David Pick
CM 2403
Formal Methods

1. I spent 1.5 hours on this assignment

2. If there is no planned direction at time t, the value of l.plannedDireciton[t] will be the empty set, {}

3. requestsPending is defined as a separate function so that later it will easier to add requests from outside the elevator.

4. It ensures that the elevator doors are closed, the direction indicator is off, there are no requests pending, there is no intended direction, and the elevator is on the first floor. In other words it initializes the elevator.

5. The four predicates used by someChange are move, requestFloor, openDoors, and closeDoors

6. For the valid predicate to be true at an instance in time, the door indicator must be off, the direction indicator must point the same direction as the planned direction, and the current floor must not be in the set of requested floors.

7. They enable us to make sure that only the things we would like to change from one time to another, actually change.

8. OnlyValid Questions
    1. We could rewrite the only valid assertion as
    ```
    assert OnlyValid {
    Valid[first]
    all t: Time |
    let t' = next[t] |
    Valid[t] => Valid[t']
    }
    ```

    2. I added the constraint

    3. After adding comments, my initTime predicate looked like this:
    ```
    pred initTime[t: Time] {
    all l: Lift {
    l.floor[t] = first[]
    --no l.requestsPending[t]
    --l.doors[t] = Closed
    no l.plannedDirection[t]
    no l.dirIndicator[t]
    }
    }
    ```

9. Assertions like OnlyValid_Move and OnlyValid_Open allow us to write smaller and more understandable predicates that check allowed transitions. It also allows use to test more specific cases, since we can constrain only certain parts of the of the system.

10. The first difference between my model and Curt's is the presence of a planned direction for the lift.

```
// The planned direction of the lift; absence of a mapping
// indicates that the lift has no plans:
plannedDirection: Time -> lone Dir,
```

This field did not exist in my lift. While I don't believe that this field is nessecary for this milestone, I do think that it will make Curt's model easier to adapt for future milestones.

The next thing I noticed was that Curt actually had the doors of his elevator opening and closing. In our model while it was possible for the doors to be closed, we made the decision that since the elevator would always be on a floor for a given time, that it would be easier to simply say that the doors always had to be open

```
fact doorAlwaysOpen {
//The elevator door is always open
//since we are always on a floor
all t: Time | Lift.door[t] = Open
}
```

The largest different I noticed between my code and Curt's was the way Curt defined a valid transition and how we did. Curt only used one fact and called several different predicates from that fact. Instead of this we used several facts, and for each said something like all t, t': Time. While each approach will work, Curt's approach makes the final code shorter and more readable.

Curt's decision to create functions to check each piece of the system not only allows him to create more readable code, but also makes it easier to write that code.

```
fact requestsCannotChange {
//If the floor changes from t1 to t2 and
//neither floor is in the requests set
//make sure that the requests set does not change
//from t1 to t2
all t1, t2: Time, l: Lift |
l.currentFloor[t1] != l.currentFloor[t2] and
l.currentFloor[t1] not in l.requests[t1] and
l.currentFloor[t2] not in l.requests[t2] implies
l.requests[t1] = l.requests[t2]
}
```

This fact from our system checks to ensure that the requests doesn't change from one time to another if the current floor is not in the set of requests. Facts like this could have been avoided had we used several simple predicates like Curt did.