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

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

October 3, 2025

1 Task 1: Structural Modeling

1.1 Object Model Diagram

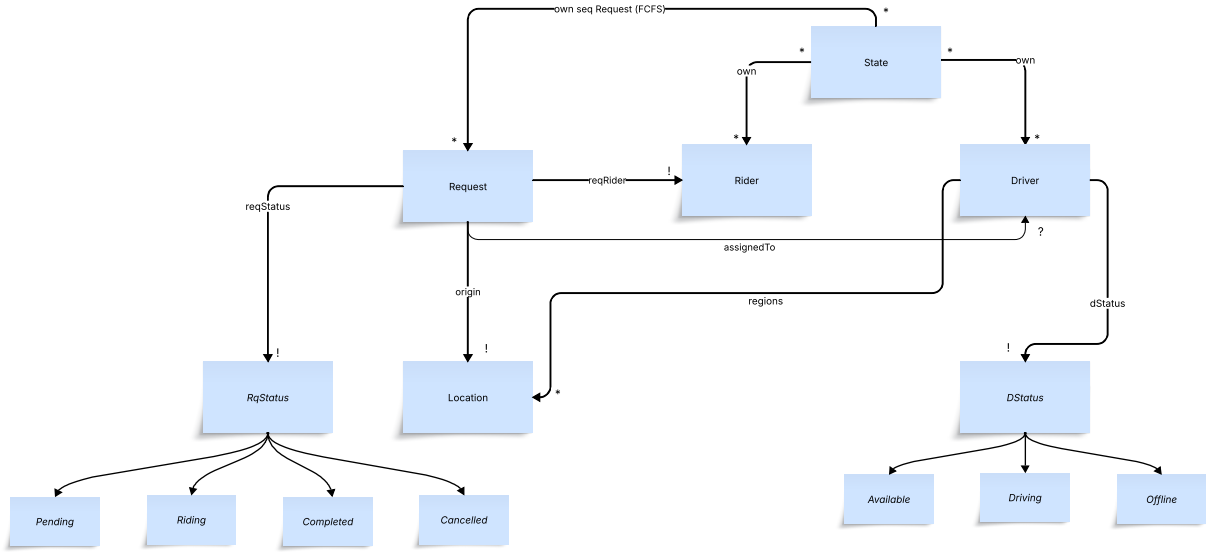

Figure 1: Object model of the *Presto* system.

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.

1.4 Scopes for Checking Assertions

To check assertions, we limited the search space using small but sufficient scopes:

- Typical runs used up to 6 riders, 6 drivers, 6 requests, and sequence lengths up to 6.
- We used `exactly 1 State` to enforce a single system snapshot for each check.
- These small scopes were sufficient to uncover design errors while keeping analysis tractable.

2 Task 2: Concurrency with FSP/LTSA

2.1 Process Structure Diagram

2.2 Protocol Design

2.3 Details Abstracted from Task 1

2.4 Details Added Beyond Task 1

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.

Figure 2: Process Structure diagram for the FSP model.

3.2 FSP/LTSA: Strengths and Weaknesses

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.