fits, comfortably.

Case C, the same decide service with a serious soil model. Suppose the modelling team replaces the two-parameter model with a spatial one at 40 milliseconds a step:

```
336,000 steps * 0.04 s = 13,440 s = 3.7 hours
```

That does not fit in a request, or in a coffee break. The service now needs a queue, or fewer scenarios, or a coarser step, or the surrogate model of Chapter 4 Sec. 4.5.3 – and

notice that only the last of those is a modelling decision. The other three are yours.

The transferable point. The gap between Case B and Case C is a factor of forty thousand, and it comes entirely from a modelling choice made in a different meeting. This is why Chapter 3 said to time one run early: the architecture on either side of that number is not the same architecture.

5.4 Step size, and what the solver is doing

5.4.1 Why a step size exists at all

The pot's real behaviour is continuous: water leaves it at every instant. A computer cannot do "every instant". So the **solver** advances the state in finite jumps of size `h`, deciding at each jump how much changed.

That is the whole idea. The solver is a loop, `h` is the increment, and everything in this section follows from the fact that the loop is an approximation of something continuous.

The rule we will use is the simplest one that exists: *assume the rate of change stays constant across the step, and multiply.* It has a name and Chapter 6 will give it; here it matters only that it is the simplest, that it is not the one you should use in production, and that everything it demonstrates is true in kind for the better ones.

5.4.2 A concrete demonstration of step-size error

The linear model cannot demonstrate this, because a straight line has a constant rate of change and the solver gets it exactly right at any `h`. That is why Chapter 4 derived the exponential form: the reading falls toward a floor, faster when it is further above it. As a rule:

```
change over a step of h hours  =  -h * (m - m_floor) / tau
```

Take `m_floor` = 400, `tau` = 48 hours, and start at `m` = 640. Simulate 24 hours of drying with no watering, at four different step sizes. The exact answer, from the mathematics rather than the solver, is 545.6.

`h` (hours)	steps	result at t = 24 h	error
24	1	520.0	25.6
12	2	535.0	10.6
6	4	540.7	4.9
3	8	543.2	2.4

Work the first row by hand to see there is no magic in it. One step of 24 hours: the reading is 240 above the floor, so the change is `-24 * 240 / 48 = -120`, giving 520. The solver assumed the reading kept falling at its initial rate for a full day. It did not; it slowed down. So the solver overshot.

Read the two right-hand columns together and the trade appears. Each halving of `h` doubles the work and roughly halves the error. That relationship has a name – this

rule is **first order** – and it is the thing to ask about, because better rules do better: some cut the error by four or sixteen for the same halving. Chapter 6 is a tour of those.

The engineering consequence, stated as a rule of thumb you can use in a review. With a first-order rule, ten times less error costs ten times more compute. If someone proposes tightening accuracy by an order of magnitude, that is an order of magnitude on the bill of Sec. 5.3.4, and it may be cheaper to change the rule than to shrink the step.

5.4.3 Fixed and adaptive stepping

The runs above used a **fixed** step. The alternative is **adaptive**: the solver estimates its own error each step and shrinks `h` where the behaviour is changing fast, growing it where nothing is happening.

You do not set `h` for an adaptive solver. You set a **tolerance** – how much error you will accept – and it chooses. That is usually the better deal, and it has two consequences a software engineer must plan for:

- **The run time is no longer predictable.** A scenario with rapid changes takes more steps. A service with a timeout and an adaptive solver behind it has a tail-latency problem that will not show up in testing on quiet data.
- **Tolerance is a real knob, with limits.** Work sweeping a solver's relative tolerance across ten orders of magnitude reports it as an accuracy-versus-speed knob, and also reports that structuring the run badly – stopping the solver at every sampling instant – both increases the required steps and *reduces the effectiveness of tolerance as a knob* [5].

That second finding is directly actionable and easy to cause by accident: if you force the solver to stop and restart at every ten-minute sample boundary, you have taken away the freedom that made it adaptive. Which brings us to the confusion that causes it.

5.4.4 Instability: the failure that is not "a bit inaccurate"

Everything so far suggests a big step means a poor answer. Past a threshold, something categorically worse happens.

Keep `tau` = 48 hours and take `h` = 120 hours. The step rule multiplies the gap above the floor by `(1 - h/tau)` each step, and `1 - 120/48 = -1.5`. Start 240 above the floor:

```
step 1:  240 * -1.5   = -360      m = 40
step 2:  -360 * -1.5  = 540       m = 940
step 3:  540 * -1.5   = -810      m = -410
step 4:  -810 * -1.5  = 1215      m = 1615
```

The pot is oscillating between soaking and impossibly dry, with the swing growing every step. Figure 5.2 puts that beside the merely-inaccurate case, because the difference is the whole point of the section.

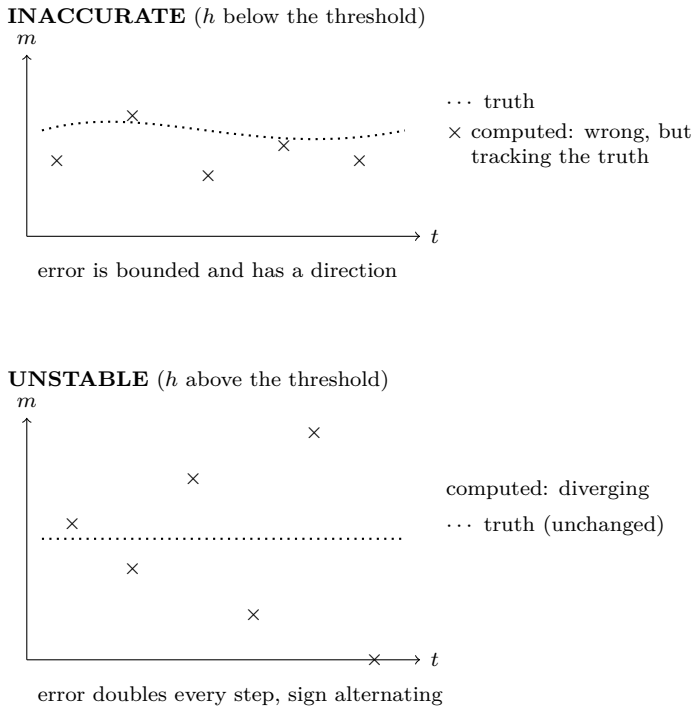

Figure 5.2 Instability is not severe inaccuracy. The left case gives a wrong number; the right gives a meaningless one, and a larger meaningless one every step.

This is **numerical instability**, and the point is that it is not an extreme case of inaccuracy. Inaccuracy gives a wrong number. Instability gives a *meaningless* number, and gives a bigger one every step.

The threshold here is `h = 2 * tau = 96` hours: below it the answers are merely poor, above it they diverge. Every solver has such a threshold and its position depends on the model.

Three things to take from this.

A model has an opinion about your step size, and you cannot discover it by reading the model. This is the strongest argument in the chapter for Chapter 4's Q3 – ask the modelling expert what step size the model is safe at, and treat "whatever you like" as a non-answer.

It is detectable, cheaply. A moisture reading of -410 fails a range check. Chapter 4's runtime envelope check (Sec. 4.7.1) catches divergence for free, which is a second reason to build it.

Some models are much worse than others in this respect. The word for a model that forces tiny steps for stability reasons rather than accuracy ones is **stiff**. That is all you need to know here; if a modeller says a model is stiff, they are telling you your compute bill is about to change, and Chapter 6 explains why.

5.4.5 Step size is not the sampling interval

These are two different numbers and people conflate them constantly, including in requirements documents.

`h`, the step size, is how far the solver advances per iteration. It is chosen for accuracy and stability, and it belongs to the model.

`Ts`, the sampling interval, is how often measurements arrive from the physical twin. It is chosen from the decision the twin supports, and it belongs to the connector.

Chapter 3 derived `Ts` = 10 minutes for the demonstrator, from the watering interval. Nothing about that derivation says anything about `h`. The model might want `h` = 15 minutes for accuracy, or `h` = 1 minute if it were stiffer, or be perfectly happy at `h` = 1 hour.

Why the conflation is expensive. Setting `h = Ts` because both are "ten minutes" forces the solver to stop and restart on every sample, which is precisely the structure reported above as both slower and less responsive to its tolerance setting [5]. Chapter 4's glossary used `h` for the sampling interval; from here on `h` is the step size and `Ts` is the sampling interval, and the two are set by different people for different reasons.

5.5 Starting, stopping, resuming

5.5.1 The initial state is an input, not a detail

Sec. 5.2.2 listed the initial state first, and it is the item most often treated as an implementation detail. It is not: **a run's answer depends on its initial state at least as much as on its model.**

For the pot, the initial state is one number, and getting it is not trivial. The sensor reports a reading, the reading is noisy, and the last one arrived some minutes ago. So what is `m` right now? Answering that is the *state estimator* of Chapter 3, and its mechanism is Chapter 6's. What Chapter 5 contributes is the reason it matters: a run started from a bad state produces a wrong trajectory from a right model, and nothing in the run will say so.

Tutorial treatments of twin simulation put initial state on the same footing as parameters, requiring values for both before an equation can be simulated at all [6]. Treat a missing initial state as a missing argument, not a default.

A design rule that follows. A simulation request should carry its initial state explicitly, not read it from a global. A runner that fetches "the current state" internally cannot be asked to reproduce yesterday's run, which is Sec. 5.6's problem arriving early.

5.5.2 Warm start, snapshot, replay

Three related operations, worth distinguishing because they solve different problems.

Warm start – begin from a saved state instead of a default. Twin platform surveys track *warm startup time* as a measurable property: the time for a twin to resume from a partially initialised or suspended state, where lower is better for fault tolerance and downtime [7]. For a twin this is not an optimisation; a twin that cold-starts has no idea what the pot's state is until it has watched for a while.

Snapshot – save enough state to resume exactly. For the pot, one number. For a large model, it is the whole state vector plus the solver's own internal state, and *the solver's internal state is the part people forget.* An adaptive solver carries a current step size and error history; resume without them and you get a different trajectory from the same snapshot.

Replay – re-run from recorded inputs rather than live ones. This is the single most useful capability in the chapter for a software engineer, because it turns a twin into something testable. The demonstrator's closest published relative supports exactly this: the physical twin can be virtualised by replaying recorded data streams, specifically so results can be shared and reproduced [8].

5.5.3 Simulating the physical twin

Replay generalises into something worth naming. Nothing stops you *simulating the physical twin itself* and pointing the digital twin at the simulation. Chapter 1 met this as the *certify-and-train* value pattern, and Chapter 3's architecture supports it directly: the connector's ingest path does not care whether the readings come from a Raspberry Pi or from a model of one.

This is how you test a twin without a greenhouse, and how you test the Chapter 2 Stage 3 command path without risking a plant. The industrial form of it is hardware-in-the-loop, in which some of the system under test is real hardware and the rest is simulated [9], and manufacturing practice treats semi-physical commissioning as a necessary step when the system's behaviour logic must change with the equipment [10]. Interview studies of continuous integration for cyber-physical systems find teams doing precisely this because hardware in the loop could not be put in the pipeline for safety reasons [11].

Chapter 14 owns this as a practice. It is named here because it is the same machinery: a model, an initial state, inputs, a horizon and a step size.

5.6 Determinism and reproducibility

Chapter 3 required two things of the store that this section makes possible or impossible. It required an answer to "what did the twin believe at time t?", and it required every command to be audited with the state that justified it. Both are worthless if the run cannot be recreated.

> **A simulation run is reproducible if re-executing it with the same recorded inputs yields the same trajectory.**

That is a stronger requirement than most research simulation lives up to, and a twin needs it more, for three reasons Chapters 1 to 3 have already established: an incident review will ask why the twin acted; Chapter 7 will want to compare model versions against the same scenario; and a regression test that is not reproducible is a flaky test attached to a pump.

What has to be recorded. All of it, or you do not have it:

Input	Why it is not optional
Model version	Chapter 3's registry holds it; the run must name it
Parameter set (`k`, `g`, ...)	Refitted parameters give a different trajectory
Initial state, with its own timestamp	Sec. 5.5.1
The full input sequence over the horizon	Doses, and any measured inputs
Horizon and step size, or tolerance	The knobs of Sec. 5.4
Solver identity and its version	Two solvers, same model, different numbers
Random seed, if anything is random	Chapters 6 and 7 make things random
The software environment	Library versions change floating-point results

Where non-determinism comes from, in decreasing order of how often it surprises people.

Unrecorded inputs. Overwhelmingly the most common cause, and not exotic: the run read "now" from a clock, or fetched the latest parameters rather than the ones it was given. Sec. 5.5.1's design rule prevents it.

Floating-point and ordering. Summing the same numbers in a different order gives a different answer in the last bits, and iterating a loop amplifies small differences. Parallel execution reorders summations by default. This is why a run that is "the same except it used four threads" is not the same run.

Library and environment drift. A different version of a numerical library changes results. This is the general reproducibility problem of computational work, and the general answer is to pin the environment rather than hope – containers exist for this and are used for it in scientific practice [12].

Genuine randomness. Only when the model has it, and then a recorded seed is sufficient. Note that a seed makes a run repeatable, not *correct*: Chapter 7 deals with the fact that one sample of a random process is not an answer.

A caution about the word. "Deterministic" here means *the computation repeats*, not *the world is predictable*. A twin can be perfectly reproducible and completely wrong.

5.7 Co-simulation: running two models together

5.7.1 Why you would

So far, one model, one solver. Real twins usually have several, for reasons that are organisational as much as technical.

- **Different parts need different formalisms.** The pot's drying is continuous; the controller's behaviour is a schedule of discrete events. Chapter 4 named that as a hybrid situation and deferred the machinery.

- **Different parts come from different tools and teams.** The soil model from an agronomy group, the pump model from a supplier, the controller from you. Rewriting all three into one program is usually not on offer.
- **Some parts are supplied as black boxes.** A vendor ships something that can be stepped and inspected, but not read.

Co-simulation is executing them together: each keeps its own solver and internal step size, and they exchange values at agreed points.

The interface convention that makes this possible across tools is a standard; you will hear FMI named, and Chapter 6 covers it. What matters here is the shape.

5.7.2 The orchestration loop

Described generically, the loop an **orchestrator** runs has five phases: load the participating units; initialise them, set their parameters, and exchange initial outputs so every unit has values for its inputs; set the start time and the step size; exchange values on the coupled ports; then advance each unit's state by one step – and repeat the last two until the end [13].

If that reads like a distributed system with a synchronisation barrier, that is because it is one. The interval between exchanges is the **macro step**, written `H`, and it is a different number from any unit's internal `h`.

```
for each macro step of length H:
      read outputs from every unit
      write inputs to every unit
      tell every unit to advance by H
```

Figure 5.3 draws the two step sizes together, because the relationship between `H` and each unit's internal `h` is where co-simulation's characteristic bugs live.

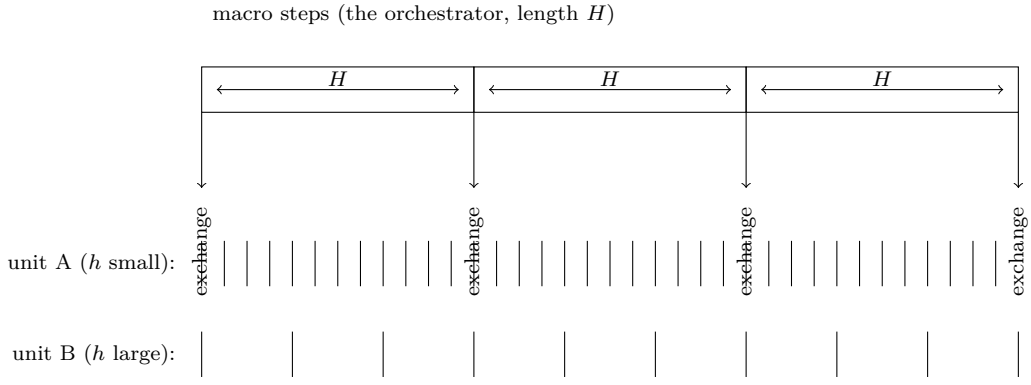

Figure 5.3 Two step sizes, not one. `h` is each unit's own business; `H` is how stale every unit is allowed to let its inputs get.

The blindness between exchanges is the thing to carry forward. Each unit can be individually correct and the coupled system still wrong, because the error lives in the

interval, not in either model – which is why Sec. 5.7.3 treats `H` as a coupling decision rather than a tuning knob.

5.7.3 The three decisions coupling forces

Every co-simulation makes these three choices. Making them by accident is the normal case.

Decision 1: the macro step `H`. Too large and the units act on badly stale inputs; too small and you pay a synchronisation cost every step, on top of each unit's own work. This is not a solved problem, and the honest evidence is that practitioners find it hard: an empirical survey of co-simulation reports that many experts had trouble picking a macro step size, setting tolerances, and dealing with numerical stability, and concludes there is a need for frameworks that ease those choices [14]. When a survey of specialists says the knob is hard to set, a non-specialist setting it silently is a risk worth naming out loud.

Decision 2: the order of execution. Do units read each other's *old* outputs and all advance together, or does one advance first so the next sees its *new* output? Both are legitimate, they give different answers, and the difference does not shrink to nothing as `H` shrinks in every case. In your own terms: this is a read-write ordering question between two services sharing state across a barrier.

Decision 3: what happens to events inside a macro step. Our pump runs for 3.5 seconds. With `H` = 10 minutes, that event is 0.6% of one step. Does the dose land at the step boundary, at its true instant, or smeared across the interval? Sec. 5.2.4 already showed a single-model version of this, where a 17:05 dose was applied at 17:20 without complaint.

The general lesson, and it is one you can carry from your own field. Coupling is harder than the sum of the parts because *the coupling itself has numerical properties.* The survey above notes that these challenges, already present in ordinary simulation, are made worse when co-simulation hides each unit behind an opaque boundary [14] – which is exactly the complaint you would make about debugging across a service boundary you cannot see into. Each unit can be individually correct and the coupled system still wrong.

5.7.4 Worked: the pot and its controller

Make it concrete with the demonstrator.

Unit A, the pot. Continuous. Input: water arriving, in millilitres. Output: moisture reading `m`. Internal step `h` = 15 minutes.

Unit B, the controller. Discrete. Input: `m`. Output: doses. Behaviour: Chapter 1's fixed daily schedule, or – in the Stage 3 twin of Chapter 2 – a rule that waters when `m` falls below a threshold.

Now make the three decisions, with reasons.

Macro step `H`. Start from the decision, as always. In the open-loop case the controller ignores `m` entirely, so `H` can be as large as the interval between scheduled doses. In the closed-loop case the controller reacts to `m`, and `H` must be short enough that it does not act on badly stale moisture. One hour is defensible; ten minutes is safer and costs six times as much synchronisation.

Order. If the pot advances first and the controller then sees the new `m`, the controller reacts within the same macro step. If they advance together from old values, the controller is always one macro step behind. With `H` = 1 hour and a threshold rule, that is an hour of extra delay before watering, which is a real difference in behaviour arising from a decision nobody wrote down.

Events. The 3.5-second pump run inside a 15-minute step. Since the pot model only cares about the total volume delivered – assumption A3 of Chapter 4 said the water mixes fast compared with sampling – delivering the whole dose at the step boundary is defensible, **and it is defensible only because of an assumption in Chapter 4's ledger**. Change the model so that delivery rate matters, and this choice stops being free.

The check that makes the closed-loop case honest. In the open-loop case the coupling is one-way: the controller affects the pot and the pot does not affect the controller. One-way coupling is much better behaved, and if you can arrange for it, do. The threshold controller creates a genuine loop – the pot's moisture changes what the controller does, which changes the pot's moisture – and loops are where `H` and the ordering start to matter. Notice that the *physical* system has the same loop and does not care; it is the *discretisation* of the loop that introduces the problem. That is a co-simulation artefact, not physics, and recognising the difference is the skill this section is teaching.

5.8 Faded example: the offshore turbine, executed

Chapters 3 and 4 built up the turbine service: streaming vibration and temperature, a hybrid model with a physics half accumulating fatigue and a learned half mapping vibration to condition, a work order to a planner.

Worked, for the first two decisions.

Which clock dominates? Fatigue accumulates over years; the decision – send a vessel this week or next month – is made monthly. So the required RTF is modest for the diagnostic, and large for any "what if we run it at reduced power for six months?" question, which needs six simulated months before the meeting ends. The same twin therefore has two very different execution requirements, which is an argument for Chapter 3's simulation runner being a queue rather than a function call.

Step size? Fatigue accumulation is slow and smooth, so a large `h` is tempting. But the loads driving it arrive as storms – short, intense, irregular. A step long enough to be cheap will step over a storm and miss the damage it caused. This is the adaptive-stepping case of Sec. 5.4.3 almost exactly: shrink where things change fast, grow where nothing happens.

Now it is your turn.

- **(a)** The vendor says the model "runs in real time". Write the two questions from Sec. 5.3 that turn that phrase into numbers, and say what answers would make you comfortable.
- **(b)** The service is asked to compare 200 operating strategies over 5 simulated years, at an average of 30 milliseconds per step with `h` = 1 hour. Compute the wall-clock

time. Then state two changes that would bring it under an hour, one of which is not a modelling change.

- **(c)** List what must be recorded for one of those 200 runs to be reproducible six months later, using Sec. 5.6's table. Which row is hardest for the *learned* half of the model, and why?
- **(d)** The physics half and the learned half are separate units in a co-simulation. Make Sec. 5.7.3's three decisions, and say which one you would escalate to the modelling team rather than deciding yourself.

Hint for (b): one of the two changes is in Sec. 5.4, and one is in Chapter 4 Sec. 4.5.3.

5.9 Posed problem: the simulation service specification

You are asked to specify the simulation runner for a twin of a district heating network: 400 buildings, a physics model of the pipe network, a learned demand model per building, and a control service that sets pump speeds every fifteen minutes.

Write the specification. It must state:

1. The two clocks, and the required real-time factor for each of the twin's services – derived from decisions, not asserted.
2. Whether the runner is a function call or a queue, with the arithmetic that decides it and the one measurement you would take first.
3. The reproducibility contract: exactly what a run records, and what your API must therefore accept as arguments rather than fetch for itself.
4. The co-simulation choices of Sec. 5.7.3, including which you are deciding and which you are escalating.
5. One failure mode from Sec. 5.4.4 that your service will detect at runtime, and how.

There is no single right specification. There is a wrong one, and it is the one whose real-time requirement is the phrase "real time".

5.10 Summary

Against the objectives.

1. **A run is a model plus three things:** an initial state, the inputs over a horizon, and a horizon with a step size (Sec. 5.2.2). It produces a trajectory, which is computed rather than measured, sampled rather than continuous, and one of many (Sec. 5.2.4).
2. **Two clocks.** Simulated time and wall-clock time, related by the real-time factor (Sec. 5.3.1). Three of Chapter 1's five value patterns need RTF above 1, and running faster than real time only helps a lagging twin if the backlog is data you already have (Sec. 5.3.3). Whether a run fits in a request is arithmetic, and the answer moved by a factor of forty thousand on one modelling decision (Sec. 5.3.4).
3. **Step size trades error against cost.** With the simplest rule, halving `h` halves the error and doubles the work (Sec. 5.4.2). Past a threshold the failure changes kind: **numerical instability** diverges rather than degrades, and it is cheap to detect

with the envelope check Chapter 4 already asked for (Sec. 5.4.4). Step size `h` and sampling interval `Ts` are different numbers set by different people (Sec. 5.4.5).

4. **Reproducibility is a recorded-inputs problem** (Sec. 5.6), and its commonest cause of failure is a runner that fetches something instead of being handed it. Chapter 3's audit trail and Chapter 7's comparisons both depend on it.
5. **Coupling is harder than the parts.** Co-simulation forces three decisions – macro step, execution order, and event placement inside a step (Sec. 5.7.3) – and specialists report the first of them as genuinely difficult. One-way coupling is much better behaved than a loop, and the loop's problems are an artefact of discretising it, not physics (Sec. 5.7.4).

The sentence to carry forward: **a correct model can produce a wrong run** – and its practical companion, **"real time" is not a requirement; a real-time factor is.**

5.11 Exercises

Objectives in brackets. Solutions and hints follow.

5.11.1 (Objective 1, easy). Using the linear model with `k` = 4 and `g` = 48, trace a run from `m` = 600 at 08:00 to 14:00 with `h` = 2 hours and a 100 ml dose at 09:30. Produce the trajectory table. Then say at what simulated time your run applied the dose, and whether that is a problem.

5.11.2 (Objective 2, easy). A model simulates 30 days of behaviour in 45 seconds. Compute the real-time factor. Then say whether it is fast enough for (a) a daily forecast of the next week, (b) comparing 2,000 scenarios each covering a year, inside a five-minute interactive session.

5.11.3 (Objective 3, easy). Using the exponential rule with `m_floor` = 400 and `tau` = 48 hours, starting at `m` = 640, compute the result at t = 24 hours with `h` = 8 hours. Where does it sit relative to the table in Sec. 5.4.2, and is that what you expected?

5.11.4 (Objective 3, medium). Sec. 5.4.4 found the instability threshold at `h` = 2 * `tau`. Suppose a modeller tells you a different model has `tau` = 20 minutes. What step size must you stay below, and what does that do to the arithmetic of Sec. 5.3.4 Case B if the horizon is still 7 days?

5.11.5 (Objective 4, medium). A colleague reports that re-running yesterday's forecast gives a slightly different answer. List the checks you would make, in the order you would make them, and say why that order.

5.11.6 (Objective 5, medium). In Sec. 5.7.4, the open-loop controller gives one-way coupling and the threshold controller gives a loop. Explain, without mathematics, why the loop makes the macro step size matter more. Then propose a change to the *controller* – not the pot model, and not `H` – that would reduce the sensitivity.

5.11.7 (Objective 2, medium). Chapter 3 deployed the demonstrator's connector on the Raspberry Pi and the rest off-box. Where should the simulation runner go, and what would change your answer? Use Sec. 5.3.4's arithmetic and Chapter 3 Sec. 3.6's three forces.

5.11.8 (Objectives 3 and 4, hard). Complete Sec. 5.8, parts (a) through (d).

5.11.9 (Objective 4, hard). Design the API for Chapter 3's simulation runner such that every run is reproducible by construction. State the request fields, state what the service is forbidden to read from anywhere else, and say how you would test the property – not the endpoint, the property.

5.11.10 (Objectives 1 through 5, hard, open-ended). Do Sec. 5.9, the district heating specification, in full.

Solutions and hints

5.11.1. Steps at 10:00, 12:00, 14:00. The dose lands in the step beginning 08:00 if you test the interval `[08:00, 10:00)`, so at 10:00 `m = 600 + 48 - 8 = 640`; then 632 at 12:00 and 624 at 14:00. The run applied a 09:30 dose as though it happened at 08:00 – 90 minutes early, and up to a full step in the general case. Whether it matters is the right question and the answer is Chapter 1's: the decision is "did a dose land", compared over a whole watering interval, so 90 minutes of placement error does not change the verdict. At `h` = 2 hours it is acceptable; if the decision were "when exactly did watering start", it would not be.

5.11.2. RTF = (30 * 24 * 3600) / 45 = 57,600. (a) Yes, easily: seven simulated days need about 10.5 seconds. (b) 2,000 years of simulated time is about 63 billion seconds; at RTF 57,600 that is about 1.1 million seconds, or thirteen days. No. The gap is three orders of magnitude, so no amount of tuning closes it – this is the surrogate case of Chapter 4 Sec. 4.5.3, or a much smaller scenario set.

5.11.3. Three steps of 8 hours. Each multiplies the gap above the floor by `1 - 8/48 = 0.8333`. So `240 * 0.8333^3 = 240 * 0.5787 = 138.9`, giving `m = 538.9`, error 6.7. It sits between the `h` = 12 row (10.6) and the `h` = 6 row (4.9), as the first-order relationship predicts – roughly proportional to `h`.

5.11.4. Stay below `2 * 20` minutes = 40 minutes, and in practice well below, since the threshold is where divergence begins rather than where accuracy becomes acceptable. Redo Case B: 7 days at `h` = 10 minutes is 1,008 steps per scenario, 500 scenarios is 504,000 steps – about 1.5 times the original, so still fine at a microsecond a step. The interesting part is what happens if the same model also costs 40 milliseconds a step: then the stability constraint and the cost per step multiply together, and this is exactly the situation the word *stiff* names.

5.11.5. *Hint:* order by cheapness and likelihood, which happen to coincide. First: did the run read anything it was not given – a clock, "latest" parameters, the current time (Sec. 5.5.1)? Second: are the model version and parameter set identical (Chapter 3's registry)? Third: is the solver configured identically, including tolerance – and if it is adaptive, did it take the same steps? Fourth: same library versions and same environment [12]? Fifth: threading or parallelism changing summation order? A seed, if any, is worth checking early because it is trivial to check, but it is rarely the cause when nothing in the model is random.

5.11.6. *Hint:* with one-way coupling, a stale input only delays an effect. With a loop, a stale input changes an action, which changes the input, so the error feeds itself. On the second half: a controller with hysteresis – water when below a low threshold, stop

when above a higher one – reacts far less to small errors in `m` than a single-threshold rule, because a single threshold is maximally sensitive exactly at the point where decisions are made. Damping the controller is often cheaper than shrinking `H`, and it is a change you can make.

5.11.7. *Hint:* Sec. 5.3.4 Case A says the run is 32 microseconds, so compute is not the constraint and either placement works. Chapter 3 Sec. 3.6's three forces then decide: latency to the decision is hours, so no pull to the edge; link availability matters for ingest but the runner needs no live link if it is handed its inputs (Sec. 5.5.1); and there is no scale argument at Case A. Off-box, with the rest. What would change the answer: a Case C model, or a closed-loop controller needing a decision faster than a round trip.

5.11.8. *Partial.* (b) `5 * 365 * 24 = 43,800` steps per run; `200 * 43,800 = 8.76` million steps; at 30 ms that is 262,800 seconds, about 73 hours. Two changes that bring it under an hour: adaptive stepping, since the loading is quiet most of the time and only storms need small steps (Sec. 5.4.3); and a surrogate for the expensive half (Chapter 4 Sec. 4.5.3) – the second is the non-modelling change only if the surrogate already exists, so the strictly non-modelling answer is *parallelism across the 200 independent runs*, which is embarrassingly parallel and needs no model change at all. Credit either, and prefer the answer that notices the 200 runs are independent. (c) The hardest row for the learned half is the model version, because a model retrained on a rolling window has a new version every retraining, and if the training data is not itself versioned the model version does not pin the model. (a) and (d) are yours.

5.11.9. *Hint:* the request carries model version, parameter set id, initial state with its timestamp, the full input sequence, horizon, step size or tolerance, solver id, and seed. The service is forbidden to read the wall clock, the "current" state, or the "latest" parameters. Testing the property rather than the endpoint means running the same request twice and asserting trajectory equality, and – better – running it twice in environments that differ in a way that should not matter, such as thread count.

5.11.10. *Hint:* no solution, but a test. Read your real-time requirement back. If it contains the phrase "real time" without a number next to it, you have written a wish. And check item 3 against item 2: a runner that is a queue and a reproducibility contract that assumes synchronous execution will contradict each other.

5.12 Where to go next

In this book. Chapter 6 is the direct sequel and answers the questions this chapter deliberately left open: which stepping rules exist and when to use them, what makes a model stiff, how discrete and continuous behaviour are combined properly, what FMI actually specifies, how state estimation produces the initial state of Sec. 5.5.1, and how Monte Carlo turns one run into an answer about many. Chapter 7 addresses the error this chapter did *not* discuss – the gap between the model and reality – and the fact that a reproducible run can still be wrong. Chapter 11 builds the services whose real-time factors Sec. 5.3.2 derived. Chapter 14 owns replay and simulate-the-physical-twin as a testing practice.

In the literature.

- *Execution and time in twins specifically:* [4] is the best single source on a twin's simulated time falling behind wall-clock time and what variable step sizes can and cannot do about it; [1] (in [15]) on twin service processing time against the gap between messages; [7] for warm startup time and related metrics; [16] for a twin whose simulation is assumed to proceed at the pace of the wall clock, and what is then compared against reality.
- *Solvers, tolerance and cost:* [5] sweeps solver tolerance as an accuracy-versus-speed knob and reports how run structure degrades it; [6] is a tutorial-level primer on simulating the pot-like case, including the requirement for parameter and initial-state values.
- *Co-simulation:* [14] is an empirical survey of practitioners and the source for macro step size and stability being hard in practice; [13] gives the orchestration loop step by step; [17] is a concrete master-algorithm implementation; [18] places co-simulation inside multi-paradigm modelling; [19] couples an architecture description to a simulation engine as a master algorithm.
- *Reproducibility:* [12] on containers for computational reproducibility, which is the practical answer to environment drift; [8] for virtualising a physical twin by replaying recorded streams, in a system closely related to our demonstrator.
- *Simulating the physical twin:* [9] on hardware-in-the-loop with executable twins; [10] on semi-physical commissioning; [11] for teams substituting simulation when hardware could not be put in the pipeline; [20] on testing twins systematically.
- *Consulted, not drawn on above:* [2] for how a distributed-simulation standard handles participants with different notions of time, [21] on consistency checks including step sizes inside a verification process, [22] on simulators as training environments, [23] for hardware-in-the-loop in the twin vocabulary, [24] on augmenting twin models with behaviour, and [25] on checking at runtime that co-simulation master algorithms are being used correctly.

In the demonstrator. Implement Sec. 5.2.3's loop – it is under twenty lines – and run it against two weeks of real watering events pulled from `GET /actuation/{unit}/watering_events`. Then plot the trajectory against the measured readings. You are not calibrating and not validating; you are looking at the two columns of Sec. 5.2.4 side by side, which is the first time in this book that a prediction and a measurement have been in the same picture.

5.13 References

[1] C. Gomes, D. E. L. Rötter, A. Iosifidis, H. Feng, H. Ejersbo, and M. Frasheri, "Sensing and Communication of Data from the Physical Twin," in *The Engineering of Digital Twins*, J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds., Cham: Springer International Publishing, 2024, pp. 147–171. doi: 10.1007/978-3-031-66719-0_7.

[2] "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)–Framework and Rules," *IEEE Std 1516-2025 (Revision of IEEE Std 1516-2010)*, pp. 1–40, May 2025, doi: 10.1109/IEEESTD.2025.11004567.

[3] B. J. Oakes *et al.*, "Case Studies in Digital Twins," in *The Engineering of Digital Twins*, J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds., Cham: Springer International Publishing, 2024, pp. 257–310. doi: 10.1007/978-3-031-66719-0_12.

[4] M. Frasheri *et al.*, "Addressing time discrepancy between digital and physical twins," *Robotics and Autonomous Systems*, vol. 161, p. 104347, Mar. 2023, doi: 10.1016/j.robot.2022.104347.

[5] C. Cimino, F. Terraneo, G. Ferretti, and A. Leva, "Efficient Control Representation in Digital Twins: An Imperative Challenge for Declarative Languages," *IEEE Transactions on Industrial Informatics*, vol. 19, no. 11, pp. 11080–11090, Nov. 2023, doi: 10.1109/TII.2023.3242806.

[6] C. Gomes *et al.*, "Digital Twin Tutorial: The Incubator Case Study," in *Engineering Trustworthy Software Systems: 6th International School, SETSS 2024, Chongqing, China, April 14–21, 2024, Tutorial Lectures*, J. P. Bowen, C. Gomes, and Z. Liu, Eds., Singapore: Springer Nature, 2025, pp. 68–101. doi: 10.1007/978-981-96-4656-2_3.

[7] K. Duran *et al.*, "Toward Digital Twin-as-a-Service (DTaaS) Platforms: A Survey on Architecture, Design Requirements, and Performance Metrics," *IEEE Communications Surveys & Tutorials*, vol. 28, pp. 1845–1878, 2026, doi: 10.1109/COMST.2025.3635582.

[8] E. Kamburjan *et al.*, "GreenhouseDT: An Exemplar for Digital Twins," in *Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, in SEAMS '24. New York, NY, USA: Association for Computing Machinery, Jun. 2024, pp. 175–181. doi: 10.1145/3643915.3644108.

[9] D. Hartmann and H. Van der Auweraer, *The Executable Digital Twin: Merging the digital and the physics worlds*. 2022.

[10] J. Leng, D. Wang, W. Shen, X. Li, Q. Liu, and X. Chen, "Digital twins-based smart manufacturing system design in Industry 4.0: A review," *Journal of Manufacturing Systems*, vol. 60, pp. 119–137, Jul. 2021, doi: 10.1016/j.jmsy.2021.05.011.

[11] F. Zampetti, D. Tamburri, S. Panichella, A. Panichella, G. Canfora, and M. Di Penta, "Continuous Integration and Delivery Practices for Cyber-Physical Systems: An Interview-Based Study," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 3, pp. 73:1–73:44, Apr. 2023, doi: 10.1145/3571854.

[12] D. Moreau, K. Wiebels, and C. Boettiger, "Containers for computational reproducibility," *Nature Reviews Methods Primers*, vol. 3, no. 1, p. 50, 2023, Accessed: Oct. 25, 2024. [Online]. Available: https://www.nature.com/articles/s43586-023-00236-9

[13] G. Abbiati, C. Gomes, M. Sandberg, Z. Kazemi, S. T. Hansen, and P. G. Larsen, "Modelling for Digital Twins," in *The Engineering of Digital Twins*, J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds., Cham: Springer International Publishing, 2024, pp. 89–127. doi: 10.1007/978-3-031-66719-0_5.

[14] G. Schweiger *et al.*, "An empirical survey on co-simulation: Promising standards, challenges and research needs," *Simulation Modelling Practice and Theory*, vol. 95, pp. 148–163, Sep. 2019, doi: 10.1016/j.simpat.2019.05.001.

[15] J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds., *The Engineering of Digital Twins*. Cham: Springer International Publishing, 2024. doi: 10.1007/978-3-031-66719-0.

[16] N. Zhang, R. Bahsoon, N. Tziritas, and G. Theodoropoulos, "Knowledge Equivalence in Digital Twins of Intelligent Systems," *ACM Transactions on Modeling and Computer Simulation*, vol. 34, no. 1, pp. 1–37, Jan. 2024, doi: 10.1145/3635306.

[17] C. Friedrich, A. Lombana, J. Fasquel, C. Schlick, N. Bennani, and M. Mendil, "CoFMPy: A Python Framework for Rapid Prototyping of FMI-based Digital Twins," in *The 2nd International Conference on Engineering Digital Twins*, 2025. Accessed: Nov. 10, 2025. [Online]. Available: https://hal.science/hal-05326255/

[18] P. Carreira, V. Amaral, and H. Vangheluwe, Eds., *Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems*. Springer Nature, 2020. doi: 10.1007/978-3-030-43946-0.

[19] J. Hugues, J. Yankel, J. Hudak, and A. Hristozov, "Twinops: Digital twins meets devops," *CARNEGIE-MELLON UNIV PITTSBURGH PA, Tech. Rep.*, 2022, Accessed: Jan. 08, 2025. [Online]. Available: https://apps.dtic.mil/sti/trecms/pdf/AD1168471.pdf

[20] Y. Ma *et al.*, "Automated and Systematic Digital Twins Testing for Industrial Processes." arXiv, Feb. 2023. doi: 10.48550/arXiv.2302.13198.

[21] R. Honcak and A. Wooley, "An MBSE approach for Virtual Verification & Validation of Systems with Digital Twins," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, Linz Austria: ACM, Sep. 2024, pp. 390–400. doi: 10.1145/3652620.3688252.

[22] X. Liu and I. David, "AI simulation by digital twins: Systematic survey, reference framework, and mapping to a standardized architecture," *Software and Systems Modeling*, Aug. 2025, doi: 10.1007/s10270-025-01306-0.

[23] T. Böttjer *et al.*, "A review of unit level digital twin applications in the manufacturing industry," *CIRP Journal of Manufacturing Science and Technology*, vol. 45, pp. 162–189, Oct. 2023, doi: 10.1016/j.cirpj.2023.06.011.

[24] D. Lehner, S. Sint, M. Eisenberg, and M. Wimmer, "A pattern catalog for augmenting Digital Twin models with behavior," *at - Automatisierungstechnik*, vol. 71, no. 6, pp. 423–443, Jun. 2023, doi: 10.1515/auto-2022-0144.

[25] E. Kamburjan, A. Pferscher, R. Schlatte, R. Sieve, S. L. T. Tarifa, and E. B. Johnsen, "Semantic Reflection and Digital Twins: A Comprehensive Overview," in *The Combined Power of Research, Education, and Dissemination*, vol. 15240, M. Hinchey and B. Steffen, Eds., Cham: Springer Nature Switzerland, 2025, pp. 129–145. doi: 10.1007/978-3-031-73887-6_11.