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

Team #7 | Andrew Chang, Divyansh (DS) Sood, Ricky Cui, Yoonmee Hwang

October 5, 2025

1 Task 1: Structural Modeling

1.1 Object Model Diagram

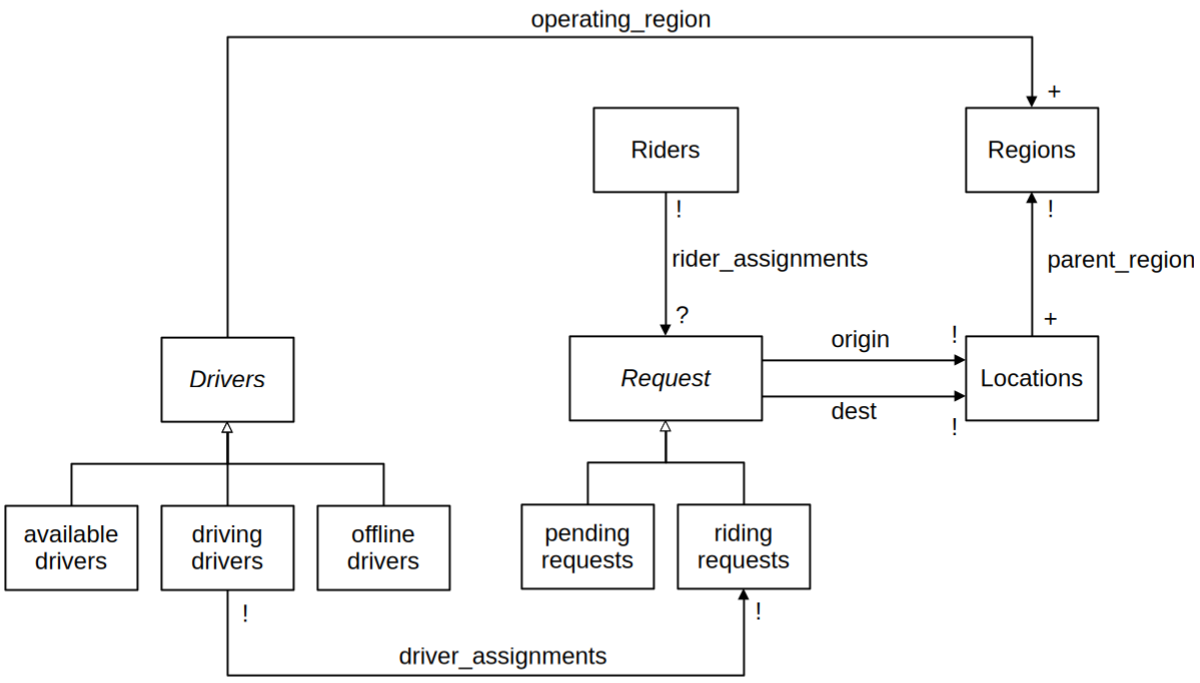

Figure 1: Object model of the *Presto* system.

1.2 Invariants Discovered During Modeling

When modeling Presto instances, we had to make sure that all drivers were referenced by every Presto instance. This is because all Presto instances represent the same app at different times. The specification never mentions whether the drivers can be removed or added, so it was assumed that the number of drivers always remains the same.

```
all p: Presto, d: Driver |
    d in p.available_drivers +
    p.offline_drivers +
    p.driving_drivers
```

In addition, we had to make sure that there were no dangling requests. All requests should be referenced by at least one Presto instance. A dangling request is essentially equivalent to the request not existing at all since no Presto instance knows about it.

```
all req: Request | some p: Presto |
    req in p.pending_requests + p.riding_requests
```

1.3 Model Validation Strategy

We validated the model using a suite of positive and negative tests to ensure all invariants and operational logic were correct.

We created the following positive tests to validate our basic logic. We created predicates for request, match, cancel, and complete to confirm that states in which we can perform these operations are reachable in the first place. Furthermore, predicates for matching a request to a driver within and outside of the assigned regions confirm that the complex logic for the matching operation is correct.

The negative tests prove that key invariants cannot be broken. We have negative tests to prove a rider cannot make a second request while one is active, a ride in progress cannot be canceled, only available drivers are matched, and the system correctly prioritizes local drivers instead of out-of-region drivers when possible. Together, these tests add another layer of confirmation that our invariants are complete and our model isn't over or under specified.

1.4 Scopes for Checking Assertions

For checking the invariant preservation assertions, we used a scope of 6. We also found that the minimum scope required to reasonably test the invariants is 3. This allows for 2 Presto states (for the before/after transition) and 1 extra potentially confounding state. This is enough to find basic flaws.

However, a scope of 6, as used in the checks, is a stronger choice. It is justified because it allows for much more complex scenarios that a minimal scope cannot. For example, it can model multiple pending requests, multiple available drivers, and multiple riders competing for resources simultaneously. According to the small scope hypothesis, if a logical flaw exists, it is likely to be found in a small instance. A scope of 6 is small enough to be analyzed quickly while being large enough to create topologically interesting scenarios that could expose subtle bugs. While a smaller scope would also likely be sufficient, using 6 provides a higher level of assurance that the invariants hold under more complex conditions.

2 Task 2: Concurrency with FSP/LTSA

2.1 Process Structure Diagram

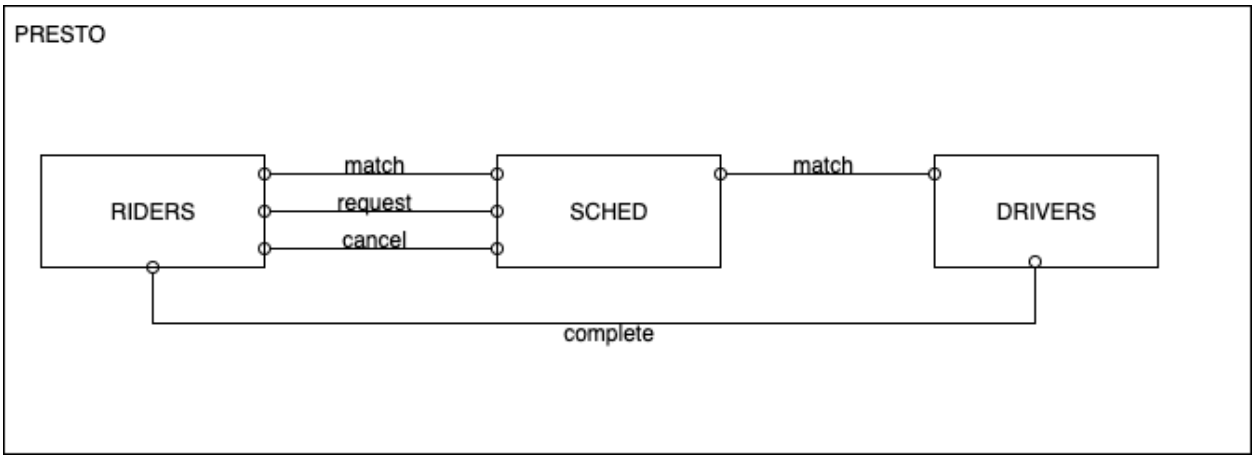

Figure 2: Process Structure diagram for the FSP model.

2.2 Protocol Design

The protocol ensures first-come, first-served (FCFS) order by using the central SCHED(scheduler) process, which acts as a queue that explicitly tracks the arrival order of requests. The Scheduler is a State-Based Queue.

- The SCHED process uses its own states to represent the contents of the request queue: Empty Queue: The scheduler starts in an empty state (SCHED).
- One Request: When the first rider (e.g., rider x) makes a request, the scheduler transitions to a state that remembers this rider is first in line (SCHED[x]).
- Full Queue & FCFS Enforcement: When a second rider (y) makes a request, the scheduler moves to a state that remembers the full order: SCHED[x][y]. In this state, the protocol enforces the FCFS rule by restricting the system's behavior: the only match action allowed is match[x]. The system is structurally prevented from matching rider y until rider x has been served or has cancelled their request.

By encoding the queue's order into its states and only allowing the head of the queue to be served, the model guarantees that requests are handled in the order they were received.

2.3 Details Abstracted from Task 1

To effectively model and analyze the FCFS concurrency protocol, several details from the structural Alloy model (Task 1) were intentionally abstracted away. These details are not relevant to the temporal ordering of events, which is the central concern of this task. Including them would needlessly complicate the state space for analysis.

- Geographic Information: All concepts related to geography were removed. This includes the Region and Location signatures, the origin and destination of a request, and the driver's operatesIn preferences. The FCFS protocol is concerned only with the order of requests, not their spatial properties.
- Complex Matching Logic: The specific rule from the Alloy model where a driver could be matched outside their preferred region was abstracted. In the FSP model, any available driver can be matched with the rider at the head of the queue, simplifying the focus to the protocol's ordering property.
- Driver States: The offline driver state was abstracted away, as offline drivers do not participate in the matching protocol. All drivers in the FSP model are implicitly available to participate in a match.

2.4 Details Added Beyond Task 1

To properly capture the system's concurrent behavior, the FSP model introduced new dynamic and behavioral details that were not present in the static Alloy model.

- Explicit Queue Mechanism: The most significant addition was the SCHED process, which explicitly models a request queue with a fixed capacity. The FSP model uses indexed states (e.g., SCHED[first][second]) to represent the dynamic, ordered contents of the queue, a concept not explicitly modeled in the structural Alloy specification.

- Rider Cancellation: The cancel action was explicitly modeled as a choice available to a rider after making a request. This introduces more complex state transitions where the queue order can change dynamically, which is a crucial aspect of a real-world concurrent system.
- Explicit Error States: The FSP model defines explicit ERROR states for protocol violations, such as a rider making a duplicate request or a new request arriving when the queue is full. This concept of a protocol-defined error state is a feature of behavioral modeling that is not typically part of a structural Alloy model.

3 Task 3: Reflection

3.1 Alloy: Strengths and Weaknesses

Strengths:

1. Alloy's declarative syntax is a good fit for modeling large and complicated systems.
2. The ability to automatically check that all operations preserve the system's invariants is one of Alloy's key strengths. It exhaustively searches for counterexamples, providing a strong guarantee within a specified scope that operations cannot corrupt the state.

Weaknesses:

1. Atoms cannot change their type. When representing drivers changing state from available to driving, it is easier to have different fields in a Presto instance and move the drivers between them, rather than trying to make drivers change their type. Doing the latter would require multiple atoms representing the same driver, each of which represents a different driver state, which becomes cumbersome very quickly.
2. When using functions (i.e. `f: a -> b`), it is not immediately obvious how to get the domain or range of the function. Apparently, this has been improved in Alloy 6.

3.2 FSP/LTSA: Strengths and Weaknesses

Strengths:

1. Explicit Concurrency Modeling: FSP's primary strength is its ability to explicitly model how concurrent processes interact using parallel composition (||) and shared actions. This provides an intuitive way to build a complex system from simpler components.
2. Automated Verification of Concurrent Properties: The LTSA tool can automatically analyze the model for critical properties that are difficult to reason about manually.
   (a) Deadlock Analysis: It automatically detects if the system can enter a state where it completely stops.
   (b) Safety Properties: It can verify that "something bad never happens," such as a violation of event ordering (like FCFS), using property processes.
   (c) Liveness Properties: It can check that "something good eventually happens" (e.g., a request is eventually served) using progress checks.

Weaknesses:

1. Limitations in Data Modeling: FSP is focused on event sequences and is not well-suited for modeling complex data states or data-dependent constraints. This was evident in Task 2, where geographic details like *Regions* had to be abstracted away.

2. State Space Explosion: Like other model checkers, LTSA is susceptible to the state space explosion problem. The number of system states grows exponentially with the number of parallel processes, making it difficult to analyze systems with a large number of components.

3.3 Other Aspects of Ride Sharing

A real ride-sharing app would have more states than this one, and its logic would depend on complex, real-world data that is difficult to model using either Alloy or FSP. For example, there should be an intermediate waiting state where the rider is waiting for the driver to arrive at their location to pick them up.

Furthermore, a more logical way to assign drivers would be to assign the closest available driver instead of or in conjunction with a first-come-first-served strategy. This would greatly increase the model complexity when working with FSP and Alloy. Similarly, neither tool could model other data-dependent features like dynamic pricing or verify critical performance requirements, such as ensuring user wait times remain below a certain threshold, as they have no built-in concept of time.