# Description of the CPEC SHK position

## Project description and long-term goals

### Demonstrator: Constant Speed Assistant

The lane change assistant will help robots (e.g., in form of an autonomous fork lift or car) to maintain a constant speed - if traffic permits. The target speed is set per robot and per destination. For example, in the case of a fork lift, depending on what is transported (weight and size) the fork lift need to set different speeds. The autonomous fork lifts are operating in a big warehouse and hence, we do not want to slow down the fork lifts. Hence, we have a constant speed assistant that ensures that fork lifts can go with their maximum speed whenever possible. Each fork lift is autonomous and executes the tasks it should work on.

In our demonstrator setup, the fork lifts will run on a three lane racetrack. Like on the freeway, the robots will drive on the right most lane of the race track and will change lanes if there is an obstacle with a lower speed in front. Obstacles can be passed on the right or left - at any speed. On the right should only be needed if robots have not activated the constant speed assistant. A robots must change lane if this would help to reach the target speed of the robot. Changing lanes must be safe, i.e., no crash must happen.

### Motivation

The motivation of this work is to investigate on how we can detect and deal with software bugs as well as bugs in Machine Learning based models.
We need to be able to deal with these bugs. Traditionally, one uses replications like TMR (triple modular redundancy) that permits to tolerate the fault of a minority of the processes. However, TMR cannot tolerate software bugs since if a software bug is activated, it would be activated in all three copies: TMR must ensure a deterministic execution of the replicas and hence, even Heisenbugs are most likely activated in all three replicas.

Moreover, if we want to address software bugs, we can follow a N-version approach - in which we have N-versions instead of only 1 software version. This can be combined with TMR. However, this raises the question if it would not be more cost effective to spend the resources used to implement the N versions into one 1 version only to ensure that the likelihood that a software bug is activated is negligible. The answer to this question depends on the tools and teams used to build the N versions versus the 1 version and also from N.

From a commercial perspective, one wants to reduce the replicas from three to two even one only. Moreover, we would like to decouple the versions (aka variants) such that we reduce the chances of common mode failures of the replicas. The version can use different algorithms and hence, the might come up with different ways to control the robots. For example, one variant might change lane earlier while other variants might change lane later. This implies that we cannot just compare the outputs of the two variants since they, for example, might decide to change lanes at different points in time. However, how can we detect that a copy is faulty, i.e., violates the specification and potential the safety of the system?

Our objective is to design a two channel system, i.e., contains two replicas with independently developed software variants.

## Architecture

The software architecture of the robots is as follows:

1. Channel 1 and 2: Constant Speed Assistant (CSA1 and CSA2)
    1. An CSA gets inputs from the sensors, the last command to the motors that were issued and computes a new set of commands for the motors.
    2. An CSA can be written in Rust (or Python or C if needed)
    3. An CSA also produces logic statement (an **explication**) that describe why a certain action was taken.
2. A supervisor:
    1. The supervisor gets the motor commands and explications CSA

1 and CSA2

2. The supervisor checks that the explication is consistent with the motor commands
3. The supervisor decides on which motor command(s) to forward to the motors and forwards this to the CSA

Example:

```
Variant 1: might issue the following explications:

lane(v1, left, minSpeed1, maxSpeed1) # changing to the left
lane, the speed needs to be in this range ; if not
permitted, minSpeed > maxSpeed
lane(v1, my, minSpeed2, maxSpeed2)  # keeping this lane,
the speed needs to be in this range
lane(v1, right, minSpeed3, maxSpeed3) # changing to the
right lane, the speed needs to be in this range

action(v1, left, L1, speed_1); # lane and speed

Variant 2: might issue the following explications:

lane(v2, left, min2Speed1, maxSpeed2)
lane(v2,my, minSpeed2, maxSpeed2)
lane(v2,right, minSpeed3, maxSpeed3)

action(v2, right, v2speed_2); # lane and speed

Supervisor:

exists action(_,L, S) in {actions} such that for lane(v1,
L, m1, M1) and lane(v2, L, m2, M2) => m1 <= S <= M1 and m2
<= S <= M2
```

## Task 1:

- get yourself familiarized with the existing software

## Task 2: „*implement a way to execute the action from above*“

- design an implement a software module for Lego Robots that follow lanes and changing lanes / speed if requested.
- the API to this model is as follows:

> **setLaneAndSpeed(targetLane : int, targetSpeed: float):**
> - we have lanes 0, …, n ; 0 is the right most lane
> - the speed is in meters per second

When calling this function, the robot is supposed to change the lane to **targetLane** (unless it is already in the correct lane) and
changes speed to **targetSpeed** unless it is already on the target speed.

## Task 3: „*implement a way to generate the lane predicate from above*“

- define a software module that - given the available software and hardware sensors, determines for a robot currently in Lane i:
  - the minimum and maximum speed permitted in lanes i-1, i, i+1 (minimum will typically be 0, maximum is determined by the speed of the vehicle ahead)
  - if a lane does not exists, the maximum speed is smaller than the minimum speed
- one can query this information via a function **getPermittedLaneSpeeds**

## Task 4: „*implement a way to generate to propose an*

*action*“

- define a software module that suggests a targetLane and targetSpeed
  - this module provides an API to set the targetSpeed: setTargetSpeed(Speed: float)
  - this module generates a preferred: **getPreferredLaneAndSpeed -> (targetLane : int, targetSpeed: float)**

## Task 5: „*put everything together… without supervisor*“

- integrate Tasks2-4 with existing implementation to run on LegoRobots.