

## Specification and implementation code for overtaking in an Autonomous Driving System (ADS):

Perfect, this is one of the classic tactical maneuvers in autonomous driving: overtaking (passing) a slower vehicle on a multi-lane road. It requires perception, decision-making, and motion planning while satisfying both safety and comfort constraints.

I'll give you:

1.  Specification (functional requirements + safety properties).
2.  Decision logic (when to overtake, when to stay).
3.  Implementation code in Python (state machine controller).

- 
1.  Specification: Overtaking Slower Vehicles

## Goal

Enable the ego vehicle to overtake slower vehicles in its lane by safely changing to an adjacent lane, passing, and then returning to the original lane (if desired).

---

## Inputs

- Ego state: position, heading, speed  $v_{ego}$ .
- Lane geometry: multi-lane road structure, lane widths, lane IDs.
- Perceived vehicles: list  $\{id, lane, pos, vel, length\}$ .
- Target speed: desired cruise speed  $v_{desired}$ .
- Traffic rules: e.g., overtake on the left only (or country-specific).
- Safety parameters:
  - $TTC_{min}$  (min time-to-collision with

vehicles in target lane).

- gap\_min\_front and gap\_min\_rear (safe distances to front/rear vehicles in target lane).
    - a\_max, a\_min (acceleration limits).
- 

## Outputs

- Decision: {KEEP\_LANE, OVERTAKE\_INIT, LANE\_CHANGE\_LEFT, LANE\_CHANGE\_RIGHT, RETURN\_LANE}.
  - Trajectory command: lane and speed target.
  - Longitudinal command: acceleration.
- 

## Functional Requirements

1. Detect when a slower vehicle ahead in the current lane blocks progress ( $v_{front} < v_{desired} - \Delta$ ).

2. Check adjacent lane availability (front and rear gaps in target lane are safe).
  3. Initiate lane change only if both front and rear gaps exceed thresholds and predicted  $TTC > TTC_{min}$ .
  4. Smoothly accelerate to overtake while maintaining safe lateral and longitudinal gaps.
  5. After overtaking, return to original lane (if free) or continue in new lane.
- 

## Safety Properties

- Never initiate a lane change if collision predicted with rear/adjacent vehicles.
- Maintain minimum headway in both ego and target lanes.
- Abort overtaking and return to lane if unsafe situation arises.

- Always obey legal overtaking rules (country-dependent: left-hand vs right-hand overtaking).
- 

## 2. Decision Logic

1. Keep Lane (default): cruise at  $v_{desired}$  unless blocked by a slower vehicle.

2. Check Overtaking Need:

- If vehicle ahead in lane has  $v_{front} < v_{desired} - \Delta$  and distance  $< d_{threshold}$ , consider overtaking.

3. Check Lane Availability:

- For each adjacent lane:
  - Compute gap to nearest front vehicle in target lane.
  - Compute gap to nearest rear vehicle in target lane.

- Check both exceed gap\_min and TTC > TTC\_min.
4. Initiate Lane Change: choose safest lane (prefer left lane unless rules allow right).
  5. Perform Lane Change: lateral trajectory planning (smooth shift).
  6. Pass Slower Vehicle: accelerate up to v\_desired.
  7. Return Lane (optional): if original lane is free ahead, merge back.
- 

### 3. Python Implementation (Simplified Controller)

This is a decision-level state machine. Lane-change trajectories are simplified; in real systems, a trajectory generator (e.g., polynomial or MPC-based) would be used.

"""

## `overtake_controller.py`

Simple tactical controller for overtaking slower vehicles on multi-lane roads.

Author: ChatGPT

"""

```
import math
```

```
class OvertakeController:
```

```
    KEEP_LANE = "KEEP_LANE"
```

```
    OVERTAKE_INIT = "OVERTAKE_INIT"
```

```
    LANE_CHANGE_LEFT =
```

```
    "LANE_CHANGE_LEFT"
```

```
    LANE_CHANGE_RIGHT =
```

```
    "LANE_CHANGE_RIGHT"
```

```
RETURN_LANE = "RETURN_LANE"

def __init__(self,
             v_desired=25.0,      # desired
cruise speed (m/s)

             gap_min_front=15.0, # min front
gap in target lane (m)

             gap_min_rear=10.0,  # min rear
gap in target lane (m)

             TTC_min=4.0,        # min
safe time-to-collision (s)

             a_max=2.0,
a_min=-4.0):

self.v_desired = v_desired

self.gap_min_front = gap_min_front
```

```
    self.gap_min_rear = gap_min_rear

    self.TTC_min = TTC_min

    self.a_max = a_max

    self.a_min = a_min

    self.state =

OvertakeController.KEEP_LANE

    self.current_lane = 1 # assume middle lane

    self.target_lane = 1

def _compute_TTC(self, d_rel, v_rel):
    """Compute time-to-collision (positive if ego
catching up)."""

    if v_rel <= 0: # other is faster or same
speed -> no collision

        return float("inf")
```

```
    return d_rel / v_rel

def _safe_gap(self, ego_pos, ego_speed, vehicles,
target_lane):
    """Check if target lane has safe gaps ahead
and behind."""
    front_gap, rear_gap = float("inf"),
float("inf")
    front_speed, rear_speed = None, None
    for v in vehicles:
        if v['lane'] == target_lane:
            d_rel = v['pos'] - ego_pos
            if d_rel > 0:  # vehicle ahead
                if d_rel < front_gap:
```

```
    front_gap = d_rel

    front_speed = v['speed']

    else: # vehicle behind

        if abs(d_rel) < rear_gap:

            rear_gap = abs(d_rel)

            rear_speed = v['speed']

# front safety: enough distance OR TTC
long enough

    front_ok = (front_gap >=
self.gap_min_front) or ¥
                                (front_speed and
self._compute_TTC(front_gap, ego_speed -
front_speed) > self.TTC_min)
```

```
# rear safety: enough distance OR TTC long  
enough
```

```
    rear_ok = (rear_gap >= self.gap_min_rear)
```

or ¥

```
        (rear_speed and  
        self._compute_TTC(rear_gap, rear_speed -  
        ego_speed) > self.TTC_min)
```

```
    return front_ok and rear_ok
```

```
def update(self, ego_pos, ego_speed, vehicles):
```

```
    """
```

```
    ego_pos: scalar longitudinal position (m)
```

```
    ego_speed: ego speed (m/s)
```

```
    vehicles: list of dicts
```

```
{ 'id','lane','pos','speed','length' }
```

"""

decision = self.state

a\_cmd = 0.0

# Find front vehicle in current lane

front\_vehicle = None

min\_dist = float("inf")

for v in vehicles:

    if v['lane'] == self.current\_lane and  
    v['pos'] > ego\_pos:

        d = v['pos'] - ego\_pos

        if d < min\_dist:

            min\_dist = d

            front\_vehicle = v

```
# --- State machine ---  
  
if self.state ==  
  
OvertakeController.KEEP_LANE:  
  
    if front_vehicle and  
front_vehicle['speed'] < self.v_desired - 2.0 and  
min_dist < 30.0:  
  
        # Need to overtake  
  
        if self._safe_gap(ego_pos,  
ego_speed, vehicles, self.current_lane + 1):  
  
            self.state =  
  
OvertakeController.LANE_CHANGE_LEFT  
  
            self.target_lane =  
self.current_lane + 1  
  
            decision = self.state  
  
            elif self._safe_gap(ego_pos,  
ego_speed, vehicles, self.current_lane - 1):
```

```
    self.state =  
    OvertakeController.LANE_CHANGE_RIGHT  
  
    self.target_lane =  
    self.current_lane - 1  
  
    decision = self.state  
  
else:  
  
    decision =  
    OvertakeController.KEEP_LANE  
  
else:  
  
    decision =  
    OvertakeController.KEEP_LANE  
  
    # Accelerate toward desired speed  
  
    if ego_speed < self.v_desired:  
  
        a_cmd = min(self.a_max,  
        (self.v_desired - ego_speed) * 0.5)
```

```
        elif self.state in
            [OvertakeController.LANE_CHANGE_LEFT,
             OvertakeController.LANE_CHANGE_RIGHT]:
                # Assume lane change completes after
                # some condition (simplified here)
                self.current_lane = self.target_lane
                self.state =
                OvertakeController.KEEP_LANE
                decision = self.state
                a_cmd = min(self.a_max,
                            (self.v_desired - ego_speed) * 0.5)
            return { 'decision': decision, 'a_cmd': a_cmd,
                     'lane': self.current_lane }

# Example run
```

```
if __name__ == "__main__":
    ctrl = OvertakeController()
    ego_pos, ego_speed = 0.0, 20.0
    vehicles = [
        {'id': 'v1', 'lane': 1, 'pos': 25.0, 'speed': 15.0, 'length': 4.5},
        # slow car ahead
        {'id': 'v2', 'lane': 2, 'pos': 50.0, 'speed': 28.0, 'length': 4.5},
        # faster car in left lane
    ]
    for step in range(10):
        out = ctrl.update(ego_pos, ego_speed,
                          vehicles)
        print(f"Step {step}: {out}")
```

```
ego_speed += out['a_cmd'] * 0.2 # simple  
integration
```

```
ego_pos += ego_speed * 0.2
```

---

#### 4. Test Scenarios

1. Slow car ahead, left lane free → ego overtakes to left.
2. Slow car ahead, left lane blocked, right lane free (if legal) → ego overtakes right.
3. Both adjacent lanes blocked → ego slows and waits.
4. After passing → ego continues cruising or merges back if lane is clear.