Advanced Event and Process View Modeling

Quiz Study Guide — Chapter 7

Manuel D. Rossetti, PhD P.E.

How to Use This Deck

This deck is a **checklist**, not a summary. Every slide is phrased as something you should be able to *state*, *derive*, or *compute* without looking it up.

- **Question formats:** multiple choice, true/false, matching, fill-in-the-blank, multi-slot fill-in-the-blank.
- The **Common Traps** slides are built from the actual wrong answers — read them last.

Theme

Everything here is a *constraint* the basic process view cannot express on its own: finite buffers, shared pools, time-varying arrivals, time-varying capacity, impatience, and inventory.

What This Chapter Covers

Constraints

- Blocking and overlapping seize/release
- Resource pools and their rules
- Non-stationary arrivals (NHPP)
- Capacity schedules

Measurement and models

- `ResponseInterval`, `ResponseSchedule`, `TimeSeriesResponse`
- Balking and reneging
- The (r, Q) inventory model
- Verification and validation

Objectives

You should be able to:

- Explain why blocking requires **overlapping** seize and release, and write the pattern
- Distinguish a resource **selection** rule from an **allocation** rule, and name the defaults
- Name the two methods of generating an NHPP and how thinning picks λ^*
- State the two capacity-change rules and what each does to a busy resource
- Write the definitions of $IU(t)$ and $\overline{SU}$ and say when they differ
- Define inventory position and the (r, Q) ordering logic
- Distinguish verification from validation

Blocking

A customer must **overlap** the seize and release of resources: it must not give up the **upstream** resource until it has acquired a unit of the **downstream** buffer, since the limited downstream waiting space constrains flow.

```
val a1 = seize(worker1); delay(st1)
val b  = seize(buffer);  release(a1)
val a2 = seize(worker2); release(b)
delay(st2); release(a2)
```

The chair (waiting space) at station 2 is modeled as a `ResourceWithQ` **of capacity 1** — not a counter, hold queue, or blocking queue.

`release(worker1)` — the resource overload — is valid because the KSL releases **all of the entity's allocations of that named resource**, which is correct when the entity holds exactly one.

Resource Pools

A `ResourcePool` generalizes `Resource` by combining individual `Resource` instances into a larger pool, allocated via **selection** and **allocation** rules.

A **selection** rule decides *which resources are eligible*; an **allocation** rule decides *how the units are taken from them*. In `ResourcePoolExample`, four resources form two `ResourcePoolWithQ` instances, with `george` in **both**.

Selection and Allocation Rules

Rule	Behavior
`ResourceSelectionRuleIfc`	`selectResources(amountNeeded, list)` returns a list that **together** has enough units; an **empty list** means the entity must wait
`FirstFullyAvailableResource`	The first resource that can **by itself** fully meet the request; a 0- or 1-element list
`AllocateInOrderListedRule`	**The default**: allocate from each resource in the order listed until filled
`ResourceAllocationRule`	Constructed with a comparator — `LeastUtilized`, `LeastSeized`, `MostAvailable` variants
`RandomAllocationRule`	Randomly permute the candidates, then allocate in that order

Data-Driven Sequences

In the Test and Repair Shop, a part's test plan is a Kotlin `List<TestPlanStep>`, where `TestPlanStep` bundles a `ResourceWithQ` with a `RandomIfc` for the processing time.

- Plans are assigned by a `REmpiricalList<List<TestPlanStep>>` according to a discrete empirical distribution.
- The process consumes the plan with a Kotlin `iterator()` and a `while (itr.hasNext())` loop calling `use(tp.resource, delayDuration = tp.processTime)`.

Non-Stationary Arrivals

The two main methods of generating a **non-homogeneous Poisson process** are **thinning** and **rate inversion**. In thinning, the constant rate λ^* used to generate candidate event times is the **maximum** of $\lambda(t)$.

> ⚠ Trap
>
> Naively switching exponential distributions per time interval does **not** produce a proper NHPP.

The NHPP Classes

Class	Role
`RateFunctionIfc`	Interface for a time-varying mean rate $\lambda(t)$
`PiecewiseConstantRateFunction`	Rate constant on each of a sequence of durations
`PiecewiseLinearRateFunction`	Rate changes linearly between specified values
`NHPPEventGenerator`	Fires arrivals according to a rate function, typically via thinning
`NHPPTimeBtwEventRV`	Returns inter-event times for an NHPP

Capacity Schedules

A `CapacitySchedule` is a sequence of **(capacity, duration)** items attached via `resource.useSchedule(schedule, ...)`, varying $c(t)$ over time.

The `CapacityChangeRule` enum has two values:

- `IGNORE` — the change starts immediately (the "inactive sign" goes up on time), but busy units **finish the in-progress entity**; they become inactive only after release, possibly **cutting the scheduled duration short**.
- `WAIT` — the change **waits** for the busy resource to finish, then runs for its **full duration**, shifted in time.

Under `IGNORE`, available units $a(t)$ can go **negative**.

Related: `RequestQueueNotificationRuleIfc` controls the order in which registered request queues are notified when capacity is gained; `registerCapacityChangeQueue` registers an external queue.

Utilization Under a Schedule

Three states are defined by $b(t)$ and $c(t)$; the **Inactive** state is $b(t) = 0$ **and** $c(t) = 0$.

$$IU(t) = \begin{cases} 0 & c(t) = 0 \\ 1 & b(t) \geq c(t) \\ b(t)/c(t) & \text{otherwise} \end{cases} \qquad \overline{SU} = \frac{\int b(t)\,dt}{\int c(t)\,dt}$$

- $a(t) = c(t) - b(t)$ is the number available.
- When $c(t)$ is **constant**, $\overline{IU}$ and $\overline{SU}$ are **equal**.
- When $c(t)$ **varies**, $\overline{SU}$ **can exceed 1** — the chapter's "professor's utilization at 106%" under the `IGNORE` rule.

Statistics by Time Period

Class	Role
`ResponseInterval`	One **(start, duration)** interval; averages for responses, totals for counters, recorded once per replication
`ResponseSchedule`	A **cycle of intervals**; `addResponseToAllIntervals(...)` attaches one response to every interval
`TimeSeriesResponse`	A fixed number of equal-length periods, recording per-period averages or totals per replication

Reading a Schedule Call

`hourlyResponseSchedule.addIntervals(0.0, 6, 60.0)` adds **six contiguous intervals of length 60.0 each**, starting at time 0.0 relative to the schedule — so six hourly buckets. `addResponseToAllIntervals(response)` then attaches one response to every interval at once.

> ⚠ Trap
>
> `TimeSeriesResponse` does **not** react to the warm-up event — data collected before warm-up is still recorded — and its output is **not** shown in the console. Access it programmatically or through the KSL database.

Balking and Reneging

In the walk-in clinic model:

- The doctor queue is `RequestQ(this, "DoctorQ", discipline = Queue.Discipline.RANKED)`, supplied to the `ResourceWithQ` via `queue = doctorQ`.
- Priorities come from a `Map<RandomVariable, Int>` keyed by the chosen service-time RV; the entity's `priority` is set in its init block from that map.
- **Balking** is an early `return@process` when `priority == 3 && doctorQ.size >= balkCriteria`.
- **Reneging** schedules an event after the renege time; when it fires the queue is searched and, if the patient is still waiting, `doctorQ.removeAndTerminate(request)` removes the request **and terminates the suspended process** of the associated entity.
- Only the **low-priority** patients renege — not all three priorities.

The (r, Q) Inventory Model

$$IP(t) = I(t) + IO(t) - BO(t)$$

on-hand plus on-order minus back-ordered. An order is placed when **inventory position** — not on-hand inventory — falls to or below r.

The amount ordered is the **smallest integer multiple of** Q sufficient to raise $IP(t)$ **above** r: $n\,Q$ with $n = \lceil (r - IP(t))/Q \rceil$ when below r, and exactly Q when $IP(t) = r$.

Replenishment. `Inventory` calls its registered `InventoryFillerIfc` (e.g. a `Warehouse` inner class), which schedules an `endLeadTime` event — attaching the demand amount as the event's **message** — whose action calls `inventory.replenishmentArrival(amount)`.

`Inventory` itself implements `InventoryFillerIfc`, so one inventory can supply another and supply chains can be built by composition.

Verification and Validation

- **Verification** — checking that the model **correctly produces the output it is designed to produce**. Software quality control for the model.
- **Validation** — building **credibility** that the model adequately represents the system **for its intended purpose**, often by comparing model output to real-system output or to analytical approximations.

Validation is **not** possible without verification.

Techniques: set all resource capacities to **infinite** so there is no queueing, and check that expected total time in system equals the **sum of the mean service times**; cross-check $\rho = \lambda/(c\mu)$ per station; trace a single entity through the model; develop the model in stages.

Whitt's Approximation

$$W_q(GI/G/c) \approx \frac{c_a^2 + c_s^2}{2} W_q(M/M/c)$$

where c_a^2 is the arrival squared coefficient of variation and c_s^2 the service SCV.

When the arrival process is **Poisson**, $c_a^2 = 1$, so the multiplier reduces to

$$\frac{1 + c_s^2}{2}$$

Note the **sum** over 2 — not the product.

Drills

This chapter's question bank has no numeric items — these are practice problems built from the formulas and rules it tests in other formats. Expect the quiz to test the same relationships, not these exact numbers.

1. A resource has $b(t) = 3$ busy units and capacity $c(t) = 2$ at time t. What is $IU(t)$? What is $a(t)$?
2. An (r, Q) policy has $r = 10$, $Q = 5$, and $IP(t) = -2$. How much is ordered?
3. A station is fed by a Poisson arrival process and has service SCV $c_s^2 = 3$. By what factor does Whitt's approximation scale $W_q(M/M/c)$?
4. `addIntervals(0.0, 8, 30.0)` — how many intervals, of what length, covering what total span?
5. Under the `IGNORE` rule, a capacity decrease of 1 is scheduled at $t = 100$ for a duration of 20 while the single unit is busy until $t = 105$. When does the unit become inactive, and how long does the change effectively last?

Common Traps

- The upstream resource is released **after** seizing the downstream buffer, never before.
- An **empty** list from a selection rule means "wait", not "allocate nothing".
- The default pool allocation rule is `AllocateInOrderListedRule`.
- Thinning uses the **maximum** of $\lambda(t)$, not the average.
- Switching exponential distributions per interval does **not** give a proper NHPP.
- $\overline{SU}$ **can exceed 1** under `IGNORE`; $\overline{IU}$ needs care too.
- `TimeSeriesResponse` **ignores** the warm-up event.
- The reorder trigger is **inventory position**, not on-hand inventory.
- Whitt's multiplier is $(c_a^2 + c_s^2)/2$ — a **sum**.

Appendix — Drill Answers

1. $b(t) = 3 \geq c(t) = 2$, so $IU(t) = \mathbf{1}$. And $a(t) = c(t) - b(t) = 2 - 3 = \mathbf{-1}$ — negative, which is exactly what the `IGNORE` rule permits.
2. $n = \lceil (10 - (-2))/5 \rceil = \lceil 2.4 \rceil = 3$, so order $3 \times 5 = \mathbf{15}$, raising IP to 13, above r.
3. Poisson arrivals give $c_a^2 = 1$, so the factor is $(1 + 3)/2 = \mathbf{2}$.
4. **Eight** intervals of length **30.0** each, spanning **240** time units from the schedule's start.
5. The unit continues serving until $t = 105$ and becomes inactive **then**. The change was scheduled to end at $t = 120$, so it effectively lasts only **15** time units — the duration is **cut short**.