This sequence diagram only corresponds to the passing of a single event containing a MOTOR_ERROR.
Once the elevator receives the motor error flag, the next time it tries to move, it will go into the MOTOR_ERROR state.
The function "checkMotorError" is used when moving or when the elevator is about to move (DOORS_CLOSED to moving).
It checks the flag and triggers the MOTOR_ERROR state when the flag is set to true.

These Thread objects each run in their own class's main loop.
In assignment 3's sequence diagram it show any looping activity, but we feel that it is crucial to our design to include some of the inner loops.

sc1: Is one thread for the Scheduler for Scheduler - Floor communication

sc2: Is one thread for the Scheduler for the Scheduler - Elevator Communications

Actor

FloorSubsystem

Passer (Local)

sc1: Scheduler

sc2: Scheduler

es :ElevatorSubsystem

sc2: Scheduler

es: ElevatorSubsystem

Thread.start()

Thread.start()

main()

Loop
[numThreads<numElevators]

Thread.start()

Thread.start()

Thread.start()

send(:eventPacket)

reply(:ack)

send(:elevatorStatusPacket)

Opt
[recv in motor error state]

reschedule

reply(es:elevatorStatusPacket)

updateAfterReceive(es)

Ref:
    Elevator-Scheduler Communication

Opt
[state="DOORS_OPEN"]

Loop
[destinations.isEmpty()]

Ref:
    Elevator-Scheduler Communication

Opt
[state="MOTOR_ERROR"]

✕

In the case where a MOTOR_ERROR is going to happen, the Scheduler is notified through the "Elevator-Scheduler Communication" above - this happens so the elevator's events are re-scheduled

Opt
[state="MOVING"]

Loop
[es.state != "IDLE"]

Ref:
    Elevator-Scheduler Communication

Here, state is the current state of the elevator.

The es.state is the state retrieved from the scheduler that will be different from the elevator's current state ONLY when moving and it needs to come to a stop

Loop
[!recv(:eventPacket)]

send(:request)

reply(:NULL) or reply(:eventPacket)