17-614: Project Write-up: Modeling *Presto* (Ride Sharing App)

Team #12 | Iris Huang, Viren Dodia, Ziqin Shen, Ray Xue

October 5, 2025

1 Task 1: Structural Modeling

1.1 Object Model Diagram

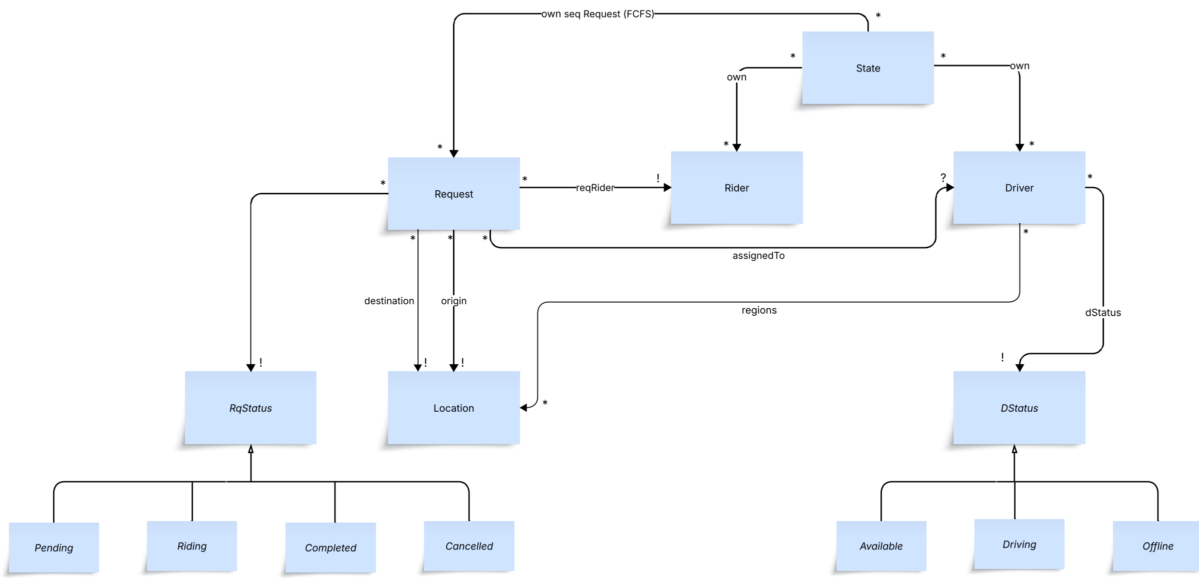

Figure 1: Object model of the *Presto* system.

1.2 Invariants Discovered During Modeling

During Alloy modeling of *Presto*, we identified several invariants that ensure consistency of the system state:

- **One active request per rider**: Each rider can have at most one request that is either `Pending` or `Riding`.
- **Driver exclusivity**: Each driver can serve at most one request at a time. A driver is in the `Driving` state if and only if they are assigned to exactly one active request.
- **Assignment consistency**: A request in `Riding` must have exactly one assigned driver. Requests in `Pending`, `Completed`, or `Cancelled` must have no assigned driver.
- **Queue well-formedness**: The set of requests in the `Pending` state must exactly equal the elements of the `pendingQ` sequence (with no duplicates).
- **Origin-destination sanity**: For realism, each request must have distinct origin and destination.

These invariants capture both explicit rules in the specification and implicit requirements necessary to preserve logical correctness.

1.3 Model Validation Strategy

Our validation strategy for the Alloy model involved:

- **Operation preservation checks**: We wrote assertions ensuring that the four core operations (`request`, `cancel`, `match`, and `complete`) preserve the system invariants across state transitions.
- **Visualization of states**: Using `run` commands, we generated concrete states, such as an empty system, one pending request, and one riding request. These helped confirm intuitive behavior.
- **Counterexample analysis**: If Alloy produced a counterexample, we refined the model (e.g., corrected scoping errors or missing conditions) until the invariants held.
- **Cross-checking with the spec**: Each invariant and operation was traced back to requirements in the Presto specification to ensure coverage.

In addition, we developed a suite of **positive and negative test predicates** to exercise the model further. Positive tests (e.g., `test_MultiplePending`, `test_MultipleConcurrentRides`) confirmed that realistic states were satisfiable, while negative tests (e.g., `test_RidingRequestInPendingQueue`, `test_AvailableDriverIsAssigned`, `test_DuplicateRequestInQueue`) demonstrated that invalid states were impossible under our invariants. Running these tests under scopes of 4–7 atoms with exactly one `State` reinforced our confidence that the model was both correct and complete.

1.4 Scopes for Checking Assertions

For invariant preservation checks, we used scopes of 6–7 objects with exactly one State and up to 6 sequence elements. This bound was sufficient because:

- It allowed us to explore diverse combinations of Riders, Drivers, and Requests while keeping the analysis tractable.
- Larger scopes (beyond 7) did not uncover new counterexamples, suggesting our invariants are robust.
- These bounds match the project's guidance for balancing coverage with solver performance.

2 Task 2: Concurrency with FSP/LTSA

2.1 Process Structure Diagram

- **ORDER**: a generator that cycles order identifiers $t \in \{1, \ldots, \texttt{NUMRIDER}\}$ and emits `request[`t`]` in round-robin order.
- **RIDER(i)** for each $i \in \{1, \ldots, \texttt{NUMRIDER}\}$: `request[`$i$`]` $\to$ `match[`i`]` $\to$ `complete[`i`]` (or `cancel[`i`]` before match).
- **SCHEDULER** as a driver pool `DRIVER(`d, t`)`:
  - if $d > 0$: `match[`t`]` uses one available driver $\Rightarrow$ `DRIVER(`$d{-}1$`, next(`t`))`
  - `cancel[`t`]` skips the slot $\Rightarrow$ `DRIVER(`d`, next(`t`))`
  - if $d <$ `NUMDRIVER`: `complete[`t`]` returns a driver $\Rightarrow$ `DRIVER(`$d{+}1$`, `t`)`

The system composition is:

$$\texttt{ASSIGN\_SYS} = (\| \ i : \texttt{Order} @ \texttt{RIDER}(i)) \ \| \ \texttt{SCHEDULER} \ \| \ \texttt{ORDER}.$$

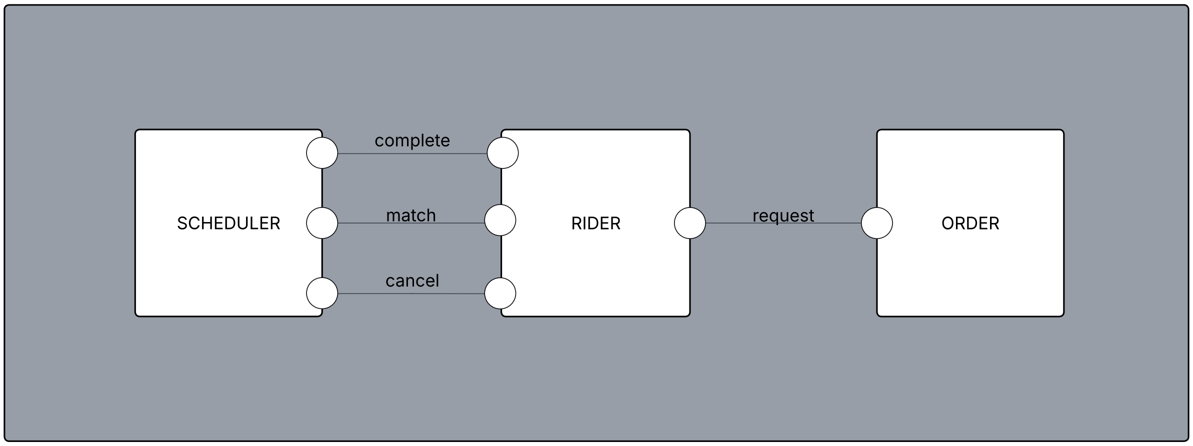

Figure 2: Process structure of the *Presto* system.

2.2 Protocol Design

Events. We model four observable events with an index t identifying the request/rider: `request[`t`]`, `match[`t`]`, `cancel[`t`]`, `complete[`t`]`.

Rider protocol. For each rider i, the process `RIDER(`i`)` performs `request[`i`]` then waits either to be matched (`match[`i`]`) and subsequently completed (`complete[`i`]`), or to cancel before being matched (`cancel[`i`]`), after which it may request again.

Scheduler / driver pool. The scheduler maintains an integer d of available drivers and a rotating pointer t. If $d > 0$, the scheduler may synchronize on `match[`t`]` (consuming one driver and moving to next(t)); upon `cancel[`t`]` it skips to next(t) without consuming a driver; upon `complete[`t`]`, it returns a driver (d+1) while keeping t to allow back-to-back completions.

Arrival order (FCFS). Arrival order is modeled by `ORDER` which emits `request[`t`]` in a round-robin over the index set use a ticketing scheme to make sure the riders are served by requesting order, giving each user request a number to manage request order. We enforce first-come, first-served with a property automaton `FIFO_CHECK` that flags an `ERROR` if `match[2]` occurs before `match[1]` when 1 arrived first (and symmetrically for two requests). This is the standard FIFO witness for two requests and is sufficient to catch out-of-order matches.

2.3 Details Abstracted from Task 1

Compared to the Alloy state model, the FSP protocol abstracts away:

- **Locations and regions**: We do not carry `origin`, `destination`, or driver `regions`. Region feasibility is left implicit (always matchable when a driver is available).
- **Structural associations**: Fields such as `reqRider`, `assignedTo`, and the explicit `pendingQ` structure are represented behaviorally via indexed events and the scheduler's pointer t.
- **Per-request object identity**: Requests are denoted by indices (t) rather than explicit objects; this is sufficient for the concurrency protocol.

2.4 Details Added Beyond Task 1

The FSP model adds explicit behavioral constraints that Alloy does not natively express:

- **Interleaving semantics**: Competing `request`, `match`, `cancel`, `complete` events from multiple riders interleave under LTSA's process-algebra rules.
- **Driver capacity dynamics**: The integer parameter d captures resource consumption at `match[`t`]` and release at `complete[`t`]`.
- **Safety property for FCFS**: The `FIFO_CHECK` property process rules out any trace where a later request is matched before an earlier one (two-request witness).
- **Progress (liveness)**: We require that completions keep happening via the LTSA progress set `progress SERVED = {complete[Order]}`; optionally, we can strengthen liveness with a per-request watchdog ensuring that after `request[`i`]` some `match[`i`]` or `cancel[`i`]` must occur.

3 Task 3: Reflection

3.1 Alloy: Strengths and Weaknesses

Strengths:

- Naturally suited for modeling structural constraints and invariants.
- Immediate visualization of counterexamples, which aids debugging.
- Compact and expressive syntax for relational properties.

Weaknesses:

- Not designed to capture concurrency or event ordering explicitly.
- Large scopes can cause performance issues.
- Expressing temporal behaviors (e.g., eventuality) requires workarounds.

3.2 FSP/LTSA: Strengths and Weaknesses

Strengths.

- Natural description of concurrent event interleavings and shared synchronizations.
- Built-in checks for deadlock and support for progress (liveness) via progress sets.
- Property processes (e.g., `FIFO_CHECK`) make safety conditions executable.

Weaknesses.

- Structural constraints (e.g., object associations, multiplicities) are not explicit; they must be encoded behaviorally or left to Alloy.
- FIFO beyond the two-request witness requires either a generalized property pattern or a larger monitor, which can grow in complexity.
- Liveness per request ("each request eventually served") often needs additional fairness assumptions or watchdog processes to exclude pathological schedules.

3.3 Other Aspects of Ride Sharing

While our models capture the core protocol, real-world ride sharing involves additional aspects:

- **Pricing and payment**: Fare calculation, dynamic pricing, and payment handling.
- **Trust and reputation**: Ratings, cancellation penalties, and fraud prevention.
- **Geographic constraints**: Real-world routing, travel times, and multi-region rides.
- **System resilience**: Handling driver disconnections, rider no-shows, or sudden surges.

Additional information : We used AI(ChatGPT 5) on Alloy, FSP modeling and latex formating