

Entwicklerdokumentation für den Standard-Workspace

Table of Contents

Motivation	3
Abhängigkeiten	3
Allgemeine Informationen	4
PSAF 1	4
PSAF 2	4
Carolo Cup	4
Schnelleinstieg	4
Installation und Setup	4
Konfiguration	5
Bauen	5
Ausführung	5
Workspace	6
Übersicht	6
Workspace kopieren	6
Starten der Nodes	7
Aufbau des Workspaces	8
Aufbau eines Packages	8
Erzeugen eines Packages	9
Übersicht über die Pakete	9
Parameter	19
Nachrichten	20
color/image_raw und depth/image_rect_raw	20
lane_detection/lane_markings	20
lane_detection/stop_line	21
object_detection/obstacle	22
parking_detection/parking_spot	22
sign_detection/sign	23
state_machine/state	24
status/status_info	24
trajectory/trajectory	25
watchdog/error_message	26
uc_bridge Nachrichten	26
State Machine	29
Testen	33

Codestyle- und Code-Konformitätsprüfungen	33
Unit-Tests	33
Integration-Tests	33
Simulationstests	34
CI-Pipeline	34
Ausführen der Tests	35
Spurerkennung	37
Zustandsautomat	51
Startbox	68
Setup	79
Ubuntu Installation	79
Auto Installation	83
Einrichtung	83
Entwicklungsumgebung	86
Bekannte Probleme und FAQ	88
Bauen des Workspaces	88
Fehlermeldung: zbar.h - No such file or directory	90
Fehlermeldung: cv_bridge: No such file or directory	90
Fehlermeldung: "permission denied <filename>"	90
Welche Schilder müssen erkannt werden?	91

Diese Dokumentation enthält Informationen über den Workspace für das PSAF 1 und 2 sowie für die Teilnahme am Carolo Cup. Die Dokumentation ist wie folgt aufgebaut:

Im Abschnitt [Motivation](#) wird zunächst beschrieben, warum ein allgemeiner Ausgangspunkt für die Entwicklung essenziell ist.

Anschließend werden [allgemeine Informationen](#) über die Seminare sowie über den Carolo Cup gegeben.

Im Abschnitt [Schnelleinstieg](#) ist beschrieben, wie am schnellsten mit der Entwicklung begonnen werden kann. Um den Schnelleinstieg verwendet zu können, muss bereits eine fertig eingerichtete Entwicklungsumgebung vorhanden sein.

Im Abschnitt [Workspace](#) werden die einzelnen Pakete sowie deren Kommunikationsschnittstellen erläutert.

Im Abschnitt [Parameter](#) wird die gezeigt, wie Parameter genutzt werden können, um das Verhalten einzelner Nodes während der Laufzeit zu ändern. Diese Funktionalität ist bereits beispielhaft im Paket [Spurerkennung](#) implementiert.

Die ausgetauschten Nachrichten werden im Abschnitt [Nachrichten](#) beschrieben.

Im darauf folgenden Kapitel [State Machine](#) wird der im Rahmen dieses Projekts verwendete

Zustandsautomat vorgestellt. Der Zustandsautomat ist bereits fertig implementiert und muss für die Regelung des Kontrollflusses genutzt werden.

Im Kapitel [Testen](#) wird zunächst eine allgemeine Übersicht über die Rahmen dieser Arbeit verwendeten Testarten gegeben. Daran anschließend folgt die Beschreibung der bereits implementierten Test für die jeweiligen Pakete.

Das Kapitel [Setup](#) stellt eine Installationsanleitung sowie eine Anleitung zum Starten des Workspace zur Verfügung.

Im Abschnitt [FAQ](#) werden Fragen und Antworten zu den Paketen bereitgestellt sowie Lösungen für die häufigsten Probleme aufgezeigt.



Die Dokumentation befindet sich derzeit in der Entwicklung. Die Studierenden müssen die Dokumentation ergänzen, falls Änderungen oder Erweiterungen durchgeführt worden sind. Dies gilt insbesondere für das Hinzufügen neuer Testfälle.



Für mehr Informationen über Nachrichtentypen sowie die allgemeine Struktur kann die [Dokumentation der libpsaf](#) herangezogen werden.

Motivation

Die Projektseminare "Autonomes Fahren 1" (PSAF 1) und "Autonomes Fahren 2" (PSAF 2) werden jedes Semester im Wechsel gehalten. Im PSAF 1 werden die grundlegenden Softwarepakete für ein autonomes Modellauto erarbeitet und von den verschiedenen Gruppen implementiert. Dies beinhaltet eine Fahrbahnerkennung, einen Einparkvorgang, die Regelung des Fahrzeugs sowie eine Hinderniserkennung. Im Seminar PSAF 2 werden diese Ergebnisse weiterverwendet und für das leistungstärkere Wettkampfauto angepasst und optimiert. Das Wettkampffahrzeug verfügt über eine erweiterte Sensorsuite und soll im [Carolo Cup](#) der TU Braunschweig eingesetzt werden.

Um den Studierenden die Einarbeitung möglichst einfach zu gestalten, wurde dieser Template-Workspace als Grundlage der Entwicklung erstellt. Durch die Bereitstellung einer festen Grundstruktur soll gewährleistet werden, dass der Code von zukünftigen Gruppen leicht weiterverwendet werden kann und der Austausch einzelner Softwarepakete einfach möglich ist. Die einzelnen Pakete erben von den Interfaces, die in der [libpsaf](#) spezifiziert sind.

Abhängigkeiten

Der Template - Workspace ist von der [libpsaf](#) und der [uc_bridge](#) abhängig. Die Abhängigkeit von einer bestimmten Version erfolgt implizit über die Wahl des Docker-Containers.

- Aktueller Docker Container: **latest**
- libpsaf **3.1.0**
- ucbridge: **2.2.0**

Die Versionen können geändert werden, indem in der `.gitlab-ci.yml` eine andere Image Version ausgewählt wird.

Wie ein neues Release erstellt wird und wie die Versionierung angepasst wird, ist im [Images Repository](#) erklärt.

Der Abschnitt [Setup](#) beschäftigt sich mit der Einrichtung der Entwicklungsumgebung. Dieser Schritt ist nicht nötig, wenn man direkt auf den Autos arbeitet, da diese bereits vollständig eingerichtet sind.

Allgemeine Informationen

PSAF 1

Generelle Informationen und eine Dokumentation für das PSAF 1 ist unter [PS_AF_1.pdf](#) im [PS AF 1 - Dokumentation](#) Repo zu finden. Weitere Material im Repo [Allgemeine Informationen](#).

PSAF 2

Informationen zum verwendeten Fahrzeug im PSAF 2 sind in der [Dokumentation](#) für das Carolo Cup Auto zu finden.

Carolo Cup

Informationen über den Carolo Cup sind in den Regularien für den [Basic Cup](#) und für den [Master Cup](#) zu finden.

Schnelleinstieg

In diesem Abschnitt wird ein beschleunigter Einstieg beschrieben. Hierfür sind die wichtigsten Schritte kompakt zusammen gefasst. Es wird dennoch empfohlen, die komplette Dokumentation zu lesen und die vorhandenen Pakete zu verstehen. Für den Schnelleinstieg wird davon ausgegangen, dass sowohl die `libpsaf`, die `uc_bridge` sowie `R0S2 Foxy` bereits installiert sind.

Installation und Setup

Zunächst muss der Standard-Workspace heruntergeladen und gebaut werden.

```
git clone https://git-ce.rwth-aachen.de/af/ws-template
```



Der im Befehl genannte Link kann auch anders sein, falls mit dem dedizierten Carolo Cup Workspace gearbeitet wird. In diesem Fall muss stattdessen aus folgendem Repository heruntergeladen werden: <https://git-ce.rwth-aachen.de/af/cc/cc-ws>

Konfiguration

Nachdem Download sollte zunächst die Konfiguration geprüft und gegebenenfalls angepasst werden. Hierzu muss die Datei `src/psaf_configuration/include/psaf_configuration/configuration.hpp` geöffnet werden. Insbesondere muss darauf geachtet werden, dass die Flag für das verwendete Auto korrekt gesetzt ist.

Bauen

Nachdem die Konfiguration geprüft wurde, muss der Workspace gebaut werden:

```
cd ws-template/  
colcon build
```

Falls es beim Bauen zu Fehlern kommt, bitte das Kapitel [FAQ](#) beachten.

Ausführung

Nach der Installation muss der Workspace erst gesourced werden, bevor die Software ausgeführt werden kann.

```
source install/setup.bash
```

Dieser Befehl muss jedes Mal erneut ausgeführt werden, wenn ein neues Terminal geöffnet wird. Alternativ kann der Befehl auch in die `~/.bashrc` Datei eingetragen werden.

Zum Starten eines Pakets muss das entsprechende Launchfile ausgeführt werden.

```
ros2 launch <package_name> <launchfile_name>
```

Am Beispiel der Spurerkennung:

```
ros2 launch psaf_lane_detection lanedetection.launch.py
```



Zum Starten eines Pakets bitte immer das entsprechende Launchfile ausführen. Bei Verwendung des Befehls `ros2 run <package_name> <node_name>` kann es zu Fehlern kommen.

Um alle Pakete auf einmal zu starten, können die Launchfiles aus dem Ordner `psaf_launch` ausgeführt werden.

```
ros2 launch psaf_launch main_psaf1.launch.py # Für das PSAF 1 Fahrzeug
ros2 launch psaf_launch main_psaf2.launch.py # Für das PSAF 2 Fahrzeug
```

Workspace

Übersicht

Der Workspace stellt den Ausgangspunkt für die Entwicklung im PSAF 1 und 2 dar. Der Workspace enthält bereits Pakete, die für die Entwicklung der Anwendungen in PSAF 1 und 2 benötigt werden. Die Kommunikationsschnittstellen zwischen den einzelnen Paketen sind bereits definiert und implementiert. Aufgabe der Studierenden ist es, die Funktionalität zu implementieren. So muss beispielsweise eine Spurerkennung, die Regelung, Objekterkennung und Parkplatzsuche umgesetzt werden. In der Abbildung [ROS Graph](#) sind die einzelnen Nodes sowie die Nachrichten, die diese untereinander austauschen, dargestellt.

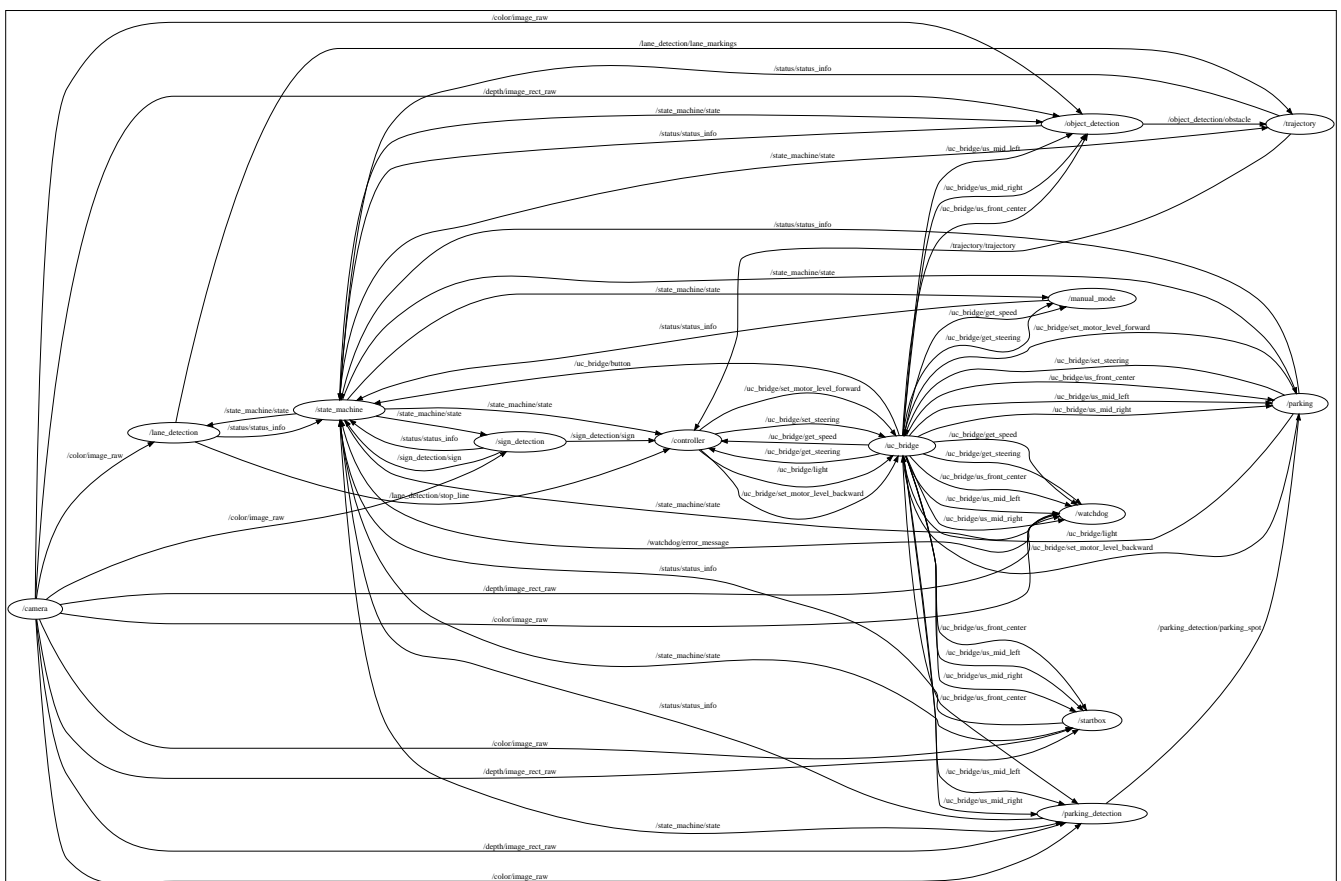


Figure 1. ROS Graph

Workspace kopiëren



Der Workspace besitzt Abhängigkeiten zur `libpsaf` und zur `uc_bridge`. Auf dem Fahrzeug sind beide Pakete bereits installiert. Bei der Entwicklung auf eigenen PCs müssen diese erst installiert werden. Eine Anleitung für die Installation ist im Abschnitt [Setup](#) zu finden.

Jede Gruppe erstellt sich eine Kopie des Start-Workspace.

Hierfür muss das Repository zunächst geklont werden:

```
git clone https://git-ce.rwth-aachen.de/af/ws-template.git
```

Danach muss das Repository gebaut werden:

```
cd ws_template/  
colcon build
```

Anschließend muss der Workspace noch gesourct werden:

```
source install/local_setup.bash
```

Damit das sourcen nicht jedes Mal erneut durchgeführt werden muss, kann der folgende Befehl verwendet werden, um das sourcen der bashrc Datei hinzuzufügen:

```
cd ws_template  
echo "source install/local_setup.bash" >> ~/.bashrc
```



Sollte es beim Bauen zu Fehlermeldungen kommen, sind Lösungen für die gängigsten Probleme im Abschnitt [Bekannte Probleme und FAQ](#) zu finden.

Starten der Nodes

Um die Nodes zu starten, verfügt jede Node über ein Launch File. Dieser liegt im Ordner **launch** des jeweiligen Pakets. Für die **LaneDetection** Node ist das Launch File beispielsweise **lanedetection.launch.py**.

Um die Node mit dem Launch-File zu starten, muss der folgende Befehl in einem Terminal ausgeführt werden.

```
ros2 launch <package_name> <launch_file_name>
```

Am Beispiel der LaneDetection Node wird folgender Befehl ausgeführt:

```
ros2 launch psaf_lane_detection lanedetection.launch.py
```

Der Workspace enthält auch Launch-Files, um alle Nodes gleichzeitig zu starten. Dies beinhaltet die Kamera und die **uc_bridge**. Da es Unterschiede bei den Parametern der **uc_bridge** sowie bei der verwendete Kamera beim PSAF 1 Auto und beim PSAF 2 Auto gibt, sind auch zwei Launch-Files

vorhanden.

Zum Starten aller Nodes auf dem PSAF 1 Auto:

```
ros2 launch psaf_launch main_psaf1.launch.py
```

Zum Starten aller Nodes auf dem PSAF 2 Auto:

```
ros2 launch psaf_launch main_psaf2.launch.py
```



Zum Starten der einzelnen Pakete muss das Launchfile verwendet werden. Beim Starten der Node über den Befehl `ros2 run <package_name> <node_name>` werden nicht alle Parameter geladen. Insbesondere wird die `update()`-Methode nicht periodisch aufgerufen, wodurch die Node keine Ergebnisse veröffentlicht.

Aufbau des Workspaces

Der Workspace enthält Ordner (Packages) für die im Carolo Cup benötigten Funktionen. Diese sind im nachfolgend genauer beschrieben. Informationen über die einzelnen Interfaces, Subscriber, Publisher und Nachrichtentypen können der [Dokumentation der libpsaf](#) entnommen werden. Eine Kurzübersicht über die Nachrichten und deren Datentypen ist im Abschnitt [Nachrichten](#) zu finden.

Aufbau eines Packages

Ein Package besteht in der Regel aus folgenden 5 Unterordnern. Ausnahmen bilden lediglich die Ordner `psaf_configuration`, `psaf_launch` und `psaf_utils`.

- `config/` - Konfigurationsdateien
- `include/` - Header-Dateien
- `launch/` - Launch-Files für das entsprechende Package
- `src/` - Quellcode
- `test/` - Test-Dateien für das entsprechende Package

Im Ordner `config/` kann die Update-Frequenz der jeweiligen Node festgelegt werden. Derzeit ist die Frequenz aller Nodes auf 30 Hz gesetzt. Die Frequenz gibt vor, wie oft die Methode `update()` der jeweiligen Node aufgerufen wird. Innerhalb dieser Methode müssen die Publisher aufgerufen werden.



Die Änderung der Frequenz im config File hat nur bei Verwendung des launch scripts einen Einfluss.

Erzeugen eines Packages

Generell sollten keine weiteren Pakete benötigt werden. Falls eine Regeländerung im Carolo Cup dies dennoch erforderlich macht, kann ein neues Package wie folgt erzeugt werden:

```
cd src/  
ros2 pkg create --build-type ament_cmake <package_name> --dependencies  
<[dependencies]>
```

Um beispielsweise das Paket **controller** zu erzeugen, wurde folgender Befehl genutzt:

```
ros2 pkg create --build-type ament_cmake controller --dependencies rclcpp libpsaf  
psaf_configuration
```

Die Abhängigkeiten können jederzeit in der **package.xml** Datei des Paketes angepasst werden.

Übersicht über die Pakete

Jedes Paket(Node) enthält bereits die benötigte Basisstruktur. Dies beinhaltet die Erzeugung der Node mit den entsprechenden Subscribern und Publishern. Die Topic-Namen sind im Paket **psaf_configuration** definiert und dürfen nicht verändert werden. Jede Node verfügt außerdem über die Methode **void update()**. Diese wird periodisch mit der in der **config** Datei gesetzten Frequenz aufgerufen. (Standard: 30 HZ). Innerhalb dieser Methode sollen die Publisher aufgerufen werden, um die Ergebnisse der Node zu veröffentlichen. Jedes Paket muss mit Testfällen ausgestattet werden. Mehr Informationen, über die bereits existierenden Tests sind im Kapitel **Testen** zu finden.

psaf_configuration

Dieses Paket wird für die Konfiguration des Workspace verwendet. Dies beinhaltet die Definition der Node- und Topic-Namen. Die Topic-Namen sind fest vorgegeben und dürfen nicht geändert werden. Alle anderen Pakete besitzen eine Abhängigkeit zu diesem Paket.



Bevor mit der Entwicklung begonnen werden kann, muss in der Datei **psaf_configuration/include/psaf_configuration/configuration.hpp** die Variable für das verwendete Fahrzeug gesetzt werden. Beim Wettkampf Auto (PSAF 2) muss der Wert auf **false** gesetzt werden.

```
#define PSAF1 true // für PSAF 1
```

Innerhalb der Datei gibt es noch zwei weitere Kontrollvariablen.

```
#define DEBUG false // Ermöglicht die Ausgabe von Debug-Informationen
```

```
#define FORCE_TEST_PASS true // Erzwingt das Bestehen mancher Testfälle
```



Zum Abschluss des Projekts muss die Flag `FORCE_TEST_PASS` auf `false` gesetzt werden, da alle Testfälle ausgeführt und bestanden werden müssen.

psaf_controller ("controller")

Die `Controller` Node ist für die Regelung des Fahrzeugs verantwortlich. Die Node verfügt über folgende Kommunikationsschnittstellen:

Schnittstelle	Topic Name	Beschreibung
StateSubscriber	<code>state_machine/state</code>	Der aktuelle Zustand der State Machine
TrajectorySubscriber	<code>trajectory/trajectory</code>	Die berechnete Trajektorie
StopLineSubscriber	<code>lane_detection/stop_line</code>	Informationen über eine Stoplinie
SignSubscriber	<code>sign_detection/sign</code>	Informationen über ein aktuell erkanntes Schild
SteeringSubscriber	<code>uc_bridge/get_steering</code>	Der tatsächliche Einschlagswinkel des Fahrzeugs
SpeedSubscriber	<code>uc_bridge/get_speed</code>	Die vom Fahrzeug gemessene Geschwindigkeit in cm/s
SteeringPublisher	<code>uc_bridge/set_steering</code>	Übermittelt den gewünschten Lenkwinkel in 1/10 Grad oder 1/100 rad
SpeedPublisher	<code>uc_bridge/set_motor_level</code>	Siehe Fußnote (*)
LightPublisher	<code>uc_bridge/light</code>	Topic zur Ansteuerung der Lichter

(*) Der Publisher besteht intern aus zwei separaten Publishern: `uc_bridge/set_motor_level_forward` und `uc_bridge/set_motor_level_backward`. Übergeben wird jedoch die Geschwindigkeit in cm/s. Der Publisher rechnet die Geschwindigkeit intern in ein Motorlevel um und veröffentlicht das Ergebnis auf dem entsprechenden Topic. Falls negative Werte übergeben werden, wird auf dem `uc_bridge/set_motor_level_backward` Topic gesendet.

psaf_lane_detection("lane_detection")

Die `LaneDetection` Node muss mehrere Aufgaben erfüllen. Die Aufgaben sind:

1. **Spurerkennung** - Hauptaufgabe der Node. Muss immer erfüllt werden.

2. **Startlinienerkennung** - Nebenaufgabe. Muss für die Disziplin "Rundkurs mit Einparken" erfüllt werden.
3. **Stopplinienerkennung** - Nebenaufgabe. Muss für die Disziplin "Rundkurs mit Hindernissen" erfüllt werden.

Die Node verfügt über folgende Kommunikationsschnittstellen:

Schnittstelle	Topic Name	Beschreibung
ImageSubscriber	color/image_raw	Das Farbbild der Kamer mit einer Auflösung von 640x480 Pixeln
StateSubscriber	state_machine/state	Der aktuelle Zustand der State Machine
LaneMarkingsPublisher	lane_detection/lane_markings	Informationen über die erkannten Spurmarkierungen
StopLinePublisher	lane_detection/stop_line	Informationen über die erkannten Stoplinie. Soll nur für die Disziplin "Rundkurs mit Hindernissen" verwendet werden.
StatusInfoPublisher	status/status_info	Statusinformationen für die StateMachine

Die Informationen über die erkannte Startlinie werden als **StatusInfo** an die StateMachine gesendet. Die entsprechende **StatusInfo** ist **PARKING_INTENT**, da nach dem Überfahren der Startlinie die Parkplatzsuche gestartet werden soll.

Der Kontrollfluss für die LaneDetection kann wie folgt aussehen:

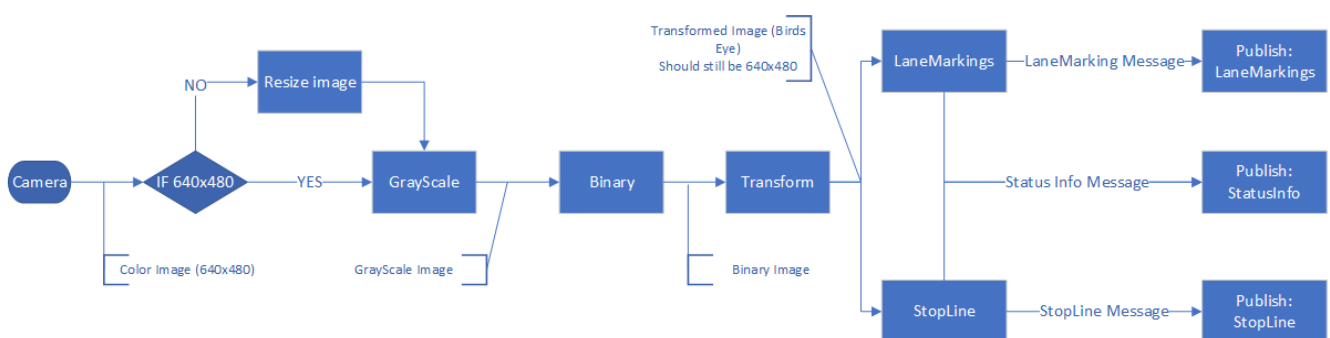


Figure 2. Beispielhafte Verarbeitungspipeline für die Spurerkennung.

Das Kamerabild im Format 640x480 wird vom **ImageSubscriber** empfangen. Falls das Bild nicht in der Auflösung 640x480 vorliegt, wird es entweder verkleinert oder vergrößert. Anschließend wird das Bild in ein Graustufenbild umgewandelt und danach in ein Binärbild. Das Binärbild wird in die Vogelperspektive transformiert. Hieraus werden dann die Spurmarkierungen extrahiert und überprüft, ob eine Stoplinie im Bild vorhanden ist. Die Erkennung der Startlinie kann in einer der beiden Methoden durchgeführt werden. Abschließend werden die Informationen in den entsprechenden Nachrichten verpackt und periodisch über die Methode **update()** veröffentlicht.



Die obige Verarbeitungspipeline ist nur als Beispiel gedacht. Falls eine andere Verarbeitungspipeline genutzt werden soll, ist dies möglich. Hierbei ist darauf zu achten, dass die Ergebnisse im korrekten Format auf den entsprechenden Topics gesendet werden. Falls die Pipeline verändert wird, müssen die Testfälle entsprechend modifiziert werden.



Die Stopplinienerkennung beinhaltet auch die Berechnung der Halteposition an Fußgängerüberwegen. Diese Funktion ist im Master-Cup erforderlich.



Die Parameter für die [Homography](#) sind in der Datei `psaf_lane_detection/include/psaf_lane_detection/lane_detection_node.hpp` gesetzt. Die Parameter unterscheiden sich für das PSaf 1 und PSaf 2 Fahrzeug. Um das beste Ergebnis zu erreichen, müssen die Werte für das verwendete Fahrzeug berechnet und im Header angepasst werden.

psaf_launch

Dieses Paket enthält Launch-Files für den Workspace. Hierüber können alle Pakete, die `uc_bridge` sowie die Realsense Kamera gestartet werden.

Zum Starten der Nodes auf dem PSaf 1 Auto:

```
ros2 launch psaf_launch main_psaf1.launch.py
```

Zum Starten der Nodes auf dem PSaf 2 Auto:

```
ros2 launch psaf_launch main_psaf2.launch.py
```

Die Konfigurationsdateien für die `uc_bridge` und die `Realsense Kamera` sind im Ordner `config` gespeichert.

psaf_manual_mode ("manual_mode")

Die `ManualMode` Node ist eine Hilfs-Node, um auf Events während der Fahrt mit Fernsteuerung zu reagieren. Die Signale der Fernsteuerung werden direkt auf dem `uc_board` verarbeitet. Die `uc_bridge` meldet den Eintritt in den manuellen Modus über das Topic `uc_bridge/manual_signals`. Die `ManualMode` Node empfängt diese und informiert die StateMachine über den Eintritt und Verlassen des manuellen Modus. Sie empfängt Informationen über die zurückgelegte Strecke und Richtung während der manuellen Fahrt. Die Node verfügt über folgende Kommunikationsschnittstellen:

Schnittstelle	Topic Name	Beschreibung
---------------	------------	--------------

StateSubscriber	state_machine/state	Der aktuelle Zustand der State Machine
ManualModeSubscriber	uc_bridge/manual_signals	Informationen über den manuellen Modus. 1, falls manuell gefahren wird, 0 sonst.
SteeringSubscriber	uc_bridge/get_steering	Der tatsächliche Einschlagswinkel des Fahrzeugs
SpeedSubscriber	uc_bridge/get_speed	Die vom Fahrzeug gemessene Geschwindigkeit in cm/s
StatusInfoPublisher	status/status_info	Statusinformationen für die StateMachine

psaf_object_detection ("object_detection")

Die **ObjectDetection** Node ist für die Erkennung von Hindernissen auf der Fahrbahn zuständig. Dies beinhaltet Objekte mit Vorfahrtrecht an Kreuzungen. Zur Erkennung der Objekte kann das Farbbild, das Tiefenbild sowie die Ultraschallsensoren genutzt werden. Die Anzahl der Ultraschallsensoren unterscheidet sich je nach Fahrzeug. Die Node verfügt über folgende Kommunikationsschnittstellen:

Schnittstelle	Topic Name	Beschreibung
ImageSubscriber	color/image_raw , depth/image_raw_rect	Das Farbbild der Kamer mit einer Auflösung von 640x480 Pixeln. Das Tiefenbild der Kamera mit einer Auflösung von 1280x720 Pixeln
UltrasonicSubscriber	uc_bridge/<pos>	<pos> beschreibt die Position des Sensors am Fahrzeug
StateSubscriber	state_machine/state	Der aktuelle Zustand der State Machine
ObjectPublisher	object_detection/obstacle	Informationen über ein Hindernis. Jedes Hindernis muss einzeln veröffentlicht werden.
StatusInfoPublisher	status/status_info	Statusinformationen für die StateMachine

psaf_parking ("parking")

Die **Parking** Node ist für das Einparken und das Ausparken zuständig. Die Erkennung eines Parkplatzes ist nicht Teil dieser Node. Zur Kommunikation verfügt das Paket über folgende

Kommunikationsschnittstellen:

Schnittstelle	Topic Name	Beschreibung
ParkingSpotSubscriber	parking_detection/parking_spot	Informationen über den erkannten Parkplatz.
UltraSonicSubscriber	uc_bridge/<pos>	<pos> beschreibt die Position des Sensors am Fahrzeug
StateSubscriber	state_machine/state	Der aktuelle Zustand der State Machine
SteeringPublisher	uc_bridge/set_steering	Der Einschlagswinkel des Fahrzeugs in 1/10 Grad oder 1/100 rad
SpeedPublisher	uc_bridge/set_motor_level	Die Geschwindigkeit des Fahrzeugs in cm/s
StatusInfoPublisher	status/status_info	Statusinformationen für die StateMachine
LightPublisher	uc_bridge/light	Informationen über die Lichter des Fahrzeugs

psaf_parking_detection ("parking_detection")

Die **ParkingDetection** ist für die Erkennung von parallelen und senkrechten Parkplätzen zuständig. Die Informationen über erkannte Parkplätze werden über das Topic [parking_detection/parking_spot](#) veröffentlicht. Zur Erkennung der Parkplätze kann das Farbbild, das Tiefenbild sowie die Ultraschallsensoren genutzt werden. Die Anzahl der Ultraschallsensoren unterscheidet sich je nach Fahrzeug. Die Node verfügt über folgende Kommunikationsschnittstellen:

Schnittstelle	Topic Name	Beschreibung
ImageSubscriber	color/image_raw , depth/image_rect_raw	Das Farbbild und Tiefenbild der Kamera
UltrasonicSubscriber	uc_bridge/<pos>	<pos> beschreibt die Position des Sensors am Fahrzeug
StateSubscriber	state_machine/state	Der aktuelle Zustand der State Machine
ParkingSpotPublisher	parking_detection/parking_spot	Informationen über einen erkannten Parkplatz.
StatusInfoPublisher	status/status_info	Statusinformationen für die StateMachine

psaf_sign_detection ("sign_detection")

Die **SignDetection** Node ist für die Erkennung von Verkehrszeichen zuständig. Die möglichen Schilder sind im Abschnitt [Faq](#) beschrieben. Die Schilderkennung ist nur im Master Cup erforderlich. Im Rahmen des PSaf 1 kann die Schilderkennung als Zusatzaufgabe implementiert werden. Die Node verfügt über folgende Kommunikationsschnittstellen:

Schnittstelle	Topic Name	Beschreibung
ImageSubscriber	color/image_raw	Das Farbbild der Kamer mit einer Auflösung von 640x480 Pixeln.
StateSubscriber	state_machine/state	Der aktuelle Zustand der State Machine
SignPublisher	sign_detection/sign	Informationen über ein erkanntes Verkehrszeichen.
StatusInfoPublisher	status/status_info	Statusinformationen für die StateMachine

psaf_start_box ("start_box")

Zu Beginn der Fahrt befindet sich das Fahrzeug in einer geschlossenen Startbox. Am Tor der Startbox befindet sich ein Stop Schild und ein QR-Code. Nachdem sich das Tor geöffnet hat, muss das Fahrzeug die Startbox verlassen. Die **Startbox** Node ist für die Erkennung der Öffnung der Startbox zuständig. Hierfür kann entweder das Stoppschild, der QR-Code oder die Ultraschallsensoren (Tor wird nicht mehr von den Ultraschallsensoren erfasst) genutzt werden. Die Node verfügt über folgende Kommunikationsschnittstellen:

Schnittstelle	Topic Name	Beschreibung
ImageSubscriber	color/image_raw	Das Farbbild der Kamer mit einer Auflösung von 640x480 Pixeln.
UltrasonicSubscriber	uc_bridge<pos>	<pos> beschreibt die Position des Sensors am Fahrzeug
StateSubscriber	state_machine/state	Der aktuelle Zustand der State Machine
StatusInfoPublisher	status/status_info	Statusinformationen für die StateMachine

psaf_state_machine ("state_machine")



Die **StateMachine** Node besitzt ein besonderes Publisher-Schema. Der State wird nicht nur durch die **update()** Methode veröffentlicht, sondern auch bei jedem Zustandswechsel direkt. Auf diese Weise soll ein neuer Zustand schnellstmöglich veröffentlicht werden und nicht erst beim nächsten Tick.

Die **StateMachine** Node ist die zentrale Steuerungseinheit des PSAF. Sie beinhaltet ein Zustandsautomat, dessen Zustände die aktuelle Fahraufgabe repräsentieren. Ein Zustandswechsel wird über die **StatusInfos** ausgelöst, die von den anderen Nodes als Reaktion auf bestimmte Events gesendet werden. So sendet die **LaneDetection** die **StatusInfo PARKING_INTENT**, falls die Startlinie detektiert wurde. Die StateMachine verfügt über Transition Guards, die die Zustandsübergänge verhindern. So kann beispielsweise in der Disziplin "Rundkurs mit Hindernissen" keine Parkplatzsucht durchgeführt werden. Die Festlegung der Disziplin erfolgt über die Knöpfe am Heck des Fahrzeugs. Da am PSAF 1 Auto keine Knöpfe angebracht werden, muss die Information manuell veröffentlicht werden. Hierfür muss auf dem Topic **uc_bridge/button** eine 0 für die Disziplin "Rundkurs mit Einparken" und eine 1 für die Disziplin "Rundkurs mit Hindernissen" gesendet werden.

Mit folgendem Befehl kann manuell eine Disziplin gesetzt werden, wobei der Integer Wert im **data** Feld der Disziplin entspricht:

```
ros2 topic pub /uc_bridge/button std_msgs/msg/Int8 "{data: 1}"
```

Die **StateMachine** Node verfügt über folgende Kommunikationsschnittstellen:

Schnittstelle	Topic Name	Beschreibung
ErrorSubscriber	watchdog/error_message	Informationen über einen (Hardware-)Fehler
StatusInfoSubscriber	status/status_info	Statusinformationen für die StateMachine
ButtonSubscriber	uc_bridge/button	Informationen über den gedrückten Knopf zur Auswahl der Disziplin
SignSubscriber	sign_detection/sign	Informationen über ein erkanntes Verkehrszeichen
StatePublisher	state_machine/state	Informationen über den aktuellen Zustand der StateMachine

psaf_trajectory ("trajectory")

Die **Trajectory** Node ist für die Planung der Trajektorie zuständig. Hierfür empfängt die Node die aktuell erkannten Fahrbahnmarkierungen und Informationen über die erkannten Objekte, die das Fahrzeug beachten und gegebenenfalls umfahren muss. Die Node verfügt über folgende

Kommunikationsschnittstellen:

Schnittstelle	Topic Name	Beschreibung
ObjectSubscriber	object_detection/obstacle	Informationen über die erkannten Objekte
StateSubscriber	state_machine/state	Der aktuelle Zustand der StateMachine
LaneMarkingsSubscriber	lane_detection/lane_markings	Informationen über die erkannten Fahrbahnmarkierungen
TrajectoryPublisher	trajectory/trajectory	Informationen über die Trajektorie
StatusInfoPublisher	status/status_info	Statusinformationen für die StateMachine

psaf_utils ("utils")

Dieses Paket enthält Hilfsfunktionen und -programme. Im Ordner `psaf_sim_controller` befindet sich eine Python ROS Node, um das Fahrzeug in der Simulationsumgebung präziser steuern zu können. Um die Node zu starten, kann der folgende Befehl genutzt werden:

```
ros2 run psaf_utils simulation_controller
```

Anschließend öffnet sich die in nachfolgend gezeigte [GUI](#). Die Topic-Namen können angepasst werden, um das Motorlevel und das Lenkwinkel Topic zu verändern. Die derzeit eingetragenen Namen entsprechen den derzeit verwendeten Topic-Namen in der Simulationsumgebung. Nach Bestätigung der Namen durch den [OK] Knopf, kann das Fahrzeug über W, A, S, D oder die Pfeiltasten gesteuert werden.

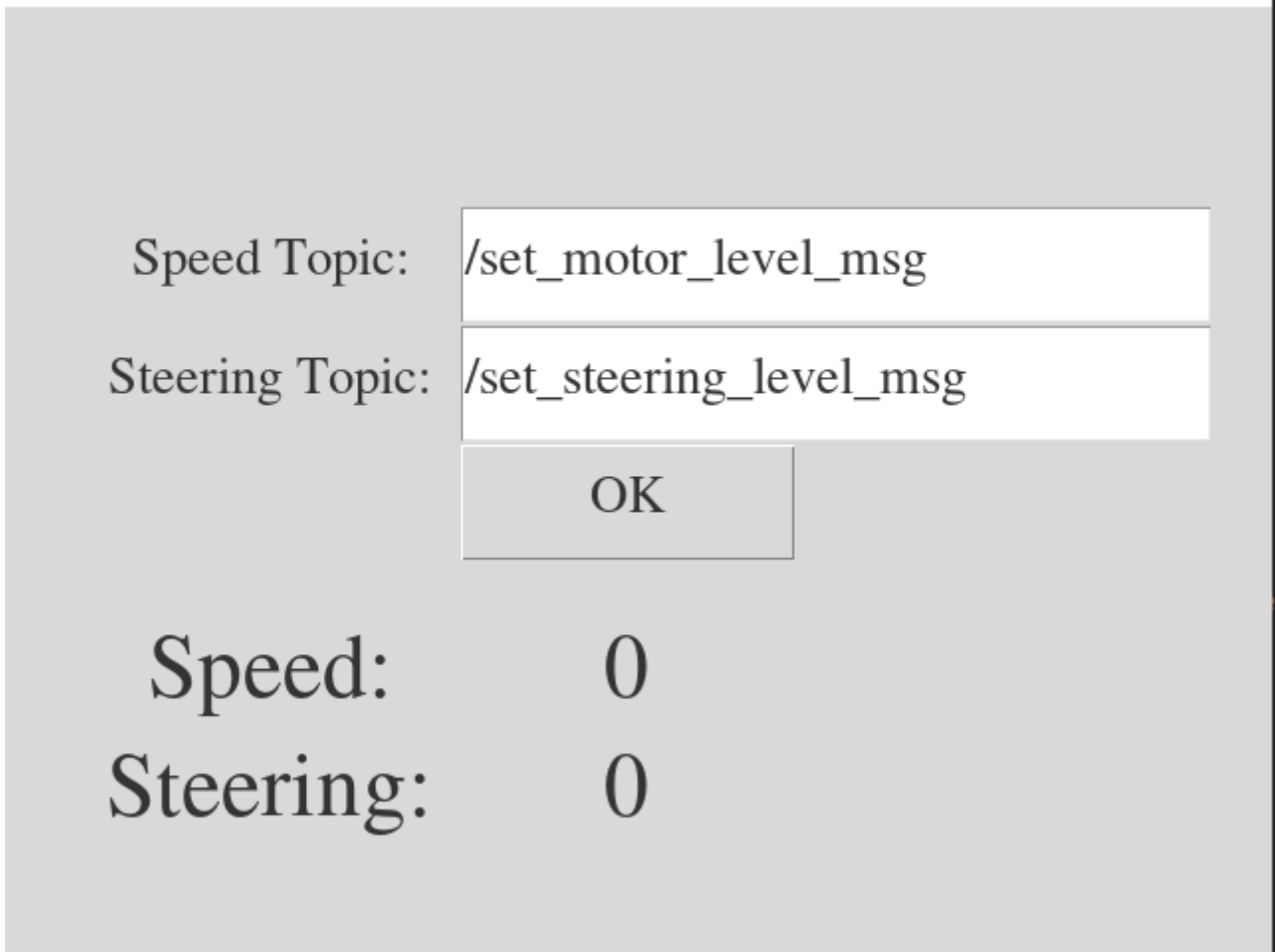


Figure 3. GUI der Fernsteuerung

Im Ordner `src` befindet sich zusätzlich noch die Datei `utils.cpp`. Hier sind verschiedene Methoden für die Generierung von Streckenabschnitten sowie Testimplmentierungen für die Detektion der Spurmarkierungen und mehr zu finden. Diese Datei wird weder gebaut noch getestet.

psaf_watchdog ("watchdog")

Die `Watchdog` Node ist für die Überwachung der Hardware des Fahrzeugs zuständig. Hierfür werden unter anderem die Kamera, Ultraschallsensoren und die zurückgemeldeten Werte für Lenkung und Geschwindigkeit überwacht. Im kritischen Fehlerfall (bspw. Ausfall der Kamera) wird eine Fehlermeldung an die `StateMachine` gesendet. Im Falle eines unkritischen Fehlers (bspw. Ausfall eines nicht genutzten Ultraschallsensors) wird eine Warnung an die `StateMachine` gesendet.

Schnittstelle	Topic Name	Beschreibung
ImageSubscriber	<code>color/image_raw</code> , <code>depth/image_rect_raw</code>	Bildinformationen der Kamera
UltrasonicSubscriber	<code>uc_bridge/<pos></code>	Messwerte der Ultraschallsensoren
SpeedSubscriber	<code>uc_bridge/get_speed</code>	Geschwindigkeitswerte des Fahrzeugs

SteeringSubscriber	uc_bridge/get_steering	Lenkswerte des Fahrzeugs
StateSubscriber	state_machine/state	Der aktuelle Zustand der StateMachine
ErrorPublisher	watchdog/error_message	Fehlermeldungen

Parameter

ROS unterstützt den Einsatz sogenannter Parameter. Die Parameter können während der Laufzeit durch einen Befehl im Terminal verändert werden. Dies kann beispielsweise dafür genutzt werden, um bestimmte Grenzwerte dynamisch während der Laufzeit zu verändern, ohne das Projekt neu bauen zu müssen. Der bereitgestellte Workspace enthält im Paket **Spurerkennung** ein Beispiel, wie die Parameter Funktionalität in einer Node realisiert werden kann.

Im gezeigten Beispiel wird der Parameter `use_secondary_algorithm` genutzt, um zwischen zwei Spurerkennungsalgorithmen zu wechseln. Dies kann für die objektive Bewertung der Algorithmen verwendet werden.

Im ersten Schritt muss der Parameter im Konstruktor deklariert werden.

```
...
this->declare_parameter("use_secondary_algorithm", false);
...
```

Zur Abfrage des Wertes muss die folgende Zeile in den Code eingefügt werden:

```
...
bool use_secondary_algorithm = this->get_parameter("use_secondary_algorithm")
.as_bool();
...
```

Das Typcasting am Ende ist notwendig, damit der Parameter als bool interpretiert wird. Falls ein anderer Datentyp als bool verwendet werden soll, muss entsprechend das passende Typcasting eingefügt werden.

Um den Parameter während der Laufzeit zu verändern wird der folgende Befehl verwendet:

```
ros2 param set <class_name> <parameter_name> <value>
```

Am Beispiel der **LaneDetectionNode** lautet der Befehl wie folgt:

```
ros2 param set /lane_detection use_secondary_algorithm true
```

Mehr Informationen über die Verwendung von Parametern kann der [ROS2 Dokumentation](#)

entnommen werden.

Nachrichten

Dieses Kapitel beschäftigt sich mit den in diesem Workspace verwendeten Nachrichten. Um eine Nachricht zu empfangen, verfügen die Pakete über entsprechende Subscriber. Um eine Nachricht zu senden, muss die Publisher Methode aufgerufen werden. Dies sollte optimalerweise in der `update()` Methode des jeweiligen Paketes erfolgen. Die `update` Methode wird periodisch entsprechend der in der `config.json` Datei definierten Frequenz aufgerufen.

color/image_raw und depth/image_rect_raw

Beide Nachrichten werden von der Realsense Kamera ausgesendet. Bei der `color/image_raw` Nachricht handelt es sich um ein RGB Farbbild mit einer Auflösung von 640x480 Pixeln. Bei der `depth/image_rect_raw` Nachricht handelt es sich um das Tiefenbild mit einer Auflösung von 1280x720 Pixeln. Die Nachrichten werden von der `libpsaf` intern vom ursprünglichen ROS Format `sensor_msgs::msg::Image` in das OpenCV Bildformat `cv::Mat` umgewandelt. Die Nachrichten sind in einem Vektor gespeichert. Der Zugriff erfolgt über die Position der Nachricht im Vektor über folgende Methode:

```
void processImage(cv::Mat & img, int sensor)
```

Der Parameter `sensor` gibt hierbei an, welches Bild genutzt werden soll. Die Werte sind standardmäßig 0 für das RGB Bild und 1 für das Tiefenbild.



Nicht jedes Paket kann beide Nachrichten empfangen. Die Spurerkennung kann nur das RGB Farbbild empfangen.

lane_detection/lane_markings

Diese Nachricht wird zum Versenden der erkannten Spurmarkierungen verwendet. Die Nachricht hat den Typ `libpsaf_msgs::msg::LaneMarkings`. Sie besteht intern aus 3 Vektoren, welche die Punkte für die Linke, Mittlere und Rechte Spurmarkierung enthalten sowie weiteren Parametern für zusätzliche Informationen. Eine genaue Definition ist in der Dokumentation der `libpsaf` zu finden.

Publisher

Die Nachricht wird über den `LaneMarkingsPublisher` versendet. Um die Nachricht zu veröffentlichen, stehen mehrere Funktionen zur Verfügung. Es kann sowohl direkt die Nachricht übergeben werden als auch die einzelnen Parameter. Falls die einzelnen Parameter übergeben werden, wird vom Publisher daraus intern eine Nachricht erzeugt und diese veröffentlicht.

```
void publishLaneMarkings(libpsaf_msgs::msg::LaneMarkings & laneMarkings);
```

```
void publishLaneMarkings(  
    std::vector<geometry_msgs::msg::Point> leftLane,  
    std::vector<geometry_msgs::msg::Point> centerLane,  
    std::vector<geometry_msgs::msg::Point> rightLane,  
    bool has_blocked_area, bool no_overtaking, int side);
```

```
void publishLaneMarkings(  
    std::vector<cv::Point> leftLane,  
    std::vector<cv::Point> centerLane,  
    std::vector<cv::Point> rightLane,  
    bool has_blocked_area, bool no_overtaking, int side);
```

Subscriber

Um `LaneMarkings` Nachrichten zu empfangen, steht die folgende Funktion zur Verfügung.

```
void processLaneMarkings(const libpsaf_msgs::msg::LaneMarkings::ConstPtr & p);
```

lane_detection/stop_line

Diese Nachricht enthält Informationen über eine Haltelinie und hat den Typ `libpsaf_msgs::msg::StopLine`. Sie verfügt intern über einen Parameter `stop_line_detected`, welcher angibt, ob eine Haltelinie erkannt wurde. Des Weiteren ist der Typ der Haltelinie (in eigener Fahrspur oder nicht) und die Position der Stopplinie als `geometry_msgs::msg::Point` enthalten. Der Punkt ist immer in der Mitte der Haltelinie.



Im Carolo Cup muss diese Nachricht nur in der zweiten Disziplin verwendet werden. In Disziplin 1 (Rundkurs mit Einparken) muss NICHT auf Stopplinien reagiert werden.

Publisher

Um `StopLine` Nachrichten zu veröffentlichen stehen die folgenden Funktionen zur Verfügung.

```
void publishStopLine(libpsaf_msgs::msg::StopLine & stopLine);
```

```
void publishStopLine(bool stop_line_detected, unsigned int type, int x, int y);
```

Subscriber

Um Nachrichten vom Typ `libpsaf_msgs::msg::StopLine` zu empfangen steht die folgende Funktion

zur Verfügung.

```
void processStopLine(const libpsaf_msgs::msg::StopLine::ConstPtr & p);
```

object_detection/obstacle

Diese Nachricht enthält Informationen über gefundene Objekte. Die Nachricht repräsentiert genau ein Objekt. Falls mehrere Objekte gefunden wurden, müssen diese separat veröffentlicht werden. Die Nachricht hat den Typ `libpsaf_msgs::msg::Obstacle`. Sie enthält unter anderem den Typ, die Position, die Distanz und die Geschwindigkeit des Objektes. Eine genaue Definition ist in der Dokumentation der `libpsaf` zu finden.

Publisher

Um `Obstacle` Nachrichten zu veröffentlichen, stehen ähnliche Methoden wie bei `LaneMarkings` zur Verfügung. So kann entweder direkt eine `libpsaf_msgs::msg::Obstacle` Nachricht veröffentlicht werden oder die einzelnen Parameter übergeben werden, welche intern in einer `libpsaf_msgs::msg::Obstacle` Nachricht verpackt werden. Hierbei ist zu beachten, dass bei der zweiten Methode die übrigen Parameter nicht gesetzt werden.

```
void publishObstacle(libpsaf_msgs::msg::Obstacle & obstacle);

void publishObstacle(
    geometry_msgs::msg::Vector3 & position, geometry_msgs::msg::Vector3 & size);

void publiscObstacle(
    double posX, double posY, double posZ, double sizeX, double sizeY, double sizeZ,
    int id, int type, int pos_in_scene, int velocity, int distance, int state);
```

Subscriber

Um Nachrichten vom Typ `libpsaf_msgs::msg::Obstacle` zu empfangen, steht die folgende Funktion zur Verfügung.

```
void processObstacle(const libpsaf_msgs::msg::Obstacle::ConstPtr & p);
```

parking_detection/parking_spot

Diese Nachricht enthält Informationen über die Position und den Typ (parallel, rechtwinklig) eines detektierten Parkplatzes. Die Nachricht hat den Typ `libpsaf_msgs::msg::ParkingSpot`. Für weitere Informationen siehe die Dokumentation der `libpsaf`.



Im Carolo Cup muss diese Nachricht nur in der ersten Disziplin verwendet werden. In Disziplin 2 (Rundkurs mit Hindernissen) muss NICHT eingeparkt werden.

Publisher

Zur Veröffentlichung der Nachricht steht die folgende Funktion zur Verfügung.

```
void publishParkingSpot(libpsaf_msgs::msg::ParkingSpot & parkingSpot);

void publishParkingSpot(
    unsigned int type, geometry_msgs::msg::Point position);
```

Subscriber

Um Nachrichten vom Typ `libpsaf_msgs::msg::ParkingSpot` zu empfangen, steht die folgende Funktion zur Verfügung.

```
void processParkingSpot(const libpsaf_msgs::msg::ParkingSpot::ConstPtr & p);
```

sign_detection/sign

Diese Nachricht enthält Informationen über ein erkanntes Schild. Sie ist vom Typ `libpsaf_msgs::msg::Sign`. Für weitere Informationen siehe die Dokumentation der [libpsaf](#).



Im Carolo Cup muss nur im MasterCup auf Schilder reagiert werden. Im BasisCup sind keine Verkehrszeichen vorhanden.

Die `Sign` Nachricht enthält Informationen über die Position des Schildes sowie den Typ.

Publisher

Es stehen drei Methoden zur Verfügung, um die `Sign` Nachricht zu veröffentlichen. Falls nicht direkt die `libpsaf_msgs::msg::Sign` Nachricht verwendet werden soll, können die Parameter übergeben werden. Der Publisher baut die `libpsaf_msgs::msg::Sign` Nachricht dann aus den Parametern auf.

```
void publishSign(libpsaf_msgs::msg::Sign & sign);

void publishSign(
    geometry_msgs::msg::Point position, int type);

void publishSign(double x, double y, double z, int type);
```

Subscriber

Die Callbackmethode um **Sign** Nachrichten zu empfangen, lautet:

```
void processSign(const libpsaf_msgs::msg::Sign::ConstPtr & sign);
```

state_machine/state

Die **State** Nachricht enthält Informationen über den aktuellen Zustand der **StateMachine**. Die Nachricht ist als **enum** kodiert. Der Zustand sollte genutzt werden, um bestimmte Funktionen innerhalb der anderen Pakete zu aktivieren oder zu deaktivieren. Beispielsweise sollte die **ParkingSpotDetection** nur aktiv sein, wenn der Zustandsautomat im Zustand **PR_SEARCH** ist.

Publisher

Der Publisher kommt nur in der **StateMachine** zum Einsatz. Die Methode lautet:

```
void publishState(int state);
```

Subscriber

Alle anderen Pakete besitzen einen Subscriber für die **State** Nachricht. Die Callback Methode lautet:

```
void updateState(const std_msgs::msg::Int64::ConstPtr & p);
```

status/status_info

Die **StatusInfo** Nachricht wird genutzt, um die **StateMachine** über bestimmte Ereignisse, beispielsweise das Überfahren der Startlinie, zu informieren. Die **StateMachine** löst, sofern es der aktuelle Zustand zulässt, einen Zustandswechsel aus. Hierfür besitzt die **StateMachine** Transition Guards, sodass unpassende **StatusInfos** (bspw. der Wunsch einen Parkplatz zu suchen während man an einer Haltelinie steht) ignoriert werden. Optimalerweise sollten die anderen Pakete jedoch so implementiert sein, dass falsche **StatusInfos** erst gar nicht veröffentlicht werden. Die Nachricht hat den Typ 'libpsaf_msgs::msg::StatusInfo'. Für weitere Informationen siehe die Dokumentation der **libpsaf**.

Publisher

Der Publisher für die **StatusInfo** lautet:

```
void StatusInfoPublisher::publishStatus(libpsaf_msgs::msg::StatusInfo msg);
```

Subscriber

Die Callbackmethode für die **StatusInfo** lautet:

```
void processStatus(libpsaf_msgs::msg::StatusInfo::SharedPtr status)
```

Aufgrund der Wichtigkeit der StatusInfos sind die möglichen StatusInfos nachfolgend aufgelistet:

```
# Status Codes
uint8 STARTBOX_OPEN = 0           # Detected opening of the start boxST
uint8 UPHILL_START = 1            # Beginn of uphill driving
uint8 DOWNHILL_START = 2          # Beginn of Downhill driving = end of uphill
uint8 DOWNHILL_END = 3            # end of downhill driving
uint8 STOP_LINE_APPROACH = 4      # Stop-line detected
uint8 STOP_LINE_REACHED = 5      # Stop_line reached
uint8 CONTINUE_NO_OBJECT = 6      # Resume drive after wait time, no object
uint8 WAIT_FOR_OBJECT = 7         # Wait for Object at Intersection
uint8 OBJECT_DETECTED = 8         # Object detected
uint8 PARKING_INTENT = 9           # Parking intent = Startline detected
uint8 PARKING_TIMEOUT = 10        # No parking spot found
uint8 PARALLEL_FOUND = 11         # Parallel Parking Spot found
uint8 PERPENDICULAR_FOUND = 12    # Perpendicular parking spot found
uint8 PARKING_FINISHED = 13       # Successfully parked in the spot
uint8 PARKING_FAILED = 14         # Parking failed
uint8 PARK_TIME_REACHED = 15      # Parked, wait for time to pass
uint8 BACK_ON_LANE = 16           # Returned to lane
uint8 STATIC_OBSTACLE = 17        # Static obstacle detected in front of car
uint8 DYNAMIC_OBSTACLE = 18       # Dynamic obstacle in lane detected
uint8 OVERTAKE_POSSIBLE = 19      # Overtaking possibility detected
uint8 PASSED_OBSTACLE = 20        # Obstacle has been passed
uint8 OVERTAKE_FINISHED = 21      # Returned to correct lane
uint8 OVERTAKE_ABORT = 22         # Abort Overtaking attempt
uint8 MANUAL_MODE_ENTER = 23      # Enter Manual Mode
uint8 MANUAL_MODE_EXIT = 24       # Exit Manual Mode
uint8 WATCHDOG_TIMEOUT = 25      # Timeout issued by the watchdog
```

trajectory/trajectory

Diese Nachricht enthält Informationen über die aktuelle Trajektorie. Sie ist vom Typ 'libpsaf_msgs::msg::Trajectory'. Für weitere Informationen siehe die Dokumentation der [libpsaf](#). Die Trajectory Nachricht selbst enthält die Punkte der Trajektorie, die das Auto abzufahren hat.

Publisher

Die Publisher Methoden für die **Trajectory** Nachricht lautet:

```
void publishTrajectory(libpsaf_msgs::msg::Trajectory & trajectory);  
  
void publishTrajectory(std::vector<geometry_msgs::msg::Point> points)
```

Subscriber

Die Callbackmethode für die **Trajectory** lautet:

```
void processTrajectory(libpsaf_msgs::msg::Trajectory::SharedPtr p);
```

watchdog/error_message

Die Watchdog Message wird genutzt, um die StateMachine über Fehler im System zu informieren. Sie hat den Typ **libpsaf_msgs::msg::Error**. Hierbei wird zwischen 3 Levels unterschieden:

1. FINE = 0 : Alles in Ordnung
2. WARNING = 1 : Warnung bei nicht kritischem Fehler
3. ERROR = 2 : Fehler im System, beispielsweise ein Ausfall der Kamera

Publisher

Die Publisher Methode für die **watchdog/error_message** Nachricht lautet:

```
void publishErrorMessage(libpsaf_msgs::msg::Error & errorMessage);  
  
void publishErrorMessage(int type, std::string info_text);
```

uc_bridge Nachrichten

Die nachfolgenden Nachrichten stehen alle in Verbindung mit der UC Bridge. Hierbei handelt es sich um Nachrichten, die Messwerte der Sensoren beinhalten oder Steuerungsbefehle für das Fahrzeug. Diese Art der Nachrichten hat im Workspace entweder nur einen Publisher(falls die Nachricht von der **uc_bridge** empfangen wird) oder nur einen Subscriber(falls die Nachricht von der **uc_bridge** gesendet wird). Mehr Informationen über die UC Bridge können [hier](#) gefunden werden.

uc_bridge/button

Die Button-Nachricht wird von der uc_bridge ausgesendet. Der Default-Wert ist -1. Sobald ein Knopf gedrückt wird, wird der Wert auf den Button-Code gesetzt und bleibt auf diesem Wert, auch wenn der Knopf nicht mehr gedrückt wird. Über den Wert des Knopfes wird die Disziplin ausgewählt.



Beim PSAF 1 Auto sind keine Knöpfe vorhanden. Der Knopfdruck muss entsprechend simuliert werden. Beispielsweise durch manuelles Publishen auf dem Topic. Ohne Knopfdruck wird die StateMachine den Zustand **STARTBOX** nicht verlassen.

Subscriber

Die Subscriber Methode für die **uc_bridge/button** Nachricht lautet:

```
void processButton(std_msgs::msg::Int8::SharedPtr p);
```

uc_bridge/get_speed

Diese Nachricht enthält die vom Fahrzeug gemessene Geschwindigkeit in cm/s. Beim PSAF 1 Auto wird die Geschwindigkeit über einen Hallsensor am linken hinteren Rad gemessen, beim PSAF 2 Fahrzeug wird die Geschwindigkeit über einen Inkrementalgeber an der Kardanwelle gemessen.

Subscriber

Die Subscriber Methode für die **uc_bridge/get_speed** Nachricht lautet:

```
void updateSpeed(std_msgs::msg::Int16::SharedPtr p);
```

uc_bridge/get_steering

Diese Nachricht enthält den vom Fahrzeug gemessenen Lenkwinkel in 1/10 Grad.



Die Fahrzeuge sind derzeit nicht in der Lage den Lenkwinkel zu messen. Deswegen wird stattdessen der angefragte Lenkwinkel zurückgesendet.

Subscriber

Die Subscriber Methode für die **uc_bridge/get_steering** Nachricht lautet:

```
void updateSteering(std_msgs::msg::Int16::SharedPtr p);
```

uc_bridge/light

Diese Nachricht wird genutzt, um die Lichter am PSAF 2 Auto zu steuern. Beispielsweise können so die Bremslichter, Blinker und Warnblinker an- und ausgeschaltet werden. Beim PSAF 1 Auto wird diese Nachricht ignoriert, da das Fahrzeug keine Lichter hat. Eine ausführliche Dokumentation inkl. der Zuweisung der Lichter kann in der Dokumentation der [libpsaf](#) gefunden werden.

Publisher

Die Publisher Methode für die `uc_bridge/light` Nachricht lautet:

```
void publishLight(int light)

void publishLight(std_msgs::msg::Int8 light)
```

`uc_bridge/manual_signals`

Diese Nachricht wird von der `uc_bridge` gesendet, wenn das Fahrzeug in manuellen Modus ist. Manuelles Fahren erfolgt durch eine Fernsteuerung und wird nur vom PSAF 2 Auto unterstützt.

Subscriber

Die Subscriber Methode für die `uc_bridge/manual_signals` Nachricht lautet:

```
void processManualSignals(std_msgs::msg::UInt8::SharedPtr p);
```

`uc_bridge/set_motor_level`

Diese Nachricht ist ein Sonderfall. Beim Erzeugen des Publishers wird der erste Teil des Topic-Namen `set_motor_level` angegeben. Der Publisher baut aus diesem die beiden Topics `uc_bridge/set_motor_level_forward` und `uc_bridge/set_motor_level_backward` auf. Die Publisher Methode erwartet Geschwindigkeiten zwischen -200 und 200 cm/s. Intern wird diese Geschwindigkeit in das Motorlevel umgerechnet und je nach Fahrtrichtung auf dem entsprechenden Topic gesendet.

Publisher

Die Publisher Methode für die `uc_bridge/set_motor_level` Nachricht lautet:

```
void publishSpeed(int speed)
```

`uc_bridge/set_steering`

Diese Nachricht wird genutzt, um den Lenkwinkel des Fahrzeugs zu setzen. Hierbei können die Werte entweder als 1/10 Grad oder als 1/100 rad angegeben werden. Eine entsprechende Flag in der Methode gibt an, in welchem Format die Werte übergeben werden. Die Umrechnung auf einen Lenkwinkel erfolgt in der `uc_bridge`. Es ist zu beachten, dass das PSAF 1 einen maximalen Lenkwinkel von -30 bis 30 Grad hat, das PSAF 2 einen maximalen Lenkwinkel von - 45 bis 45 Grad. Negative Werte bewirken ein Lenkausschlag in Fahrtrichtung rechts.

Publisher

Die Publisher Methode für die `uc_bridge/set_steering` Nachricht lautet:

```
void publishSteering(int value, bool rad)
```

`uc_bridge/us_<pos>`

Diese Nachricht wird genutzt, um die Messwerte der Ultraschallsensoren zu empfangen. Der Zusatz `<pos>` gibt an, wo sich der Sensor am Fahrzeug befindet. Die Messwerte liegen in cm vor.

Mögliche Position am PSAF 1 Auto sind:

- `us_front_center`
- `us_mid_right`
- `us_mid_left`

Beim PSAF 2 Auto sind zusätzlich die folgenden Sensoren verfügbar:

- `us_front_left`
- `us_front_right`
- `us_rear_left`
- `us_rear_right`
- `us_rear_center`

Subscriber

Die Subscriber Methode für die `uc_bridge/us_<pos>` ist nachfolgend aufgeführt. `int sensor` gibt hierbei die Position des Topics im Topic Vector an. Die Definition hiervon ist im Paket `psaf_configuration` enthalten.

```
void updateSensorValue(sensor_msgs::msg::Range::SharedPtr p, int sensor)
```

State Machine

Der Zustandsautomat ist das Herzstück des Workspace. Durch diesen wird vorgegeben, welche Fahraufgabe durchzuführen ist. Alle anderen Nodes bekommen den aktuellen Zustand mitgeteilt und können damit entscheiden, ob derzeit eine Aktion ausgeführt werden soll. Der Zustandsautomat ist in der Abbildung [Zustandsautomat](#) dargestellt.

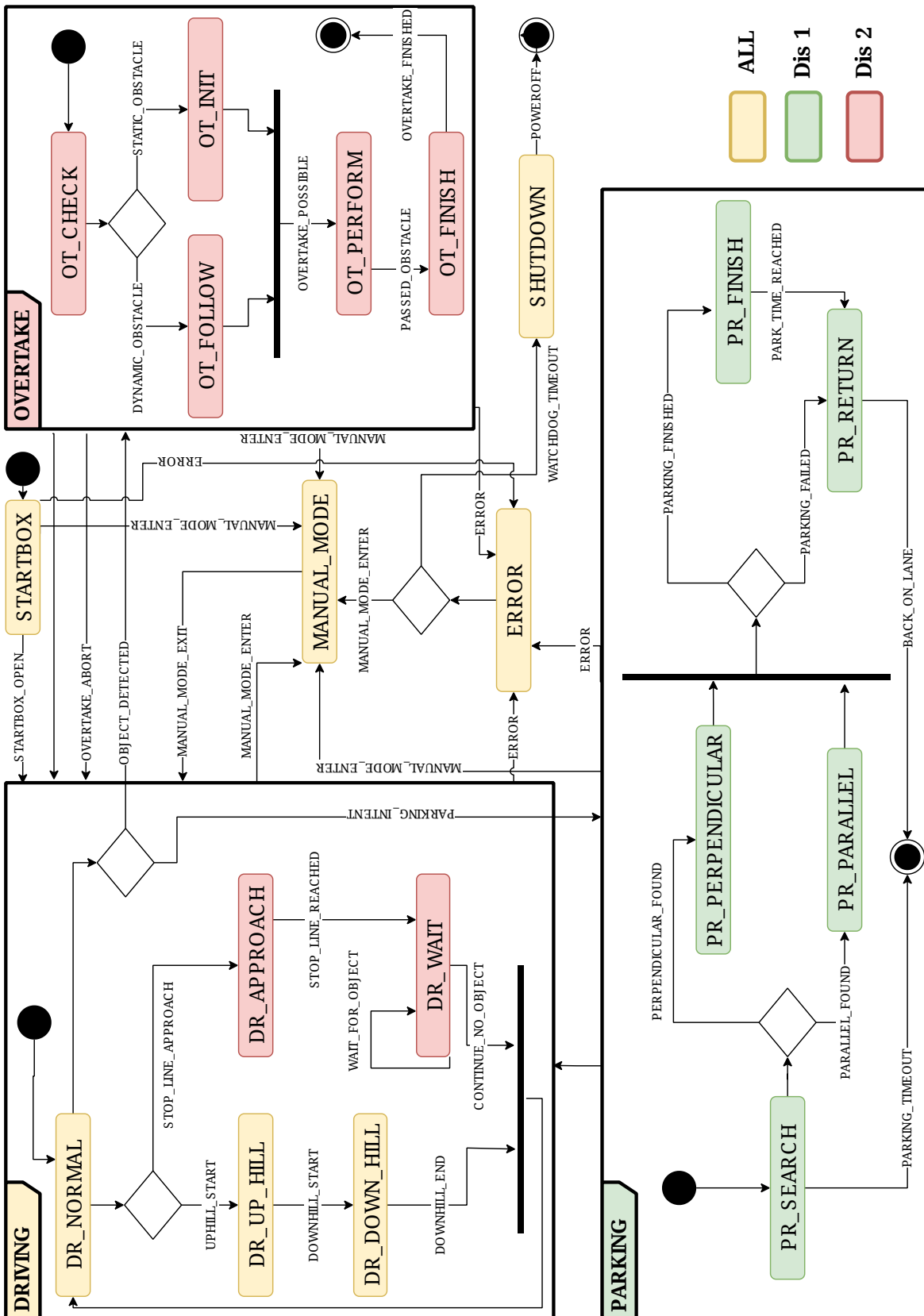


Figure 4. Der Zustandsautomat im Start-Workspace

Zustandsübergänge werden über **StatusInfos** ausgelöst. Diese werden über Nachrichten vom Typ **libpsaf_msgs::msg::StatusInfo** auf dem Topic **status/status_info** übermittelt. Alle Nodes, die einen Zustandswechsel auslösen können, verfügen über einen StatusInfo-Publisher. Die möglichen Zustandsübergänge sind in der **libpsaf** definiert und nachfolgend aufgelistet.

uint8 STARTBOX_OPEN = 0	# Detected opening of the start box
uint8 UPHILL_START = 1	# Beginn of uphill driving
uint8 DOWNHILL_START = 2	# Beginn of Downhill driving
uint8 DOWNHILL_END = 3	# end of downhill driving
uint8 STOP_LINE_APPROACH = 4	# Stop-line detected
uint8 STOP_LINE_REACHED = 5	# Stop_line reached
uint8 CONTINUE_NO_OBJECT = 6	# Resume drive after wait time, no object
uint8 WAIT_FOR_OBJECT = 7	# Wait for Object at Intersection
uint8 OBJECT_DETECTED = 8	# Object detected
uint8 PARKING_INTENT = 9	# Parking intent = Startline detected
uint8 PARKING_TIMEOUT = 10	# No parking spot found
uint8 PARALLEL_FOUND = 11	# Parallel Parking Spot found
uint8 PERPENDICULAR_FOUND = 12	# Perpendicular parking spot found
uint8 PARKING_FINISHED = 13	# Successfully parked in the spot
uint8 PARKING_FAILED = 14	# Parking failed
uint8 PARK_TIME_REACHED = 15	# Parked, wait for time to pass
uint8 BACK_ON_LANE = 16	# Returned to lane
uint8 STATIC_OBSTACLE = 17	# Static obstacle detected in front of car
uint8 DYNAMIC_OBSTACLE = 18	# Dynamic obstacle in lane detected
uint8 OVERTAKE_POSSIBLE = 19	# Overtaking possibility detected
uint8 PASSED_OBSTACLE = 20	# Obstacle has been passed
uint8 OVERTAKE_FINISHED = 21	# Returned to correct lane
uint8 OVERTAKE_ABORT = 22	# Abort Overtaking attempt
uint8 MANUAL_MODE_ENTER = 23	# Enter Manual Mode
uint8 MANUAL_MODE_EXIT = 24	# Exit Manual Mode
uint8 WATCHDOG_TIMEOUT = 25	# Timeout issued by the watchdog

Zum Testen kann manuell eine StatusInfo veröffentlicht werden. Hierfür kann die folgende Nachricht genutzt werden.

```
ros2 topic pub status_info libpsaf_msgs/msg/StatusInfo "type: 0"
```



Für die korrekte Funktionsweise muss eine Disziplin gesetzt werden. Auf dem Wettkampfauto erfolgt dies durch Drücken des entsprechenden Knopfes am Heck des Autos. Beim PSAF 1 Auto oder um die StateMachine zu testen, muss der Knopfdruck simuliert werden. Dies erfolgt, indem man manuell eine Nachricht auf dem **ButtonPublisher** veröffentlicht. Ohne das Setzen der Disziplin kann der Zustandsautomat nicht in die Subautomaten **PARKING** und **OVERTAKE** wechseln.

```
ros2 topic pub /uc_bridge/button std_msgs::msg::Int8 "data: 0"
```



Zur Auswahl der Disziplin 1 muss eine **1** anstatt der **0** veröffentlicht werden.

Die Zustände des **Zustandsautomaten** sind wie folgt kodiert:

```

/**
 * @file state_definitions.hpp
 * @brief The state definitions for the state machine
 * @author PSAF
 * @date 2022-06-01
 */
#ifndef PSAF_STATE_MACHINE__STATE_DEFINITIONS_HPP_
#define PSAF_STATE_MACHINE__STATE_DEFINITIONS_HPP_

/**
 * @enum STATE
 * @brief The state definitions for the state machine
 */
enum STATE
{
    STARTBOX,           // 0 Initial State, Car in Startbox
    ERROR,              // 1 Error State, Critical error occurred. Stop car
    MANUAL_MODE,        // 2 Manual driving via remote control
    STOP,               // 3 Stop in case of error - not used yet
    SHUTDOWN,           // 4 Timeout after error, shutting down car. Recovery impossible

    // Overtaking = Obstacle evasion
    OT_CHECK,           // 5 Check if overtaking is possible
    OT_FOLLOW,          // 6 Follow dynamic obstacle
    OT_INIT,            // 7 Initiate the lanechange
    OT_PERFORM,         // 8 Perform the lanechange
    OT_FINISH,          // 9 Finish the lanechange

    // Driving
    DR_NORMAL,          // 10 Driving in the right lane
    DR_UP_HILL,         // 11 Driving UP Hill
    DR_DOWN_HILL,       // 12 Driving DOWN Hill
    DR_APPROACH,        // 13 Approach an intersection
    DR_WAIT,            // 14 Wait at an Intersection

    // Parking
    PR_SEARCH,          // 15 Search for parking spot
    PR_PARALLEL,        // 16 Parallel Park
    PR_PERPENDICULAR,   // 17 Perpendicular park
    PR_FINISH,          // 18 Finish Parking and wait for xx s
    PR_RETURN           // 19 Leave the parking spot and resume driving
};

#endif // PSAF_STATE_MACHINE__STATE_DEFINITIONS_HPP_

```

Der Zustandsautomat besitzt Transition-Guards, die überprüfen, ob eine Transition ausgeführt werden darf. Falls dies nicht der Fall ist, so wird kein Zustandswechsel ausgelöst. So kann der Subautomat **Overtaking** in Disziplin 1 nie betreten werden, da Überholen nur in Disziplin 2 erforderlich ist.

Testen

Ein essenzieller Bestandteil des Entwicklerprozesses ist das Testen. ROS2 bietet hierfür eine gute Unterstützung. Zu den Tests gehören neben Codestyle- und Code-Konformitätsprüfungen auch individuelle Unit-, Integration- und Simulationstests. Diese werden nachfolgend genauer erklärt.

Beispiele für ROS2 spezifische Tests können im [rclcpp Repo](#) gefunden werden.

Jedes Paket besitzt bereits einen Dummy-Unittest-File. Für das Hinzufügen weiterer Tests muss die CMake-Datei angepasst werden. Hierbei müssen auch die nötigen Abhängigkeiten angegeben werden. Als Beispiele, wie die CMake-Datei anzupassen ist, können die CMake-Dateien in den Paketen `psaf_lanedetection`, `psaf_state_machine` und `psaf_startbox` genutzt werden.

Nach der Erklärung erfolgt eine Auflistung der bereits vorhandenen Tests pro Paket.



Manche Tests können zum Bestehen gezwungen werden. Dies ist über die Flag `FORCE_TEST_PASS` in der Datei `configuration.hpp` im Paket `psaf_configuration` möglich. Falls die Flag auf `true` gesetzt ist, wird im Testfall anstatt der tatsächlichen Testbedingung die Bedingung `ASSERT_TRUE(true)` evaluiert. Diese Funktion kann auch während der Entwicklung genutzt werden, um ein Fehlschlagen von bereits implementierten Testfällen zu verhindern. Für die Abgabe muss diese Flag auf `false` gesetzt werden.

Codestyle- und Code-Konformitätsprüfungen

ROS2 Foxy nutzt die Google Codstyle-Standards. Diese sind in der offiziellen [ROS2-Dokumentation](#) zu finden. Um die Einhaltung zu Testen, wird zum einen `uncrustify`, zum anderen `cpplint` verwendet. `ament` bietet eine Möglichkeit, den Code automatisch nach den `uncrustify` Regeln zu formatieren. Hierfür muss im `src/` Ordner des Workspaces oder eines einzelnen Pakets folgender Befehl ausgeführt werden:

```
ament_uncrustify --reformat
```

Unit-Tests

Für die Unit-Tests wird das `googletest`-Framework verwendet. Die Tests werden im Ordner `test/` für jedes Paket separat erstellt. Ziel sollte es sein, jede Methode mindestens einmal auszuführen. Falls es innerhalb einer Methode mehrere Pfade gibt, so müssen diese alle getestet werden.

Integration-Tests

Die Integrationstests sollen die Interaktion zwischen den einzelnen Teilen des Systems testen. Insbesondere soll die Reaktion auf ankommende Nachrichten überprüft werden. Hierfür müssen "Dummy"-Nachrichten erstellt werden. In der Datei `test/include/test_util.hpp` ist eine Klasse definiert, mit der beliebige Nachrichten empfangen und gesendet werden können. Bei den

Integrationstests sollte der Fokus auf dem Testen der extremen Fälle liegen. So sollte bei Bildverarbeitenden Nodes mindestens die Reaktion auf leere Bilder, auf Bilder mit ungültigen Daten und auf Bilder mit ungültigen Dimensionen getestet werden.

Simulationstests

Für die Simulationstests wird die bereits vorhandene [Simulationsumgebung](#) verwendet. Die Simulation unterstützt sowohl das PSAF 1 als auch das PSAF 2 Fahrzeug. Sofern möglich, sollen spezielle Szenarien erstellt werden. Das Szenario wird als ROS Bag-Datei gespeichert. Um ein ROS Bag aufzunehmen, muss der folgende Befehl ausgeführt werden, während das Szenario läuft:

```
# Um ein Topic aufzunehmen  
ros2 bag -o <bagname> record <topic_name>
```

```
#Um mehrere Topics aufzunehmen  
ros2 bag record -o <bagname> <topic_name_1> <topic_name_2> <topic_name_3>
```

Mehr Informationen zur ROS Bags sind der offiziellen [Dokumentation](#) zu entnehmen.

Bei den ROS Bags ist darauf zu achten, dass diese am besten nur kurze Sequenzen enthalten und ein bestimmtes Szenario testen (Beispielsweise eine Fahrt auf gerader Strecke oder eine Kurve).



Die Steuerung des Modellfahrzeugs wird in der Simulationsumgebung selbst nur sehr rudimentär unterstützt. Es wird nur ein Geschwindigkeitswert für die Vorwärts- und Rückwärtsbewegung des Modellfahrzeugs verwendet. Das Lenken ist nur mit vollständigem Lenkausschlag nach rechts und links oder mit keinem Lenkausschlag möglich. Um eine präzisere Steuerung zu ermöglichen, kann der im Paket `psaf_utils` enthaltene `Controller` verwendet werden.

CI-Pipeline

CI (Continuous Integration) ist ein Prozess, der die Builds und Tests nach jedem Commit ins GitLab automatisch ausführt. Die CI-Pipeline ist in der Datei `.gitlab-ci.yml` definiert. Der Dateiname darf nicht verändert werden, da Gitlab diese ansonsten nicht ausführen kann. Die Pipeline in diesem Workspace besteht aus 5 Schritten:

1. Build - Bauen des Workspaces
2. Test - automatisiertes Testen
3. Analysis - Analyse des Codes und Berechnung der Code Coverage
4. Documentation - Erstellen der Dokumentation: [Doxygen](#) und [Asciidoc](#).
5. Release - Erstellen eines Releases (nur bei Commit mit Tag)

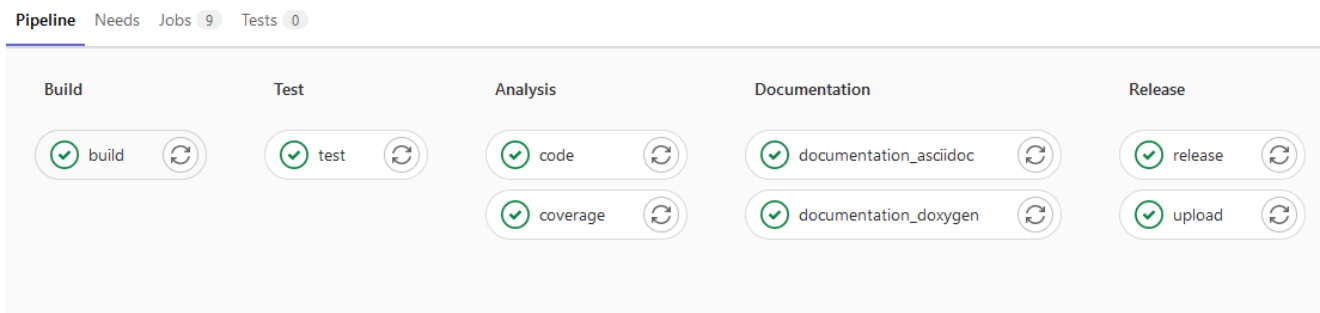


Figure 5. CI-Pipeline



Es dürfen nie zwei Pipelines gleichzeitig gestartet werden, beispielsweise in verschiedenen Branches. Dies führt zu ungewollten Interaktionen zwischen den einzelnen Pipelines und somit zu einem Fehlschlagen der Tests. Aus dem gleichen Grund werden die Tests in der CI-Pipeline nicht parallel ausgeführt.



Die Integrationstests des Zustandsautomaten schlagen in der Pipeline aufgrund von ungeklärten Ursachen teilweise fehl. Deshalb wird jeder Testfall bis zu 10x wiederholt.

Spätestens zum Ende des Projektes muss die Pipeline erfolgreich durchlaufen. Optimalerweise sollte nach jeder Codeänderung ein Commit durchgeführt werden. Falls die Pipeline fehlschlägt, sollte der Fehler sofort behoben werden und erst danach mit der Entwicklung fortgefahren werden. Die Flag `FORCE_TEST_TRUE` in der Datei `configuration.hpp` muss spätestens bei Abgabe auf `false` geändert werden.

Ausführen der Tests

Die Tests werden automatisch in der CI - Pipeline ausgeführt. Um lokal zu überprüfen, ob die Tests durchlaufen, kann folgender Befehle im Wurzelverzeichnis des Workspace ausgeführt werden:

1. Bauen des Projektes:

```
# Bauen des gesamten Projektes
colcon build --symlink-install
```

```
# Bauen eines einzelnen Paketes
colcon build --packages-select <package_name> --symlink-install
```

2. Testen des gesamten Projektes:

```
# Testen des gesamten Projektes
colcon test
```

```
# Testen eines einzelnen Paketes
colcon test --packages-select <package_name>
```

```
# Testen mit Ausgabe der Ergebnisse
colcon test --packages-select <package_name> --event-handlers console_direct+
```

3. Berechnen der Code Coverage:

```
# Builden mit cmake flags
colcon build --packages-select <package_name> --cmake-args -DCMAKE_CXX_FLAGS="
-fprofile-arcs -ftest-coverage " -DCMAKE_C_FLAGS="-fprofile-arcs -ftest-coverage
-DCOVERAGE_RUN=1"
```

```
# Initialisieren
colcon lcov-result --packages-select <package_name> --zero-counters
colcon lcov-result --packages-select <package_name> --initial
```

```
# Running the tests
colcon test --packages-select <package_name>
```

```
# Calculating the coverage
colcon lcov-result --packages-select <package_name> --verbose
```

Die Ergebnisse der Code Coverage Berechnung sind im neuen Ordner **lcov/** gespeichert.

Alternativ können die Tests und die Coverage Berechnung auch über die Skripte im Ordner **scripts/** ausgeführt werden. Die Skripte müssen im Basisordner des Projektes aufgerufen werden.

Zum Starten der Tests für ein Paket:

```
. scripts/run_tests.sh
```

Zum Starten der Code Coverage Berechnung für ein Paket:

```
. scripts/calc_coverage.sh
```

Nchfolgend sind die bereits vorhandenen Testfälle detailliert aufgelistet. In jeder Tabelle wird

hierbei genau ein Testfall beschrieben. Die Tabellen bestehen aus folgenden Einträgen:

1. **Name:** Der Name des Testfalls
2. **Testobjekt:** Das Testobjekt, das vom Testfall geprüft wird. Hierbei kann es sich um einzelne Methoden, eine Kommunikationsschnittstelle oder eine komplette Node handeln.
3. **Beschreibung:** - Eine Beschreibung des Testfalls.
4. **Vorbedingung:** - Eine Beschreibung der Vorbedingungen, die für den Testfall vorliegen müssen.
5. **Eingabewert(e):** Ein oder mehrere Werte, die als Eingabe für den Testfall verwendet werden.
6. **Erwarteter Ausgabe:** Der erwartete Rückgabewert des Testobjekts.

Spurerkennung

Die Testfälle für die LaneDetection sind im Ordner `test/` zu finden. Sie bestehen aus Unit-, Integrations- und Simulationstests. Die Tests sind für den vorgeschlagenen [Kontrollfluss](#) ausgelegt. Die Unit-Tests überprüfen hierbei die einzelnen Methoden, die Integrationstests simulieren den Eingang verschiedener Nachrichten und prüfen die Reaktion des Paketes auf diese und die Simulationstests prüfen repräsentative Fahrsituationen.



Sollte der Kontrollfluss oder Methodensignaturen geändert werden, müssen die Testfälle ebenfalls angepasst werden. Um kurzzeitig das Bestehen der Testfälle zu garantieren, können die Bedingungen durch ein `ASSERT_TRUE(true)` ersetzt werden. Dies bewirkt, dass der Testfall in jedem Fall als erfolgreich markiert wird. Spätestens zum Abschluss des Projektes müssen die Testfälle mit echten Testbedingungen bestanden sein.

Unit-Tests

Die Unit-Tests sind in der Datei `test/unit_tests.cpp` zu finden. Hier werden die einzelnen Methoden des LaneDetection Pakets getestet. Jede Methode wird hierbei mindestens einmal aufgerufen. Die Unittests sind nachfolgend alle aufgelistet. Der Eintrag Eingabewerte beschreibt hierbei immer den zu testenden Parameter. Wenn eine Methode mehr als einen Parameter akzeptiert, kann davon ausgegangen werden dass die nicht genannten Variablen gültig sind.

Testcase 1:

Name	TestResizeImage
Testobjekt	resizeImage(...)
Beschreibung	Testet, ob ein Eingabebild auf die korrekte Größe verkleinert wird.
Vorbedingung	Keine
Eingabewert(e)	Ein Eingabebild mit einer Größe abweichend von 640x480
Erwartete Ausgabe	Bild mit korrekter Größe

Testcase 2:

Name	TestImageResizeNoChange
Testobjekt	resizeImage(...)
Beschreibung	Testet, ob ein Eingabebild nicht verändert wird, wenn die Größe bereits korrekt ist.
Vorbedingung	Keine
Eingabewert(e)	Ein Eingabebild mit einer Größe 640x480
Erwartete Ausgabe	Bild mit korrekter Größe

Testcase 3:

Name	TestDoesNotResizeEmptyImage
Testobjekt	resizeImage(...)
Beschreibung	Testet, ob ein Eingabebild nicht verändert wird, wenn das Eingabebild leer ist.
Vorbedingung	Keine
Eingabewert(e)	Ein leeres Eingabebild mit einer Größe 0x0
Erwartete Ausgabe	Leeres Bild und kein Absturz

Testcase 4:

Name	TestGrayScaleImage
Testobjekt	grayscaleImage(...)
Beschreibung	Testet, ob ein Eingabebild in Graustufen konvertiert wird.
Vorbedingung	Keine
Eingabewert(e)	Ein Farbbild mit einer Größe 640x480
Erwartete Ausgabe	Bild mit nur einem Kanal

Testcase 5:

Name	TestDoesNotGrayScaleEmptyImage
Testobjekt	grayscaleImage(...)
Beschreibung	Testet, ob ein Eingabebild nicht in Graustufen konvertiert wird, wenn das Eingabebild leer ist.
Vorbedingung	Keine
Eingabewert(e)	Ein leeres Eingabebild mit einer Größe 0x0
Erwartete Ausgabe	Leeres Bild und kein Absturz

Testcase 6:

Name	TestCanHandleGrayScaleImageAsInput
Testobjekt	grayScaleImage(...)
Beschreibung	Testet, ob ein 1-Kanal Graustufenbild korrekt behandelt wird.
Vorbedingung	Keine
Eingabewert(e)	Ein Graustufenbild mit einer Größe 640x480
Erwartete Ausgabe	Das unveränderte Eingabebild

Testcase 7:

Name	TestDoesGrayScaleCorrectly
Testobjekt	grayScaleImage(...)
Beschreibung	Testet, ob ein Eingabebild in Graustufen konvertiert wird.
Vorbedingung	Lookup Table mit den korrekten Graustufenwerten muss vorhanden sein.
Eingabewert(e)	Ein Bild mit unterschiedlichen Farbsegmenten.
Erwartete Ausgabe	Bild mit Graustufen

Testcase 8:

Name	TestDoesNotTransformEmptyImage
Testobjekt	transformImage(...)
Beschreibung	Testet, ob ein Eingabebild nicht transformiert wird, wenn das Eingabebild leer ist.
Vorbedingung	Keine
Eingabewert(e)	Ein leeres Eingabebild mit einer Größe 0x0
Erwartete Ausgabe	Leeres Bild und kein Absturz

Testcase 9:

Name	TestDoesNotTransformEmptyHomography
Testobjekt	transformImage(...)
Beschreibung	Testet, ob ein Eingabebild nicht transformiert wird, wenn die Homographiematrix leer ist.
Vorbedingung	Keine
Eingabewert(e)	Eine leere Homographiematrix
Erwartete Ausgabe	Leeres Bild und kein Absturz

Testcase 10:

Name	TestDoesNotTransformHomographyNot3x3
Testobjekt	transformImage(...)
Beschreibung	Testet, ob ein Eingabebild nicht transformiert wird, wenn die Homographiematrix nicht 3x3 ist.
Vorbedingung	Keine
Eingabewert(e)	Eine Homographiematrix mit einer Größe 2x3
Erwartete Ausgabe	Leeres Bild und kein Absturz

Testcase 11:

Name	TestDoesNotBinarizeEmptyImage
Testobjekt	binarizeImage(...)
Beschreibung	Testet, ob ein Eingabebild nicht in ein Binärbild konvertiert wird, wenn das Eingabebild leer ist.
Vorbedingung	Keine
Eingabewert(e)	Ein leeres Eingabebild mit einer Größe 0x0
Erwartete Ausgabe	Leeres Bild und kein Absturz

Testcase 12:

Name	TestResultEmptyIfLowerGreaterThanUpper
Testobjekt	binarizeImage(...)
Beschreibung	Testet, ob ein Eingabebild nicht in ein Binärbild konvertiert wird, wenn der untere Grenzwert größer ist als der obere.
Vorbedingung	Keine
Eingabewert(e)	Threshold_low > Threshold_high
Erwartete Ausgabe	Leeres Bild und kein Absturz

Testcase 13:

Name	TestResultEmptyIfLowerIsEqUpper
Testobjekt	binarizeImage(...)
Beschreibung	Testet, ob ein Eingabebild nicht in ein Binärbild konvertiert wird, wenn der untere Grenzwert gleich dem oberen ist.
Vorbedingung	Keine

Eingabewert(e)	Threshold_low == Threshold_high
Erwartete Ausgabe	Leeres Bild und kein Absturz

Testcase 14:

Name	TestDoesCreateBinaryImage
Testobjekt	binarizeImage(...)
Beschreibung	Testet, ob ein Eingabebild in ein Binärbild konvertiert wird. Das Eingabebild entspricht einer oberen Dreiecksmatrix mit Pixelwerten von 180 und 255.
Vorbedingung	Keine
Eingabewert(e)	Das oben beschriebene Eingabebild
Erwartete Ausgabe	Das resultierende Binärbild

Testcase 15:

Name	TestDoesReturnEmptyImageIfElementsBelowThresh
Testobjekt	binarizeImage(...)
Beschreibung	Testet, ob ein leeres Bild zurückgegeben wird, falls alle Pixelwerte niedriger als der untere Grenzwert sind.
Vorbedingung	Keine
Eingabewert(e)	Ein Eingabebild mit einer Größe 640x480 mit Pixelwerten von 126
Erwartete Ausgabe	Ein leeres Bild mit einer Größe 640x480

Testcase 16:

Name	TestDoesNotExtractLanesEmptyImage
Testobjekt	extractLaneMarkings(...)
Beschreibung	Testet, ob in einem leeren Bild auch keine Spurmarkierungen detektiert werden.
Vorbedingung	Keine
Eingabewert(e)	Ein leeres Eingabebild mit einer Größe 0x0
Erwartete Ausgabe	Ein leeres Ergebnisvektor = keine detektierten Spurmarkierungen

Testcase 17:

Name	TestDoesExtractLanesThreeStraightLanes
Testobjekt	extractLaneMarkings(...)
Beschreibung	Testet, ob in einem Eingabebild drei gerade Linien detektiert werden.
Vorbedingung	Lookup Tabelle mit den Koordinaten der Linien
Eingabewert(e)	Computergeneriertes Eingabebild mit drei geraden Linien.
Erwartete Ausgabe	Vektor von Vektoren mit den Punkten der drei Spurmarkierungen

Testcase 18:

Name	TestCanDetectDoubleSolidMiddle
Testobjekt	extractLaneMarkings(...)
Beschreibung	Testet, ob in einem Eingabebild eine doppelte Mittellinie detektiert wird. Dies entspricht einer Zone, in der nicht überholt werden darf
Vorbedingung	Keine
Eingabewert(e)	Ein Eingabebild mit einer Größe 640x480 mit einer doppelten Mittellinie.
Erwartete Ausgabe	Datenfeld <code>no_overtaking_ == true</code>

Testcase 19:

Name	TestExtractLanesRightCurve
Testobjekt	extractLaneMarkings(...)
Beschreibung	Testet, ob die Spurmarkierungen in einer Rechtskurve detektiert werden.
Vorbedingung	Lookup Tabelle mit den Koordinaten der Linien
Eingabewert(e)	Ein computergeneriertes Eingabebild mit einer rechtskurve. Zur Erzeugung wird die Formel $y = 58 * e^{(-0.015 * x)}$ mit anschließendem Tausch der x und y Koordinaten sowie einer Verschiebung um 130/290/450 Pixel nach rechts verwendet.
Erwartete Ausgabe	Vektor von Vektoren mit den Punkten der Spurmarkierungen

Testcase 20:

Name	TestExtractLanesSnakeCurve
-------------	----------------------------

Testobjekt	extractLaneMarkings(...)
Beschreibung	Testet, ob die Spurmarkierungen in einer Verschwenkung detektiert werden.
Vorbedingung	Lookup Tabelle mit den Koordinaten der Linien
Eingabewert(e)	Ein computergeneriertes Eingabebild mit einer Verschwenkung. Zur Erzeugung wird die Formel $y = 30 * \sin(x/100) + 30$ mit anschließendem Tausch der x und y Koordinaten sowie einer Verschiebung um 10/220/450 Pixel nach rechts verwendet.
Erwartete Ausgabe	Vektor von Vektoren mit den Punkten der Spurmarkierungen

Testcase 21:

Name	TestDoesReturnEmptyForNoMarkings
Testobjekt	extractLaneMarkings(...)
Beschreibung	Testet, ob ein leerer Ergebnisvektor zurückgegeben wird, wenn das Eingabebild keine Spurmarkierungen enthält werden.
Vorbedingung	Keine
Eingabewert(e)	Ein Eingabebild mit einer Größe 640x480 mit Pixelwerten von 0
Erwartete Ausgabe	Ein leerer Ergebnisvektor = keine detektierten Spurmarkierungen

Testcase 22:

Name	TestDoesNotFindStopLineInEmptyImage
Testobjekt	extractStopLine(...)
Beschreibung	Testet, ob in einem leeren Bild auch keine Stoplinie detektiert wird.
Vorbedingung	Keine
Eingabewert(e)	Ein leeres Eingabebild mit einer Größe 0x0
Erwartete Ausgabe	Keine Detektion der Stoplinie

Testcase 23:

Name	TestDoesDetectSolidStopLine
Testobjekt	extractStopLine(...)

Beschreibung	Testet, ob in einem Eingabebild eine durchgezogene Stoplinie detektiert wird.
Vorbedingung	Keine
Eingabewert(e)	Ein Eingabebild mit einer Größe 640x480 mit einer durchgezogenen Stoplinie.
Erwartete Ausgabe	Entfernung der Stoplinie in Pixeln korrekt

Testcase 24:

Name	TestDoesDetectDashedStopLine
Testobjekt	extractStopLine(...)
Beschreibung	Testet, ob in einem Eingabebild eine gestrichelte Stoplinie detektiert wird.
Vorbedingung	Keine
Eingabewert(e)	Ein Eingabebild mit einer Größe 640x480 mit einer gestrichelten Stoplinie.
Erwartete Ausgabe	Entfernung der Stoplinie in Pixeln korrekt

Testcase 25:

Name	TestDoesTransformCorrectly
Testobjekt	transformImage(...)
Beschreibung	Testet, ob ein Eingabebild korrekt transformiert wird.
Vorbedingung	Keine
Eingabewert(e)	Eine 3x3 Matrix mit den Werten 1,2,3,4,5,6,7,8,9 als Eingabebild. Verschiedene Homographiematrizen.
Erwartete Ausgabe	Die der Transformation entsprechenden Werte der Matrix

Integrationstests

Die Integrationstests sind in der Datei `test/integration_tests.cpp` zu finden. Die Integrationstests überprüfen, ob die LaneDetectionNode korrekt auf eingehende Nachrichten reagiert und ob die Node eigene Nachrichten korrekt sendet. Für das Senden werden Dummy Publisher genutzt.

Testcase 1:

Name	TestNodeCanBeCreated
Testobjekt	LaneDetectionNode

Beschreibung	Testet, ob die LaneDetectionNode erstellt werden kann.
Vorbedingung	rclcpp::init wurde aufgerufen
Eingabewert(e)	Keine
Erwartete Ausgabe	Node im Ros-Netzwerk sichtbar

Testcase 2:

Name	TestTopicCount
Testobjekt	LaneDetectionNode
Beschreibung	Testet, ob die LaneDetectionNode genau die richtige Anzahl an Publishern und Subscriber hat.
Vorbedingung	Keine
Eingabewert(e)	Keine
Erwartete Ausgabe	Anzahl der Subscriber und Publisher korrekt

Testcase 3:

Name	TestCanReceiveImageMessages
Testobjekt	LaneDetectionNode::processImage(...)
Beschreibung	Testet, ob ein Bild in der LaneDetectionNode korrekt empfangen wird.
Vorbedingung	LaneDetection muss gestartet sein
Eingabewert(e)	Eine Bildnachricht mit einem leeren Bild, die über das Netzwerk gesendet wird
Erwartete Ausgabe	Empfange Spurmarkierungsnachricht enthält 3 leere Vektoren

Testcase 4:

Name	TestCanReceiveStateChange
Testobjekt	LaneDetectionNode::updateState(...)
Beschreibung	Testet, ob der gesendete Zustand in der LaneDetectionNode korrekt empfangen wird.
Vorbedingung	LaneDetection muss gestartet sein
Eingabewert(e)	Ein Zustandsnachricht mit einem Zustand, der über das Netzwerk gesendet wird
Erwartete Ausgabe	Interne Zustandsvariable entspricht dem gesendeten Zustand.

Testcase 5:

Name	TestDoesNotSendStatusInfoWithoutStateChange
Testobjekt	LaneDetectionNode::update(...)
Beschreibung	Testet, ob ohne Zustandswechsel keine Nachricht gesendet wird.
Vorbedingung	LaneDetection muss gestartet sein. Zustand ist noch nicht geändert worden.
Eingabewert(e)	Keine
Erwartete Ausgabe	Nach 3 Sekunden wird noch keine Spurmarkierungsnachricht im Testfall empfangen.

Testcase 6:

Name	TestDoesSendStatusInfoInStateTen
Testobjekt	LaneDetectionNode::update(...)
Beschreibung	Testet, ob eine Spurmarkierungsnachricht gesendet wird, wenn der Zustand auf 10 gesetzt wird.
Vorbedingung	LaneDetection muss gestartet sein. Aktueller Zustand ist 10.
Eingabewert(e)	Keine
Erwartete Ausgabe	Spurmarkierungsnachricht wird innerhalb von 3 Sekunden empfangen.

Testcase 7:

Name	TestDoesSendStopLineIfInCorrectState
Testobjekt	LaneDetectionNode::update(...)
Beschreibung	Testet, ob eine Spurmarkierungsnachricht gesendet wird, wenn der aktuelle Zustand 10, 13 oder 14 ist.
Vorbedingung	LaneDetection muss gestartet sein. Zustand ist 10, 13 oder 14.
Eingabewert(e)	Keine
Erwartete Ausgabe	Spurmarkierungsnachricht wird innerhalb von 3 Sekunden für jeden Zustand empfangen.

Testcase 8:

Name	TestCanResizeImage
-------------	--------------------

Testobjekt	LaneDetectionNode::processImage(...)
Beschreibung	Testet, ob ein Bild mit falschen Dimensionen korrekt verarbeitet wird.
Vorbedingung	LaneDetection muss gestartet sein.
Eingabewert(e)	Eine Bildnachricht mit einem Bild mit falschen Dimensionen, die über das Netzwerk gesendet wird
Erwartete Ausgabe	Interne Bildvariablen entsprechen hat die korrekte Dimension von 640x480 Pixeln.

Simulationstests

Die Simulationstests testen die Node als Ganzes. Hierfür wurden [ROS Bags](#) genutzt. Bei der Initialisierung der Test Suite wird der Bag gelesen und die Nachrichten deserialisiert. Dieser Schritt ist essenziell, da die gespeicherten Nachrichten ansonsten nicht verarbeitet werden können. Die deserialisierten Nachrichten werden in einem Vektor gespeichert und können anschließend gepublisht werden. Durch die Speicherung in einem Vektor wird das wiederholte Laden des Bags vermieden, was die Simulationstests beschleunigt. Die Bags enthalten 4 Szenarien:

1. Fahrt auf gerader Strecke
2. Fahrt auf der äußeren Spur eines Kreises
3. Fahrt auf der inneren Spur eines Kreises
4. Spurwechsel auf die linke Fahrbahn

Die Szenarien sind sowohl für das PSAF 1 sowie für das PSAF 2 Auto mithilfe der [Simulationsumgebung](#) erstellt worden. Die Auswahl der Bags je nach Seminar erfolgt automatisch, indem die Flag `PSAF1` in der Datei `psaf_configuration.hpp` ausgewertet wird. In der Abbildung [Simulation](#) sind die Unterschiede zwischen den Szenarien zu sehen.

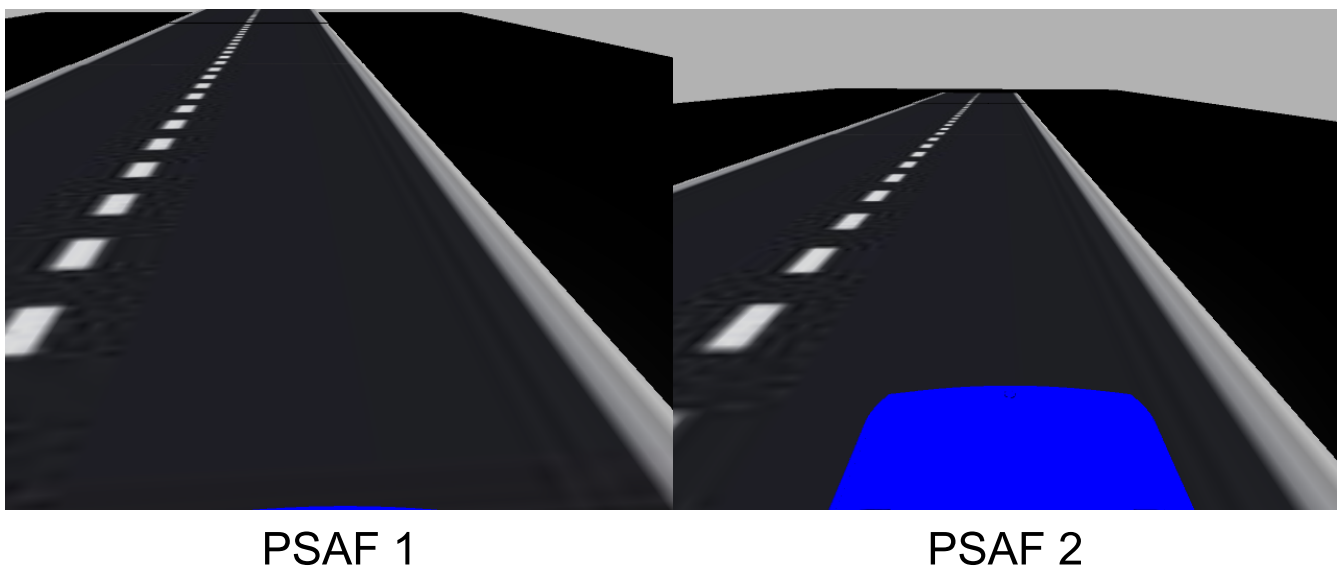


Figure 6. Beispiele aus der Simulationsumgebung

Die Simulationstests nutzen verschiedene Methoden, um die Korrektheit der eigenen Algorithmen zu überprüfen. Der Tests für die **Fahrt auf gerader Strecke** berechnet eine Gerade aus dem ersten und dem letzten Punkt der erkannten Spurmarkierungen. Alle Punkte dazwischen müssen auf oder geringfügig neben dieser Gerade liegen.

Für die Überprüfung der Kreisfahrt enthalten die Simulationstests eine Implementierung des **SlidingWindow Algorithmus**. Die Tests überprüfen, ob die durch die eigene Implementierung detektierten Punkte in der Nähe der vom Sliding Window Algorithmus detektierten Punkte liegen. Sollte es hierzu zu Problemen kommen, können einzelne Bilder übersprungen werden, indem man eine entsprechende Abfrage einfügt. Dies sollte aber nur als letzter Ausweg genutzt werden. Die vom eigenen Algorithmus erkannten Punkte dürfen maximal in einem Radius von 25 Pixeln um die vom Referenz Sliding Window Algorithmus erkannten Punkte liegen. Bei den Spurwechseltests wird geprüft, ob sich das Fahrzeug am Ende auf der korrekten Fahrspur befindet.

Testcase 1:

Name	TestNumberOfMessagesFromEachBag
Testobjekt	Testsuite itself
Beschreibung	Testet, ob die Anzahl der eingelesenen Nachrichten aus den Bags korrekt ist.
Vorbedingung	Testsuite muss gestartet und die Bags eingelesen und deserialisiert sein.
Eingabewert(e)	Keine
Erwartete Ausgabe	Anzahl der Nachrichten aus den Bags stimmt mit der Anzahl der Nachrichten überein.

Testcase 2:

Name	TestResolutionOfImageMessages
Testobjekt	Vektor mit den eingelesenen Nachrichten
Beschreibung	Testet, ob die Bildgröße der eingelesenen Nachrichten korrekt ist.
Vorbedingung	Testsuite muss gestartet und die Bags eingelesen und deserialisiert sein.
Eingabewert(e)	Keine
Erwartete Ausgabe	Die Bildgröße der eingelesenen Nachrichten stimmt mit 640x480 Pixeln überein.

Testcase 3:

Name	TestCanReceiveImageMessages
Testobjekt	LaneDetectionNode

Beschreibung	Testet, ob die Nachrichten in der LaneDetectionNode empfangen werden können.
Vorbedingung	LaneDetectionNode muss gestartet sein. Bags müssen eingelesen und deserialisiert sein.
Eingabewert(e)	Keine
Erwartete Ausgabe	LaneDetectionNode empfängt Bildnachrichten. Interne Bildvariable hat die korrekte Dimension von 640x480 Pixeln.

Testcase 4:

Name	TestResultOfStraightLaneExtraction
Testobjekt	LaneDetectionNode
Beschreibung	Testet, ob die detektierten Spurmarkierungen aus der LaneDetectionNode korrekt sind.
Vorbedingung	LaneDetectionNode muss gestartet sein. Bags müssen eingelesen und deserialisiert sein.
Eingabewert(e)	Nachrichten mit der simulierten Fahrt auf gerader Strecke
Erwartete Ausgabe	Alle Punkte für jede Spurmarkierung liegen auf einer Linie.

Testcase 5:

Name	TestDetectedPointsMatchLanesInInnerCircle
Testobjekt	LaneDetectionNode
Beschreibung	Testet, ob die detektierten Spurmarkierungen bei der Fahrt auf der inneren Spur eines Kreises korrekt sind.
Vorbedingung	LaneDetectionNode muss gestartet sein. Bags müssen eingelesen und deserialisiert sein.
Eingabewert(e)	Nachrichten mit den Bildern einer simulierten Fahrt auf der inneren Spur eines Kreises.
Erwartete Ausgabe	Zurückgelieferte Spurmarkierungen entsprechen den Punkten, die vom Kontroll Sliding-Window Algorithmus erkannt wurden.

Testcase 6:

Name	TestDetectedPointsMatchLanesInOuterCircle
Testobjekt	LaneDetectionNode

Beschreibung	Testet, ob die detektierten Spurmarkierungen bei der Fahrt auf der äußeren Spur eines Kreises korrekt sind.
Vorbedingung	LaneDetectionNode muss gestartet sein. Bags müssen eingelesen und deserialisiert sein.
Eingabewert(e)	Nachrichten mit den Bildern einer simulierten Fahrt auf der äußeren Spur eines Kreises.
Erwartete Ausgabe	Zurückgelieferte Spurmarkierungen entsprechen den Punkten, die vom Kontroll Sliding-Window Algorithmus erkannt wurden.

Testcase 7:

Name	TestDetectsStaysOnRightLane
Testobjekt	LaneDetectionNode
Beschreibung	Testet, ob das Fahrzeug immer die Fahrt auf der rechten Spur detektiert, wenn kein Spurwechsel vorkommt.
Vorbedingung	LaneDetectionNode muss gestartet sein. Bags müssen eingelesen und deserialisiert sein.
Eingabewert(e)	Nachrichten mit den Bildern einer simulierten Fahrt auf gerader Strecke.
Erwartete Ausgabe	Zurückgelieferte Nachrichten enthalten als Fahrspurseite immer den Wert 0.

Testcase 8:

Name	TestCanDetectLaneChange
Testobjekt	LaneDetectionNode
Beschreibung	Testet, ob das Fahrzeug einen Spurwechsel korrekt detektiert, wenn ein Spurwechsel durchgeführt wird.
Vorbedingung	LaneDetectionNode muss gestartet sein. Bags müssen eingelesen und deserialisiert sein.
Eingabewert(e)	Nachrichten mit den Bildern einer simulierten Fahrt auf gerader Strecke, in denen ein Spurwechsel auf die linke Spur durchgeführt wird.

Erwartete Ausgabe	Die erste zurckgelieferte Spurmarkierungsnachricht enthält als Fahrspurseite den Wert 0. Die letzte zurückgelieferte Spurmarkierungsnachricht enthält als Fahrspurseite den Wert 1.
--------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Zustandsautomat

Die Tests für die StateMachine sind abgeschlossen und befinden sich im Ordner `psaf_state_machine/test`.



Die Testfälle sind korrekt und laufen durch. Es kann dennoch vorkommen, dass die Tests in der CI-Pipeline fehlschlagen. Falls ein Test fehlschlägt, wird diese Stage in der CI-Pipeline bis zu zwei weitere Male durchgeführt. In einem Großteil der Fälle läuft die Pipeline dann auch durch. Falls die Pipeline in allen 3 Versuchen fehlschlagen sollte, kann die Testdurchführung manuell erneut gestartet werden oder wird automatisch beim nächsten Commit wiederholt.

Die Testfälle sind nachfolgend genau beschrieben.

test_state_machine.cpp

In dieser Datei werden die unabhängigen Zustände geprüft. Diese sind:

1. STARTBOX
2. MANUAL_MODE
3. ERROR
4. SHUTDOWN

Testcase 1:

Name	TestIsInStartboxState
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob sich der Zustandsautomat nach der Initialisierung im Startbox-Zustand befindet.
Vorbedingung	keine
Eingabewert(e)	keine
Erwartete Ausgabe	Der Zustandsautomat ist im Startbox-Zustand. (state == STARTBOX)

Testcase 2:

Name	TestStaysInStartBoxIfNoButtonWasPressedAndInvalidStatusInfo
-------------	-------------------------------------------------------------

Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat im Startbox-Zustand bleibt, wenn kein Button gedrückt wurde und ein ungültiges Zustandsübergangsevent übergeben wird.
Vorbedingung	kein Drücken oder Simulieren eines Knopfes
Eingabewert(e)	StatusInfo mit ungültigem Typ
Erwartete Ausgabe	Der Zustandsautomat bleibt im Startbox-Zustand. (state == STARTBOX)

Testcase 3:

Name	TestStaysInStartBoxIfNoButtonWasPressedButValidStatus
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat im Startbox-Zustand bleibt, wenn kein Button gedrückt wurde und ein gültiges Zustandsübergangsevent übergeben wird.
Vorbedingung	kein Drücken oder Simulieren eines Knopfes
Eingabewert(e)	StatusInfo mit gültigem Typ
Erwartete Ausgabe	Der Zustandsautomat bleibt im Startbox-Zustand. (state == STARTBOX)

Testcase 4:

Name	TestStaysInStartBoxStateIfButtonWasPressedButInvalidStatusInfo
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat im Startbox-Zustand bleibt, wenn ein Button gedrückt wurde und ein ungültiges Zustandsübergangsevent übergeben wird.
Vorbedingung	Drücken oder Simulieren eines Knopfes erfolgt
Eingabewert(e)	StatusInfo mit ungültigem Typ
Erwartete Ausgabe	Der Zustandsautomat bleibt im Startbox-Zustand. (state == STARTBOX)

Testcase 5:

Name	TestTransitToNormalDrive
Testobjekt	StateMachine::current_state

Beschreibung	Testet, ob der Zustandsautomat in den Normal-Fahrmodus wechselt, wenn der Button gedrückt wurde und das gültige Zustandsübergangsevent übergeben wird.
Vorbedingung	Drücken oder Simulieren eines Knopfes erfolgt
Eingabewert(e)	StatusInfo mit gültigem Typ
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Normal-Fahrmodus. (state == DR_NORMAL)

Testcase 6:

Name	TestTransitIntoErrorState
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat in den Error-Zustand wechselt, wenn ein ErrorEvent übergeben wird.
Vorbedingung	keine
Eingabewert(e)	ErrorEvent
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Error-Zustand. (state == ERROR)

Testcase 7:

Name	TestTransitIntoManualMode
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat in den Manual-Modus wechselt, wenn das StatusEvent für den manuellen Modus übergeben wird.
Vorbedingung	keine
Eingabewert(e)	StatusEvent(MANUAL_MODE_ENTER)
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Manual-Modus. (state == MANUAL_MODE)

Testcase 8:

Name	TestCanReturnFromManualMode
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat in den normalen Fahrmodus wechseln kann, wenn der manuelle Modus beendet wird.
Vorbedingung	Zustandsautomat im Zustand MANUAL_MODE

Eingabewert(e)	StatusEvent(MANUAL_MODE_EXIT)
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Normal-Fahrmodus. (state == DR_NORMAL)

Testcase 9:

Name	TestCanEnterManuelModeFromErrorState
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat in den manuellen Modus wechseln kann, wenn der Zustandsautomat im Error-Zustand ist.
Vorbedingung	Zustandsautomat im Zustand ERROR
Eingabewert(e)	StatusEvent(MANUAL_MODE_ENTER)
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Manual-Modus. (state == MANUAL_MODE)

Testcase 10:

Name	TestDoesNotRecoverInErrorState
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat in den Zustand SHUTDOWN wechselt, nachdem der Watchdog das TIMEOUT-Event ausgelöst hat.
Vorbedingung	Zustandsautomat im Zustand ERROR
Eingabewert(e)	StatusEvent(WATCHDOG_TIMEOUT)
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Zustand SHUTDOWN. (state == SHUTDOWN)

Testcase 11:

Name	TestDoesNotReactToStatusEventFromDriving
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat nicht auf ein ungültiges StatusEvent aus dem DRIVING Subautomat reagiert, während das Automat im Zustands STARTBOX ist.
Vorbedingung	Zustandsautomat im Zustand STARTBOX
Eingabewert(e)	StatusEvent(UPHILL_START)
Erwartete Ausgabe	Der Zustandsautomat bleibt im Startbox-Zustand. (state == STARTBOX)

Testcase 12:

Name	TestDoesNotReactToStatusEventFromOvertake
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat nicht auf ein ungültiges StatusEvent aus dem OVERTAKE Subautomat reagiert, während das Automat im Zustands STARTBOX ist.
Vorbedingung	Zustandsautomat im Zustand STARTBOX
Eingabewert(e)	StatusEvent(DYNAMIC_OBSTACLE)
Erwartete Ausgabe	Der Zustandsautomat bleibt im Startbox-Zustand. (state == STARTBOX)

Testcase 13:

Name	TestDoesNotReactToStatusEventFromParking
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat nicht auf ein ungültiges StatusEvent aus dem PARKING Subautomat reagiert, während das Automat im Zustands STARTBOX ist.
Vorbedingung	Zustandsautomat im Zustand STARTBOX
Eingabewert(e)	StatusEvent(PARALLEL_FOUND)
Erwartete Ausgabe	Der Zustandsautomat bleibt im Startbox-Zustand. (state == STARTBOX)

test_state_machine_discipline_one.cpp

In dieser Testklasse werden die Zustandsübergänge für die erste Disziplin im Carolo Cup "Rundkurs mit Einparken" getestet. Hierbei werden parametrisierte Tests genutzt. Bei einem parametrisierten Test werden die Eingabewert in einer Liste definiert. **googletest** generiert für jeden Eingabewert einen eigenen Testfall. Mehr Informationen über den Einsatz von parametrisierten Tests können in der Dokumentation von **googletest** gefunden werden.

Testcase 1:

Name	TestIfValidTransitionsWorks
Testobjekt	StateMachine::current_state
Beschreibung	Dieser Testfall bildet den Rahmen für die parametrisierten Tests. Übergeben wird jeweils ein Vektor, der aus {Startzustand, Zustandsübergang, Zielzustand} besteht. Dabei wird der Zustandsübergang ausgeführt und das Zielzustand überprüft.
Vorbedingung	Disziplin 1 ausgewählt

Eingabewert(e)	Vektor aus {Startzustand, Zustandsübergang, Zielzustand}
Erwartete Ausgabe	Der übergebene Zielzustand stimmt mit dem übergebenen Zielzustand überein.

Testcase 2:

Name	TestIfStateMachineDoesNotReactToInvalidOvertakeTransition
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat nicht auf ein ungültiges StatusEvent aus dem OVERTAKE Subautomat reagiert, wenn über den Knopf Disziplin 1 ausgewählt wurde.
Vorbedingung	Disziplin 1 ausgewählt
Eingabewert(e)	StatusEvent(OVERTAKE_POSSIBLE)
Erwartete Ausgabe	Der Zustandsautomat bleibt im DRIVING Subautomat. (state == DR_NORMAL)

Testcase 3:

Name	TestDoesNotReactToInvalidIndependentTransition
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat nicht auf ein ungültiges StatusEvent aus den unabhängigen Zuständen reagiert.
Vorbedingung	Disziplin 1 ausgewählt
Eingabewert(e)	StatusEvent(WATCHDOG_TIMEOUT)
Erwartete Ausgabe	Der Zustandsautomat bleibt im DRIVING Subautomat. (state == DR_NORMAL)

Testcase 4:

Name	TestCanEnterErrorModeInDriving
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat in den Error-Zustand wechselt, wenn der Zustandsautomat im DRIVING Subautomat im Zustand DR_NORMAL ist und ein ERROR-Event ausgelöst wird.
Vorbedingung	Zustandsautomat im Zustand DR_NORMAL
Eingabewert(e)	StatusEvent(ERROR)

Erwartete Ausgabe	Der Zustandsautomat wechselt in den Zustand ERROR . (state == ERROR)
--------------------------	-----------------------------------------------------------------------------

Testcase 5:

Name	TestCanEnterErrorFromParking
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat in den Error-Zustand wechselt, wenn der Zustandsautomat im PARKING Subautomat im Zustand PR_SEARCH ist und ein ERROR-Event ausgelöst wird.
Vorbedingung	Zustandsautomat im Zustand PR_SEARCH
Eingabewert(e)	StatusEvent(ERROR)
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Zustand ERROR . (state == ERROR)

test_state_machine_discipline_two.cpp

Diese Testklasse enthält die Tests für die zweite Disziplin im Carolo Cup "Rundkurs mit Hindernissen". Der Aufbau ist der gleiche wie bei der ersten Disziplin. Auch in dieser Testklasse werden parametrisierte Tests genutzt.

Testcase 1:

Name	TestIfValidTransitionsWorks
Testobjekt	StateMachine::current_state
Beschreibung	Dieser Testfall bildet den Rahmen für die parametrisierten Tests. Übergeben wird jeweils ein Vektor, der aus {Startzustand, Zustandsübergang, Zielzustand} besteht. Dabei wird der Zustandsübergang ausgeführt und das Zielzustand überprüft.
Vorbedingung	Disziplin 2 ausgewählt
Eingabewert(e)	Vektor aus {Startzustand, Zustandsübergang, Zielzustand}
Erwartete Ausgabe	Der übergebene Zielzustand stimmt mit dem übergebenen Zielzustand überein.

Testcase 2:

Name	TestIgnoresInvalidParkingTransition
Testobjekt	StateMachine::current_state

Beschreibung	Testet, ob der Zustandsautomat nicht auf ein ungültiges StatusEvent aus dem PARKING Subautomat reagiert, wenn über den Knopf Disziplin 2 ausgewählt wurde.
Vorbedingung	Disziplin 2 ausgewählt
Eingabewert(e)	StatusEvent(PARKING_INTENT)
Erwartete Ausgabe	Der Zustandsautomat bleibt im DRIVING Subautomat. (state == DR_NORMAL)

Testcase 3:

Name	TestIgnoresInvalidIndependentTransition
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat nicht auf ein ungültiges StatusEvent aus den unabhängigen Zuständen reagiert.
Vorbedingung	Disziplin 2 ausgewählt
Eingabewert(e)	StatusEvent(WATCHDOG_TIMEOUT)
Erwartete Ausgabe	Der Zustandsautomat bleibt im DRIVING Subautomat. (state == DR_NORMAL)

Testcase 4:

Name	TestCanEnterErrorModeInDriving
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat in den Error-Zustand wechselt, wenn der Zustandsautomat im DRIVING Subautomat im Zustand DR_NORMAL ist und ein ERROR-Event ausgelöst wird.
Vorbedingung	Zustandsautomat im Zustand DR_NORMAL
Eingabewert(e)	StatusEvent(ERROR)
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Zustand ERROR . (state == ERROR)

Testcase 5:

Name	TestCanEnterErrorFromOvertake
Testobjekt	StateMachine::current_state
Beschreibung	Testet, ob der Zustandsautomat in den Error-Zustand wechselt, wenn der Zustandsautomat im OVERTAKE Subautomat ist und ein ERROR-Event ausgelöst wird.

Vorbedingung	Zustandsautomat im Subautomat OVERTAKE
Eingabewert(e)	StatusEvent(ERROR)
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Zustand ERROR . (state == ERROR)

Integrationstests

Die Datei **integration_test.cpp** enthält die Integrationstests für die StateMachine. Die Integrationstests prüfen, ob die StateMachine korrekt auf externe Trigger reagiert. Externe Trigger sind:

1. Status Info Nachrichten
2. Manual Mode aktiv Meldungen
3. Error Nachrichten
4. Button Press Nachrichten

Die Nachrichten werden von einem Dummy Publisher versendet, der im **include/** Ordner des Tests liegt. Dieser Empfängt auch die Antwort der StateMachine und gibt sie an die Testsuite zurück. In den Integrationstests ist auch ein Beispiel für die automatische Erzeugung von Testfällen enthalten. Hierbei werden zufällige Sequenzen von 5, 25, 50, 100 und 500 Nachrichten erzeugt. Mithilfe eines Testorakles werden die erwarteten Nachrichten generiert und mit den erhaltenen Nachrichten verglichen. Das Testorakle nutzt die Datei **test/include/random_tests.hpp** um mithilfe der gültigen Transitionen die erwarteten Zielzustände zu generieren.

Testcase 1:

Name	TestCheckNodeName
Testobjekt	StateMachine
Beschreibung	Testet, ob der Name der StateMachineNode richtig gesetzt wurde.
Vorbedingung	rclcpp::init wurde aufgerufen
Eingabewert(e)	Keine
Erwartete Ausgabe	Der Name der StateMachineNode ist state_machine .

Testcase 2:

Name	TestTopicCount
Testobjekt	StateMachine
Beschreibung	Testet, ob die Anzahl der Topics richtig gesetzt wurde.
Vorbedingung	Keine
Eingabewert(e)	Keine

Erwartete Ausgabe	Die Anzahl der Topics ist 5 .
--------------------------	--------------------------------------

Testcase 3:

Name	TestStaysInStartBoxState
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat im Startzustand bleibt, wenn ein StatusInfo-Event empfangen aber noch kein Knopfdruck ausgeführt wurde.
Vorbedingung	Keine
Eingabewert(e)	Alle StatusInfos die einen Zustandswechsel auslösen können, außer Error und ManualMode
Erwartete Ausgabe	Der Zustandsautomat bleibt im Startzustand. (state == STARTBOX)

Testcase 4:

Name	TestLeavesStartBoxState
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat den Startzustand verlässt, falls eine Disziplin ausgewählt und eine StatusInfo-Nachricht empfangen wurde.
Vorbedingung	Disziplin ausgewählt
Eingabewert(e)	StatusInfo(STARTBOX_OPEN)
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Subautomat DRIVING . (state == DR_NORMAL)

Testcase 5:

Name	TestFullDisciplineOneParallelPark
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat die Zustandsübergänge für ein paralleles Parken richtig umsetzt.
Vorbedingung	Disziplin 1 ausgewählt
Eingabewert(e)	Sequenz an StatusInfo-Nachrichten, die bei einer Fahrt mit parallelem Einparken ausgelöst werden würden.
Erwartete Ausgabe	Der Zustandsautomat befindet sich final wieder im Fahrzustand DR_NORMAL . (state == DR_NORMAL)

Testcase 6:

Name	TestFullDisciplineOnePerpendicularPark
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat die Zustandsübergänge für ein senkrechtes Parken richtig umsetzt.
Vorbedingung	Disziplin 1 ausgewählt
Eingabewert(e)	Sequenz an StatusInfo-Nachrichten, die bei einer Fahrt mit senkrechtem Einparken ausgelöst werden würden.
Erwartete Ausgabe	Der Zustandsautomat befindet sich final wieder im Fahrzustand DR_NORMAL . (state == DR_NORMAL)

Testcase 7:

Name	TestNoParkingSpotFound
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat den Subautomat PARKING verlässt, falls kein Parkplatz gefunden wurde.
Vorbedingung	Disziplin 1 ausgewählt
Eingabewert(e)	Sequenz an StatusInfo-Nachrichten, die bei einer Fahrt mit abgebrochener Parkplatzsuche ausgelöst werden würden.
Erwartete Ausgabe	Der Zustandsautomat wechselt wieder in den Subautomat DRIVING . (state == DR_NORMAL)

Testcase 8:

Name	TestParallelParkingFailed
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat den Subautomat PARKING verlässt, falls ein paralleles Einparken fehlgeschlagen ist.
Vorbedingung	Disziplin 1 ausgewählt
Eingabewert(e)	Sequenz an StatusInfo-Nachrichten, die bei einer Fahrt mit abgebrochenem parallelen Einparken ausgelöst werden würden.
Erwartete Ausgabe	Der Zustandsautomat wechselt wieder in den Subautomat DRIVING . (state == DR_NORMAL)

Testcase 9:

Name	TestPerpendicularParkingFailed
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat den Subautomat PARKING verlässt, falls ein senkrechtes Einparken fehlgeschlagen ist.
Vorbedingung	Disziplin 1 ausgewählt
Eingabewert(e)	Sequenz an StatusInfo-Nachrichten, die bei einer Fahrt mit abgebrochenem senkrechtem Einparken ausgelöst werden würden.
Erwartete Ausgabe	Der Zustandsautomat wechselt wieder in den Subautomat DRIVING . (state == DR_NORMAL)

Testcase 10:

Name	TestTransitIntoManualModeFromStartBox
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat vom Zustand STARTBOX in den Zustand MANUAL_MODE wechseln kann, falls eine entsprechende StatusInfo-Nachricht empfangen wurde.
Vorbedingung	Keine
Eingabewert(e)	StatusInfo(MANUAL_MODE_ENTER)
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Subautomat MANUAL_MODE . (state == MANUAL_MODE)

Testcase 11:

Name	TestTransitIntoManualModeDisciplineOne
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat von jedem möglichen Zustand in Disziplin 1 in den Zustand MANUAL_MODE wechseln kann, falls eine entsprechende StatusInfo-Nachricht empfangen wurde.
Vorbedingung	Keine
Eingabewert(e)	StatusInfo(MANUAL_MODE_ENTER)
Erwartete Ausgabe	Der Zustandsautomat wechselt immer in den Zustand MANUAL_MODE . (state == MANUAL_MODE)

Testcase 12:

Name	TestDoesNotReactToInvalidMessage
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat im Zustand DR_NORMAL nicht auf eine ungültige StatusInfo-Nachricht reagiert.
Vorbedingung	Keine
Eingabewert(e)	Alle ungültigen StatusInfo Nachrichten für den Zustand DR_NORMAL
Erwartete Ausgabe	Der Zustandsautomat bleibt im Zustand DR_NORMAL . (state == DR_NORMAL)

Testcase 13:

Name	TestDisciplineObstacleEvasionCourse
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat alle Zustände für Disziplin 2 richtig umsetzt.
Vorbedingung	Disziplin 2 ausgewählt
Eingabewert(e)	Sequenz an StatusInfo-Nachrichten, die bei einer Fahrt mit Hindernissen ausgelöst werden würden.
Erwartete Ausgabe	Die zurückgelieferte Sequenz an Zuständen entspricht der erwarteten Sequenz.

Testcase 14:

Name	TestTransitIntoManualModeDisciplineTwo
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat von jedem möglichen Zustand in Disziplin 2 in den Zustand MANUAL_MODE wechseln kann, falls eine entsprechende StatusInfo-Nachricht empfangen wurde.
Vorbedingung	Disziplin 2 ausgewählt
Eingabewert(e)	StatusInfo(MANUAL_MODE_ENTER)
Erwartete Ausgabe	Der Zustandsautomat wechselt immer in den Zustand MANUAL_MODE . (state == MANUAL_MODE)

Testcase 15:

Name	TestDoesNotReactToInvalidMessage2
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat im Zustand DR_NORMAL nicht auf ungültige StatusInfo-Nachricht reagiert.
Vorbedingung	Disziplin 2 ausgewählt
Eingabewert(e)	Alle ungültigen StatusInfo Nachrichten für den Zustand DR_NORMAL
Erwartete Ausgabe	Der Zustandsautomat bleibt im Zustand DR_NORMAL . (state == DR_NORMAL)

Testcase 16:

Name	TestDoesNotReactToInvalidMessageInPark
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat im Subautomat PARKING nicht auf ungültige StatusInfo-Nachricht reagiert.
Vorbedingung	Disziplin 1 ausgewählt
Eingabewert(e)	Alle ungültigen StatusInfo Nachrichten im Subautomat PARKING
Erwartete Ausgabe	Der Zustandsautomat bleibt im Subautomat PARKING . (state == PARKING)

Testcase 17:

Name	TestStaysInManualMode
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat im Zustand MANUAL_MODE nicht auf ungültige StatusInfo-Nachricht reagiert.
Vorbedingung	Zustand MANUAL_MODE
Eingabewert(e)	Alle ungültigen StatusInfo Nachrichten für den Zustand MANUAL_MODE
Erwartete Ausgabe	Der Zustandsautomat bleibt im Zustand MANUAL_MODE . (state == MANUAL_MODE)

Testcase 18:

Name	TestEntersShutdown
Testobjekt	StateMachine

Beschreibung	Testet, ob der Zustandsautomat vom Zustand ERROR in den Zustand SHUTDOWN wechselt, falls eine WATCHDOG_TIMEOUT Nachricht empfangen wurde.
Vorbedingung	Zustand Error
Eingabewert(e)	StatusInfo(WATCHDOG_TIMEOUT)
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Zustand SHUTDOWN . (state == SHUTDOWN)

Testcase 19:

Name	TestIgnoresWarning
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat nicht auf eine ERROR -Nachricht vom Typ WARNING reagiert. Die Reaktion auf Warnungen wird vom Zustandsautomat nicht unterstützt.
Vorbedingung	Zustand DR_NORMAL
Eingabewert(e)	ERROR -Nachricht vom Typ WARNING
Erwartete Ausgabe	Der Zustandsautomat bleibt im Zustand DR_NORMAL . (state == DR_NORMAL)

Testcase 20:

Name	TestCanReturnFromManualMode
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat vom Zustand MANUAL_MODE in den Zustand DR_NORMAL wechselt, falls eine MANUAL_MODE_EXIT Nachricht empfangen wurde.
Vorbedingung	Zustand MANUAL_MODE
Eingabewert(e)	StatusInfo(MANUAL_MODE_EXIT)
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Zustand DR_NORMAL . (state == DR_NORMAL)

Testcase 21:

Name	TestReactsToErrorInDrive
Testobjekt	StateMachine

Beschreibung	Testet, ob der Zustandsautomat vom Zustand DR_NORMAL in den Zustand ERROR wechselt, falls eine ERROR Nachricht vom Typ ERROR empfangen wurde.
Vorbedingung	Zustand DR_NORMAL
Eingabewert(e)	Error-Nachricht
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Zustand ERROR . (state == ERROR)

Testcase 22:

Name	TestReactsToErrorInPark
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat vom Subautomat PARKING in den Zustand ERROR wechselt, falls eine ERROR Nachricht vom Typ ERROR empfangen wurde.
Vorbedingung	Subautomat PARKING
Eingabewert(e)	Error-Nachricht
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Zustand ERROR . (state == ERROR)

Testcase 23:

Name	TestReactsToErrorInOvertake
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat vom Subautomat OVERTAKE in den Zustand ERROR wechselt, falls eine ERROR Nachricht vom Typ ERROR empfangen wurde.
Vorbedingung	Subautomat OVERTAKE
Eingabewert(e)	Error-Nachricht
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Zustand ERROR . (state == ERROR)

Testcase 24:

Name	TestCanLeaveErrorToManualMode
Testobjekt	StateMachine

Beschreibung	Testet, ob der Zustandsautomat vom Zustand ERROR in den Zustand MANUAL_MODE wechselt, falls eine MANUAL_MODE_ENTER Nachricht empfangen wurde.
Vorbedingung	Zustand ERROR
Eingabewert(e)	StatusInfo(MANUAL_MODE_ENTER)
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Zustand MANUAL_MODE . (state == MANUAL_MODE)

Testcase 25:

Name	TestCanReactToSignMessage
Testobjekt	StateMachine
Beschreibung	Testet, ob eine SignMessage mit einen Parkplatzschild wie eine StatusInfo Nachricht vom PAKING_INTENT behandelt wird.
Vorbedingung	Zustand DR_NORMAL
Eingabewert(e)	SignMessage
Erwartete Ausgabe	Der Zustandsautomat wechselt in den Zustand PR_SEARCH

Testcase 26:

Name	TestCanIgnoreSignMessage
Testobjekt	StateMachine
Beschreibung	Testet, ob alle anderen Schiler-Nachrichten ignoriert werden. Andere Schilder als Parkplatzschilder werden noch nicht unterstützt.
Vorbedingung	Zustand DR_NORMAL
Eingabewert(e)	SignMessage
Erwartete Ausgabe	Der Zustandsautomat bleibt im Zustand DR_NORMAL . (state == DR_NORMAL)

Testcase 27:

Name	TestCanReceiveFromUpdateMethod
Testobjekt	StateMachine::update()
Beschreibung	Testet, ob die Update() Methode den aktuellen Zustand versenden kann.
Vorbedingung	Keine

Eingabewert(e)	Keine
Erwartete Ausgabe	Der aktuelle Zustand wird im Testfall empfangen.

Die nächsten Testfälle werden automatisiert erstellt. Im Testfall wird eine zufällige Testsequenz erstellt und an den Zustandsautomaten übergeben. Ein Testorakel erstellt mithilfe einer Lookup-Tabelle aus den Zustandsübergängen die erwartete Sequenz von Zuständen. Die erwartete und die tatsächliche Sequenz werden verglichen. Da der Aufbau immer gleich ist, werden die Testfälle in einer Tabelle zusammengefasst.

Testcase 28 - 32:

Name	TestRandom<NBR>Sequence
Testobjekt	StateMachine
Beschreibung	Testet, ob der Zustandsautomat auf eine zufällige Sequenz von StatusInfo Nachrichten korrekt reagieren kann. <NBR> steht für die Länge der Sequenz. Diese nimmt mit jedem Testfall zu. Die Staffelung ist 5, 25, 50, 100, 500. Die Disziplin wird ebenfalls zufällig ausgewählt.
Vorbedingung	Keine
Eingabewert(e)	Zufällige Sequenz von StatusInfo Nachrichten
Erwartete Ausgabe	Gleichheit der tatsächlichen Zustandssequenz und der vom Orakel generierten Sequenz.

Startbox

Die Tests für die StartboxNode befinden sich im Ordner `psaf_startbox/test`. Die Testfälle bestehen aus Unit-Tests, Integrationstests und Simulationstests.

Unittests

Bei den Unit-Tests werden die Methoden der StartboxNode auf Korrektheit überprüft. Hierbei wird jede Methode mindestens einmal zur Ausführung gebracht. Die Startbox Node schaltet sich ab, sobald der Zustand STARTBOX verlassen wird. Diese Funktionalität wird ebenfalls von den Unit-Tests überprüft.

Testcase 1:

Name	TestCanSetupTestSuite
Testobjekt	Testsuite
Beschreibung	Testet, ob die Testsuite korrekt initialisiert wird.
Vorbedingung	Keine
Eingabewert(e)	Keine

Erwartete Ausgabe	Kein Absturz
--------------------------	--------------

Testcase 2:

Name	TestCanReadQRCode
Testobjekt	readQR(...)
Beschreibung	Testet, ob der QR-Code korrekt gelesen wird.
Vorbedingung	Keine
Eingabewert(e)	QR-Code mit Inhalt "STOP"
Erwartete Ausgabe	Variable last_read_qr_ ist auf "STOP" gesetzt.

Testcase 3:

Name	TestDoesNotDetectQRCode
Testobjekt	readQR(...)
Beschreibung	Testet, ob kein QR-Code erkannt wird, wenn das Eingabebild keinen QR-Code enthält.
Vorbedingung	Keine
Eingabewert(e)	Eingabebild ohne QR-Code
Erwartete Ausgabe	Variable last_read_qr_ ist ohne Inhalt = "".

Testcase 4:

Name	TestCanDetectQRCodeNotStop
Testobjekt	readQR(...)
Beschreibung	Testet, ob ein QR-Code mit dem Inhalt "TEST" erkannt wird.
Vorbedingung	Keine
Eingabewert(e)	Eingabebild mit QR-Code mit Inhalt "TEST"
Erwartete Ausgabe	Variable last_read_qr_ ist auf "TEST" gesetzt.

Testcase 5:

Name	TestSetsReadFlagCorrectly
Testobjekt	readQR(...)
Beschreibung	Testet, ob die Variable detected_at_least_once_ richtig gesetzt wird. Diese zeigt an, ob der QR-Code mindestens einmal erkannt wurde.
Vorbedingung	Keine
Eingabewert(e)	Eingabebild mit QR-Code mit Inhalt "TEST"

Erwartete Ausgabe	Variable detected_at_least_once_ ist auf true gesetzt.
--------------------------	--------------------------------------------------------

Testcase 6:

Name	TestCanSetOpenCorrectly
Testobjekt	readQR(...)
Beschreibung	Testet, ob die Variable open_ richtig gesetzt wird. Diese zeigt an, ob die Startbox bereits geöffnet wurde. Die Variable wird auf true gesetzt, nachdem mindestens einmal STOP gelesen und anschließend für 11 Frames kein QR-Code erkannt wurde.
Vorbedingung	no_qr_message_counter_ = 10
Eingabewert(e)	Eingabebild ohne QR-Code
Erwartete Ausgabe	Variable open_ ist auf true gesetzt.

Testcase 7:

Name	TestCanProcessSensorZero
Testobjekt	processImage(...)
Beschreibung	Testet, ob nur die Bilder RGB-Kamera (sensor = 0) verarbeitet werden.
Vorbedingung	Keine
Eingabewert(e)	Farbbild ohne QR-Code und Sensor 0
Erwartete Ausgabe	Variable last_read_qr_ ist ohne Inhalt = "".

Testcase 8:

Name	TestCanProcessSensorOne
Testobjekt	processImage(...)
Beschreibung	Testet, ob Bilder von der Tiefenbildkamera (sensor = 1) ignoriert werden.
Vorbedingung	Keine
Eingabewert(e)	Bild mit QR-Code und Sensor 1
Erwartete Ausgabe	Variable last_read_qr_ bleibt auf dem initialen Wert "INIT".

Testcase 9:

Name	TestCanReactToEmptyImage
-------------	--------------------------

Testobjekt	processImage(...)
Beschreibung	Testet, ob die StartboxNode mit einem leeren Bild umgehen kann.
Vorbedingung	Keine
Eingabewert(e)	Leeres Bild mit Dimensionen 0x0
Erwartete Ausgabe	Variable last_read_qr_ bleibt auf dem initialen Wert "INIT" und es kommt nicht zu einem Absturz.

Testcase 10:

Name	TestCanReactToWrongWidth
Testobjekt	processImage(...)
Beschreibung	Testet, ob die StartboxNode mit einem Bild mit einer falschen Breite umgehen kann.
Vorbedingung	Keine
Eingabewert(e)	QR-Code Bild mit Breite 1280x480
Erwartete Ausgabe	Das Bild wird in die richtige Auflösung skaliert und der QR-Code korrekt gelesen. (Inhalt "STOP").

Testcase 11:

Name	TestCanReactToWrongHeight
Testobjekt	processImage(...)
Beschreibung	Testet, ob die StartboxNode mit einem Bild mit einer falschen Höhe umgehen kann.
Vorbedingung	Keine
Eingabewert(e)	QR-Code Bild mit Höhe 640x960
Erwartete Ausgabe	Das Bild wird in die richtige Auflösung skaliert und der QR-Code korrekt gelesen. (Inhalt "STOP").

Testcase 12:

Name	TEstCanReactToWrongWidthAndHeight
Testobjekt	processImage(...)
Beschreibung	Testet, ob die StartboxNode mit einem Bild mit einer falschen Breite und Höhe umgehen kann.
Vorbedingung	Keine
Eingabewert(e)	QR-Code Bild mit Breite 1280x960

Erwartete Ausgabe	Das Bild wird in die richtige Auflösung skaliert und der QR-Code korrekt gelesen. (Inhalt "STOP").
--------------------------	----------------------------------------------------------------------------------------------------

Testcase 13:

Name	TestReadBarCodeButDoesNotReact
Testobjekt	processImage(...)
Beschreibung	Die verwendete Bibliothek zbar kann auch Barcodes lese. Auf diese darf jedoch keine Reaktion erfolgen.
Vorbedingung	Keine
Eingabewert(e)	Bild mit einem Barcode
Erwartete Ausgabe	Variable last_read_qr_ bleibt auf dem initialen Wert "INIT".

Testcase 14:

Name	TestCanAssignState
Testobjekt	updateState(...)
Beschreibung	Testet, ob die Variable current_state_ richtig gesetzt wird. Hierfür wird die Callback-Funktion updateState aufgerufen.
Vorbedingung	Keine
Eingabewert(e)	StateNachricht mit Wert 42
Erwartete Ausgabe	Variable current_state_ ist auf 42 gesetzt.

Testcase 15:

Name	TestCanCallUpdate
Testobjekt	update(...)
Beschreibung	Testet, ob die Methode update() aufgerufen werden kann und es nicht zu einem Absturz kommt.
Vorbedingung	is_open_ = true
Eingabewert(e)	Keine
Erwartete Ausgabe	Die Methode update() wird aufgerufen. Es kommt nicht zu einem Absturz.

Testcase 16:

Name	TestCanCallUpdateWithFalseIsOpen
-------------	----------------------------------

Testobjekt	update(...)
Beschreibung	Testet, ob die Methode <code>update()</code> aufgerufen werden kann und es nicht zu einem Absturz kommt.
Vorbedingung	is_open_ = false
Eingabewert(e)	Keine
Erwartete Ausgabe	Die Methode <code>update()</code> wird aufgerufen. Es kommt nicht zu einem Absturz.

Testcase 17:

Name	TestDoesIgnoreIfSensorIsNotZero
Testobjekt	updateSensorValue(...)
Beschreibung	Testet, ob Messwerte die nicht von US-Sensor 0 (Front) sind, ignoriert werden.
Vorbedingung	Keine
Eingabewert(e)	Messwert von US-Sensor 1
Erwartete Ausgabe	Variable last_received_distance_ bleibt auf dem initialen Wert "0.0".

Testcase 18:

Name	TestDoesReactToZeroSensor
Testobjekt	updateSensorValue(...)
Beschreibung	Testet, ob Messwerte von US-Sensor 0 (Front) richtig verarbeitet werden.
Vorbedingung	Keine
Eingabewert(e)	Messwert 100.0 von US-Sensor 0
Erwartete Ausgabe	Variable last_received_distance_ wird auf den Wert empfangenen Wert "100.0" gesetzt.

Testcase 19:

Name	TestIgnoresZeroRange
Testobjekt	updateSensorValue(...)
Beschreibung	Testet, ob Messwerte mit Wert 0.0 ignoriert werden. Diese treten nur bei fehlerhaften Messungen auf.
Vorbedingung	Keine
Eingabewert(e)	Messwert 0.0 von US-Sensor 0

Erwartete Ausgabe	Variable last_received_distance_ bleibt auf dem im Testfall gesetzten Wert initialen Wert "-1.0".
--------------------------	---------------------------------------------------------------------------------------------------

Testcase 20:

Name	TestIgnoredNegativeRange
Testobjekt	updateSensorValue(...)
Beschreibung	Testet, ob Messwerte mit negativen Werten ignoriert werden. Diese treten nur bei fehlerhaften Messungen auf.
Vorbedingung	Keine
Eingabewert(e)	Messwert -10.0 von US-Sensor 0
Erwartete Ausgabe	Variable last_received_distance_ bleibt auf dem im Testfall gesetzten Wert initialen Wert "5.0".

Testcase 21:

Name	TestIncreasesCounterIfRangeGreaterThirty
Testobjekt	updateSensorValue(...)
Beschreibung	Testet, ob der Zähler für Messungen mit Werten größer 0.30 erhöht wird.
Vorbedingung	Keine
Eingabewert(e)	Messwert 0.31 von US-Sensor 0
Erwartete Ausgabe	Variable us_message_counter_ wird von 0 auf 1 erhöht.

Testcase 22:

Name	TestDoesNotIncreaseCounterIfRangeLessThanThirty
Testobjekt	updateSensorValue(...)
Beschreibung	Testet, ob der Zähler für Messungen mit Werten kleiner 0.30 nicht erhöht wird.
Vorbedingung	Keine
Eingabewert(e)	Messwert 0.29 von US-Sensor 0
Erwartete Ausgabe	Variable us_message_counter_ bleibt auf dem initialen Wert 0.

Testcase 23:

Name	TestDoesIncreaseCounterIfRangeIsThirty
-------------	----------------------------------------

Testobjekt	updateSensorValue(...)
Beschreibung	Testet, ob der Zähler für Messungen mit Werten 0.30 erhöht wird.
Vorbedingung	Keine
Eingabewert(e)	Messwert 0.30 von US-Sensor 0
Erwartete Ausgabe	Variable us_message_counter_ wird von 0 auf 1 erhöht.

Testcase 24:

Name	TestCanIncreaseMultipleTimes
Testobjekt	updateSensorValue(...)
Beschreibung	Testet, ob der Zähler für Messungen mit Werten größer 0.30 erhöht wird.
Vorbedingung	Keine
Eingabewert(e)	3 Messwerte von US-Sensor 0 mit Werten 0.31, 0.32, 0.3
Erwartete Ausgabe	Variable us_message_counter_ wird von 0 auf 3 erhöht.

Testcase 25:

Name	TestCanResetCounterToZeroIfRangeWasLessThanThirty
Testobjekt	updateSensorValue(...)
Beschreibung	Testet, ob der Zähler bei Messungen mit Werten kleiner 0.30 zurückgesetzt wird.
Vorbedingung	Keine
Eingabewert(e)	Messwert 0.29 von US-Sensor 0
Erwartete Ausgabe	Variable us_message_counter_ wird auf 0 zurückgesetzt.

Testcase 26:

Name	TestSetIsOpenAfterElevenMeasurementsOverThirty
Testobjekt	updateSensorValue(...)
Beschreibung	Testet, ob die Variable is_open_ auf true gesetzt wird, wenn die 11te Messung mit Werten größer 0.30 erfolgt.
Vorbedingung	Keine

Eingabewert(e)	Messwert 0.31 von US-Sensor 0
Erwartete Ausgabe	Variable is_open_ wird auf true gesetzt.

Testcase 27:

Name	TestShutdownCanBeCalled
Testobjekt	update()
Beschreibung	Testet, ob sich die Node abschaltet falls der aktuelle Zustand nicht "STARTBOX" ist und die update()-Methode aufgerufen wird.
Vorbedingung	Zustand ist nicht "STARTBOX"
Eingabewert(e)	Keine
Erwartete Ausgabe	Node wird beendet.

Integrationstest

Die Integrationstests überprüfen die Kommunikationsschnittstellen der Startbox. Diese sind:

1. ImageSubscriber
2. UltrasonicSubscriber
3. StateSubscriber
4. StatusInfoPublisher

Testcase 1:

Name	TestNodeCanBeCreated
Testobjekt	StartboxNode
Beschreibung	Testet, ob eine StartboxNode erstellt werden kann.
Vorbedingung	rclcpp::init() wurde aufgerufen
Eingabewert(e)	Keine
Erwartete Ausgabe	"startbox" im ROS-Nodegraph

Testcase 2:

Name	TestCanInitTestSuite
Testobjekt	TestSuite
Beschreibung	Testet, ob die TestSuite initialisiert werden kann.
Vorbedingung	Keine
Eingabewert(e)	Keine
Erwartete Ausgabe	Kein Fehler

Testcase 3:

Name	TestTopicCount
Testobjekt	StartboxNode
Beschreibung	Testet, ob die Anzahl der Topics im ROS-Nodegraph korrekt ist.
Vorbedingung	Keine
Eingabewert(e)	Keine
Erwartete Ausgabe	6 + Anzahl der US-Sensoren

Testcase 4:

Name	TestCanReceiveQRCodeImageMessageAndDecode
Testobjekt	StartboxNode
Beschreibung	Testet, ob ein QR-Code-Bild erfolgreich empfangen und decodiert wird.
Vorbedingung	Keine
Eingabewert(e)	Bildnachrichten mit und ohne QR-Code
Erwartete Ausgabe	Empfang der StatusInfo "STARTBOX_OPEN"

Testcase 5:

Name	TestCanReceiveUltrasonicMessageAndDecode
Testobjekt	StartboxNode
Beschreibung	Testet, ob ein Ultraschallsensor-Wert erfolgreich empfangen und decodiert wird.
Vorbedingung	Keine
Eingabewert(e)	Messwert von Ultraschallsensor 0
Erwartete Ausgabe	Empfang der StatusInfo "STARTBOX_OPEN"

Testcase 6:

Name	TestCanReceiveStateAndShutsDown
Testobjekt	StartboxNode
Beschreibung	Testet, ob nach dem Empfang eines anderen Zustands die Node abgeschaltet wird.
Vorbedingung	Keine
Eingabewert(e)	Nachricht mit Zustand 12
Erwartete Ausgabe	Node wird beendet.

Simulationstests

In den Simulationstests wird überprüft, ob die StartboxNode in der Lage ist das Öffnen der Startbox zu erkennen. Hierfür sind 2 Bags vorhanden. Einmal mit Bildern, einmal mit Ultraschallsignalen.



Die Bags enthalten Daten aus echten Aufnahmen von Modellautos, da die Simulationsumgebung weder die Simulation der Startboxöffnung unterstützt noch konkrete Ultraschallwerte geliefert hat. Sobald die Simulationsumgebung entsprechend ergänzt wurde, müssen die Bags entsprechend ausgetauscht werden.

Testcase 1:

Name	TestCanInitTestSuite
Testobjekt	TestSuite
Beschreibung	Testet, ob die TestSuite initialisiert werden kann.
Vorbedingung	Keine
Eingabewert(e)	Keine
Erwartete Ausgabe	Kein Fehler

Testcase 2:

Name	TestImageBagCount
Testobjekt	StartboxNode
Beschreibung	Testet, ob die Anzahl der Nachrichten im lokalen Vektor mit den Anzahl der Bilder im Bag-File übereinstimmt.
Vorbedingung	Keine
Eingabewert(e)	Keine
Erwartete Ausgabe	Anzahl der Nachrichten = 74

Testcase 3:

Name	TestRangeBagCount
Testobjekt	StartboxNode
Beschreibung	Testet, ob die Anzahl der Nachrichten im lokalen Vektor mit der Anzahl der Ultraschallsignale im Bag-File übereinstimmt.
Vorbedingung	Keine
Eingabewert(e)	Keine
Erwartete Ausgabe	Anzahl der Nachrichten = 251

Testcase 4:

Name	TestCanDetectOpeningBoxWithImage
Testobjekt	StartboxNode
Beschreibung	Testet, ob die Öffnung der Startbox über einen Videostream erkannt wird.
Vorbedingung	Keine
Eingabewert(e)	Bildnachrichten, die eine Toröffnung zeigen.
Erwartete Ausgabe	node → is_open_ = true

Testcase 5:

Name	TestCanDetectOpenBoxWithUS
Testobjekt	StartboxNode
Beschreibung	Testet, ob die Öffnung der Startbox über Ultraschallsignale erkannt wird.
Vorbedingung	Keine
Eingabewert(e)	Ultraschallsignale, die eine Toröffnung zeigen.
Erwartete Ausgabe	node → is_open_ = true

Setup

Für die Entwicklung wird [ROS2 Foxy](#) auf Ubuntu 20.04 verwendet. In diesem Abschnitt wird die Installation der benötigten Pakete auf Ubuntu erklärt, sowie der Setup Prozess für die Entwicklung unter Windows erklärt.

Ubuntu Installation

Windows Host

Die direkte Entwicklung von ROS Anwendungen unter Windows ist möglich, führt aber oftmals zu Problemen und ungewolltem Verhalten. Aus diesem Grund werden zwei Alternativen vorgestellt. Die Entwicklung mit einer VM oder und dem Windows Subsystem for Linux.

VM

Die Entwicklung in einer Virtual Machine (VM) ermöglicht es, trotz Windows OS, mit Ubuntu zu arbeiten. Es sei angemerkt, dass die Performance der VM eingeschränkt sein kann und es vor allem bei der Nutzung einer Kamera zu Bandbreiten und Verbindungsproblemen kommen kann. In diesem Tutorial wird die Einrichtung der VM mit [Oracle VirtualBox](#) erläutert. Der [VMWare Player](#) kann ebenfalls verwendet werden, jedoch wird hierfür keine Installationsanleitung bereitgestellt.

Automatisches Erstellen des VM Images

Im Repository [Images](#) sind Scripte zu finden, um automatisiert ein eingerichtetes VM Image zu erstellen. Ausführlichere Informationen zu den verwendeten Paketen und den Installationsscripten kann in der [Dokumentation](#) des [images](#) Repo gefunden werden. Im Folgenden ist eine Schnellanleitung beschrieben.

1. [VirtualBox](#) installieren.
2. Klonen des [Image](#) Repositorys

```
git clone https://git-ce.rwth-aachen.de/af/images
```

3. [Packer](#) für Windows herunterladen und entpacken
4. Die Datei [packer.exe](#) in den zuvor geklonten [image](#) Ordner kopieren.
5. Öffnen der Windows Eingabeaufforderung (**WIN+R** → **cmd**) oder der Windows Powershell (**WIN+R** → **powershell**).
6. In den [image](#) Ordner navigieren.

```
cd <path/to/folder/images>
```

Falls der [images](#) Ordner im [Downloads](#) Order liegt, ost der Befehl

```
$ cd %HOMEPATH%/Downloads
```

Anschließend müssen die folgende Scripte in der genannten Reihenfolge ausgeführt werden.

1. Installieren des Ubuntu 20.04 Server Image in einer VM. Dieser Prozess kann einige Zeit in Anspruch nehmen.

```
$ packer build ubuntu.json      # cmd  
$ ./packer.exe build ubuntu.json # powershell
```

2. Installieren des VB-GuestAdditions Add-on sowie der Ubuntu Gnome Desktopumgebung

```
# installation der role aus ansible-galaxy  
$ ansible-galaxy install PeterMosmans.virtualbox-guest  
# installation des desktops  
$ packer build desktop.json
```

3. Abschließend muss noch ROS installiert werden:


```
packer build ros.json
```

Manuelle Einrichtung der VM

In der VM muss eine Linux Distribution installiert werden. Zu empfehlen sind [Ubuntu 20.04](#) oder [XUbuntu 20.04](#). [XUbuntu](#) ist in der Regel etwas schneller in der Ausführung.

1. Herunterladen von [Ubuntu 20.04](#) oder [XUbuntu 20.04](#)
2. Herunterladen von [Oracle VM Virtual Box](#) oder [VMWare Player](#).
3. Öffnen von Virtual Box
4. Erstellen einer neuen virtuellen Maschine
 1. Auf "Neu" klicken
 2. Eingabe von Name und Speicherort der VM. WICHTIG: Unbedingt **Linux** als Typ und **Ubuntu** bei Version auswählen.
 3. Im folgenden Dialog müssen zunächst die Parameter **Speichergröße** und **Virtuelle Festplattengröße** festgelegt werden
 4. Festlegung der übrigen Parameter. Hierzu die zuvor erstellte VM in in der Liste auswählen und dann auf **Ändern klicken**. Eine Übersicht über alle Parameter ist in [Tabelle 1](#) dargestellt.
5. Starten der VM. In dem geöffneten Fenster "Medium für Start auswählen" muss jetzt de zuvor heruntergeladene Ubuntu oder XUbuntu **.iso** Datei ausgewählt werden.
6. Auswählen von **Install Ubuntu**. Im InstallWizard müssen folgende Schritte durchgeführt werden:
 1. Auswählen von Install Ubuntu und **continue**
 2. Auswahl des korrekten Tastaturlayouts. Am einfachsten geht das über die **Detect Keyboard** Funktion. Danach **continue**
 3. (Nur bei Ubuntu) **minimal installation** und **continue**
 4. **erase disk and install ubuntu**.
 5. **Jetzt installieren** → **continue**
 6. Im Menü **Who are you** müssen die Felder ausgefüllt werden. Als Benutzername sollte **psaf**, als Passwort **letmein** gewählt werden.
7. Nach der Installation ein Terminal öffnen (**Strg + ALT + T**) und folgenden Befehl ausführen:

```
sudo apt-get update && sudo apt-get upgrade
```

Falls das Fenster in der VM nur sehr klein dargestellt wird, können folgende Schritte ausgeführt werden, um es an die Displaygröße anzupassen:

Für Ubuntu:

1. Terminal öffnen (**Strg + Alt + T**)
2. Ausführen von:

```
sudo apt-get update
sudo apt-get install build-essential gcc make perl dkms
reboot
```

3. Nach dem Neustart im Menü von Virtualbox auf "Geräte" → "Gasterweiterungen einlegen" klicken
4. Der Installationswizard öffnet sich automatisch. Diesem muss gefolgt werden.
5. Neustart.

Für XUbuntu:

1. Terminal öffnen (**Strg + Alt + T**)
2. Ausführen von:

```
sudo apt-get update
sudo apt-get install build-essential gcc make perl dkms
reboot
```

3. Nach dem Neustart im Menü von Virtualbox auf "Geräte" → "Gasterweiterung einlegen" klicken.
4. Termin öffnen und ausführen von:

```
sudo /media/psaf/<guest_addition_version>/VBoxLinuxAdditions.run
```

5. Neustart

Table 1. Parameter der VM

Parameter	Wert
Speichergröße	4096 MB
Virtuelle Festplattengröße	40 GB
Prozessoren	2
Grafikspeicher	128 MB
USB	USB-3.0 Controller

Anschließend müssen die benötigten Pakete installiert werden. Dies ist im Abschnitt [Einrichtung](#) beschrieben.

WSL

Das [Windows Subsystem für Linux](#) (WSL) ermöglicht es, ein Linux Subsystem in die Windows Umgebung zu integrieren und mit diesem zu interagieren. Der Vorteil von WSL ist, dass der typische Overhead, der bei Verwendung von VMs anfällt, nicht existiert. Um WSL nutzen zu können muss mindestens Windows 10, Version 2004 installiert sein. Die Installationsanleitung ist [hier](#) zu finden. Nach der Installation muss noch Ubuntu 20.04 aus dem [Microsoft Store](#) installiert werden. Die Interaktion mit Ubuntu erfolgt dann entweder über das Terminal (Ubuntu 20.04 in die Suchleiste eingegeben) oder direkt über die IDE. Um die WSL in die IDE zu integrieren, stellen [CLion](#) und [VSCode](#) Tutorials zur Verfügung.

Nach der erfolgreichen Installation von Ubuntu müssen noch die benötigten Pakete installiert werden. Dies ist im Abschnitt [Einrichtung](#) beschrieben.

Auto Installation

Die Autos sind bei Übergabe an die Studierenden fertig eingerichtet. Sollte es dennoch erforderlich sein das Auto neu aufzusetzen, können folgende Schritte befolgt werden.

Die Installation auf dem Auto erfolgt durch ein fertiges Script. Dieses Script ist [Images Repo](#) zu finden. Hierfür einfach das Repo klonen und den Befehl

```
./scripts/install-car.sh
```

ausführen. Mehr Informationen sind in der [Dokumentation](#) des Repos zu finden.

Zur Installation der benötigten Pakete bitte dem Abschnitt [Einrichtung](#) folgen.

Einrichtung

Installation benötigter Pakete



Dieser Schritt entfällt bei Verwendung der eingerichteten VM und beim Auto, da hierbei die Pakete direkt installiert wurden.

Um mit der Entwicklung beginnen zu können, müssen zunächst einige benötigte Pakete installiert werden. Hierfür stehen wieder zwei Möglichkeiten zur Verfügung:

Automatische Installation

Das Repository [Images](#) stellt ein Installationsscript für die Einrichtung bereit. Eine Anleitung kann im Repository gefunden werden.

Manuelle Installation

ROS 2 Foxy

Die Installation von ROS2 Foxy ist am einfachsten als Binary Paket. Hierfür einfach der [Anleitung](#) folgen.

Nach der Installation sollte ROS noch in der `bashrc` Datei gesourced werden, damit dies nicht jedes Mal, wenn ein neues Terminal geöffnet wird, geschehen muss.

```
gedit ~/.bashrc
```

Am Ende des Editors dann folgendes einfügen:

```
source /opt/ros/foxy/setup.bash
```

Nach der Eingabe muss das Terminal dann neu gestartet werden oder der Befehl

```
source ~/.bashrc
```

einggegeben werden. Dies ist nötig, damit ROS2 Foxy im Terminal verwendet werden kann.

Realsense Kamera

Für die Installation der Realsense Abhängigkeiten kann der folgende Befehl genutzt werden:

```
sudo apt-get install ros-foxy-realsense2-camera
```

libpsaf

Die [libpsaf](#) bildet die Grundlage für die Entwicklung. Sie stellt die benötigten Interfaces, Subscriber und Publisher bereit. Die Installation kann mittels eines Debian Pakets oder manuell durchgeführt werden.

Installation mittels Debian Paket

1. Download der Pakete für die `libpsaf` (`ros-foxy-libpsaf_2.0.3-0focal_amd64.deb`) und der `libpsaf_msgs` (`ros-foxy-libpsaf-msgs_2.0.3-0focal_amd64.deb`) aus dem [Release](#) Abschnitt des Repository. Falls es bereits eine neuere Version der `libpsaf` gibt, ist diese zu wählen.
2. Installation der Pakete:

```
sudo dpkg -i ros-foxy-libpsaf-msgs_3.0.1-0focal_amd64.deb
sudo dpkg -r ros-foxy-libpsaf_3.0.1-0focal_amd64.deb
```



Die **libpsaf_msgs** müssen vor der **libpsaf** installiert werden.

Manuelle Installation

Falls eine bestimmte Version der **libpsaf** benötigt wird oder die automatische Installation fehlschlägt, kann die **libpsaf** auch manuell installiert werden.

1. Klonen des **Libpsaf** Repositorys

```
git clone https://git-ce.rwth-aachen.de/af/library.git
```

2. Installation der library

```
cd ~/library
colcon build --symlink install
```

3. Source der Installation

```
source install/local_setup.bash
```

Falls die **libpsaf** nicht jedes Mal beim Öffnen einer Konsole erneut gesourced werden soll, kann dies auch über die `bashrc` Datei gemacht werden.

```
echo "source ~/library/install/local_setup.bash" >> ~/.bashrc
```

uc_bridge

Die **uc_bridge** wird für die Kommunikation zwischen dem Hauptrechner und dem **uc_board** benötigt. Die Installation kann mittels Debian Paket oder manuell erfolgen.

Installation mittels Debian Paket

1. Download der Pakete für die **uc_bridge** (`ros-foxy-psaf-ucbridge_2.1.1-0focal_amd64.deb`) und der **ucbridge_msgs** (`ros-foxy-psaf-ucbridge-msgs_2.1.1-0focal_amd64.deb`) aus dem [Release](#) Abschnitt des Repository. Falls es bereits eine neuere Version der **uc_bridge** gibt, ist diese zu wählen.
2. Installation der Pakete:

```
sudo dpkg -i ros-foxy-psaf-ucbridge-msgs_2.1.1-0focal_amd64.deb
sudo dpkg -r ros-foxy-psaf-ucbridge_2.1.1-0focal_amd64.deb
```



Die `ucbridge_msgs` müssen vor der `uc_bridge` installiert werden.

Manuelle Installation

Falls eine bestimmte Version der `uc_bridge` benötigt wird oder die automatische Installation fehlschlägt, kann die `uc_bridge` auch manuell installiert, werden.

1. Klonen des `uc_bridge` Repositorys

```
git clone https://git-ce.rwth-aachen.de/af/psaf_ucbridge.git
```

2. Installation der `psaf_ucbridge`

```
cd psaf_ucbridge  
colcon build --symlink install
```

3. Sourcen der Installation

```
source install/local_setup.bash
```

Das sourcen kann erneut in der `bashrc` Datei gemacht werden.

```
echo "source ~/psaf_ucbridge/install/local_setup.bash" >> ~/.bashrc
```

Entwicklungsumgebung

Clion

CLion ist eine C/C++ IDE von JetBrains, die ROS2 unterstützt. CLion kann mit der Windows WSL genutzt werden. Die Anwendung ist für Studierende kostenfrei. Die Registrierung um eine kostenfreie Lizenz zu bekommen erfolgt [hier](#).

Nach der erfolgreichen Installation muss der Workspace geöffnet werden. Anschließend kann CLion eingerichtet werden:

Installation der PlugIns:

1. "File" → "Setting" → "PlugIns". Folgende PlugIns sollten ausgewählt werden:
 - AsciiDoc
 - ROS Support
 - (Kite) hilfreiche Erweiterung für Codevervollständigung
2. Falls eine WSL genutzt wird, müssen folgende Schritte durchgeführt werden:

Verbindung mit WSL

Dieser Schritt ist bei Verwendung auf dem Auto oder innerhalb der VM nicht nötig.

1. "File" → "Settings" → "Build, Execution, Deployment" → "Toolchain"
2. WSL an Anfang der Liste stellen

Eine ausführliche Anleitung gibt es auch auf der Website von [Clion/JetBrains](#)

Zum Ausführen von Code in der WSL in CLion muss ein neues Terminal geöffnet werden. Über die Terminal-Auswahlleiste muss "Ubuntu 20.04" ausgewählt sein.

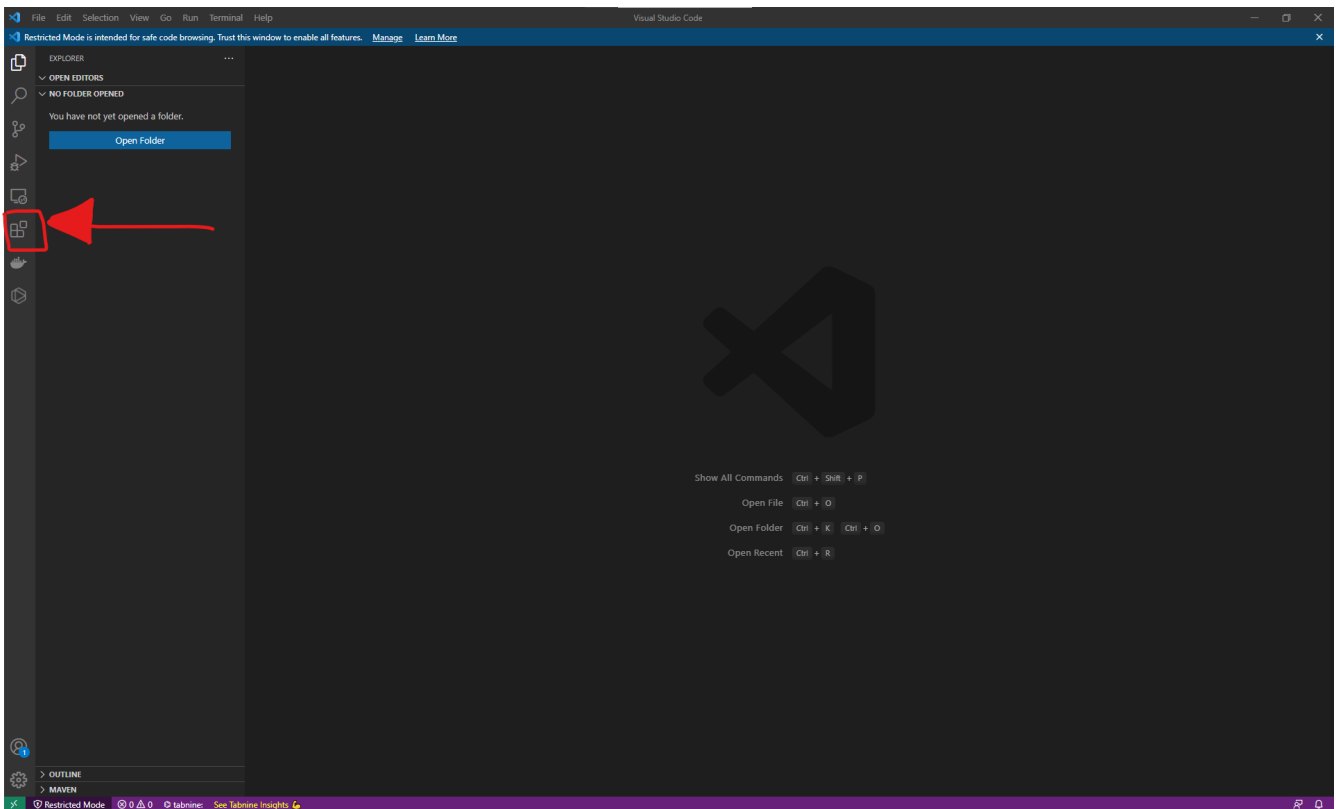
Visual Studio Code

Visual Studio Code ist eine Alternative zu CLion und unter Linux, Windows und Mac ausführbar. VsCode besitzt ebenfalls hilfreiche PlugIns für die Entwicklungen im Rahmen dieses Seminars.

Installation der PlugIns

Die PlugIn Installation erfolgt direkt in VsCode, indem man auf im Bild markierte Symbol klickt. Folgende Plugins sollten installiert werden:

- AsciiDoc
- ROS
- (Kite) - hilfreiche Erweiterung für Code Vervollständigung



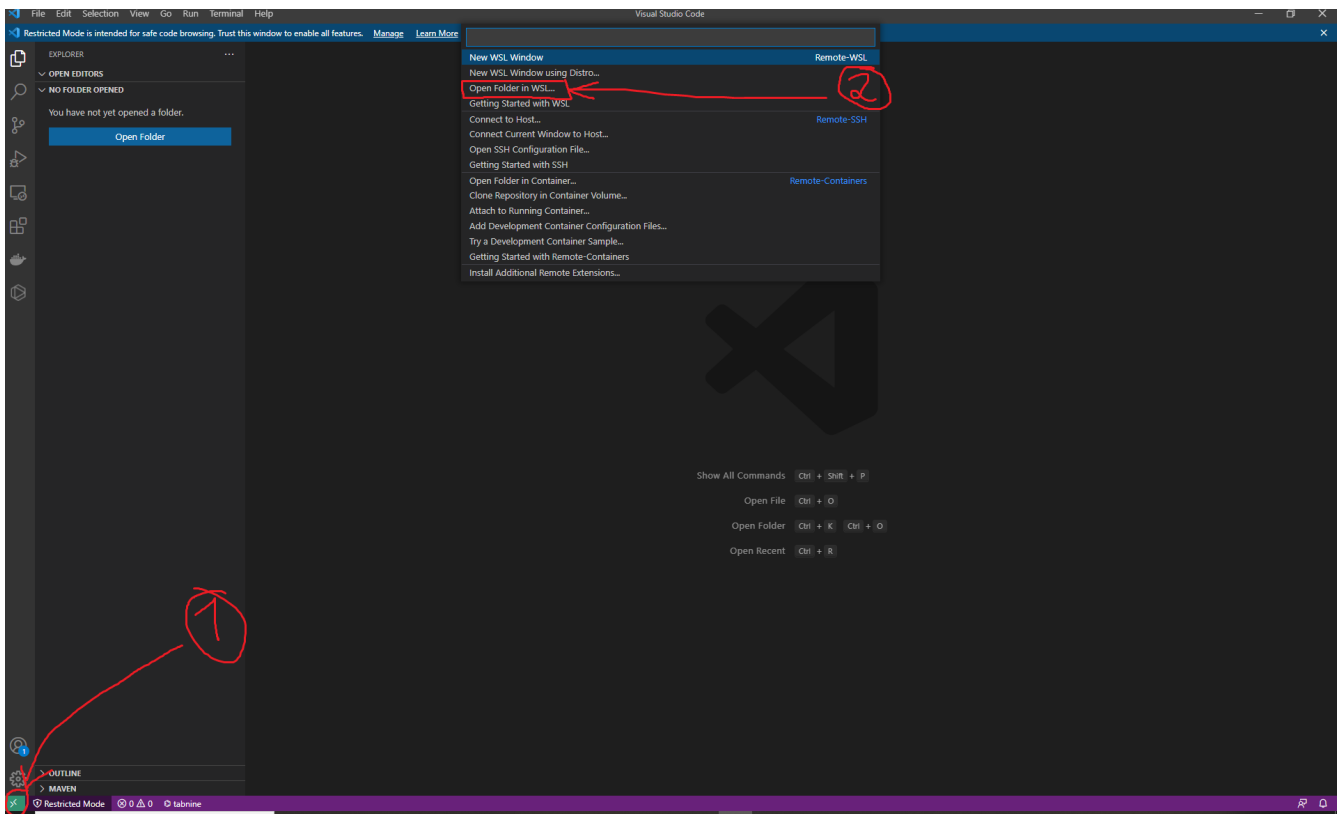
Verbindung mit WSL

Um Code mit WSL ausführen zu können, müssen folgende Schritte ausgeführt werden:

1. Installation des [Remote Development Extension Packs](#)
2. Workspace in VSCode öffnen
3. Das Symbol in der linken unteren Ecke klicken (siehe Bild)
4. "Open Folder in WSL" auswählen und kurz warten.

Eine ausführliche Anleitung findet sich [hier](#)

Zum Ausführen von Code in der WSL in VSCode muss ein neues Terminal geöffnet werden. Über die Terminal-Auswahlleiste muss "Ubuntu 20.04 (WSL)" ausgewählt sein.



Bekannte Probleme und FAQ

Bauen des Workspaces

Fehlermeldung: Fehlende **libpsaf** oder **uc_bridge**

Diese Fehlermeldung kann mehrere Ursachen haben. Zuerst sollte überprüft werden, ob die **libpsaf** sowie die **uc_bridge** auf dem Computer vorhanden sind und gebaut wurden.

Über die Ausgabe folgender Befehle kann dies überprüft werden:

```
find library/install  
find psaf_ucbridge/install
```

Falls die Ausgabe **find: 'libpsaf/install': No such file or directory** oder **find:**

'psaf_ucbridge/install': No such file or directory in der Konsole steht, so ist das entsprechende Paket nicht installiert. Eine Installationsanleitung für die beiden Pakete ist im Abschnitt [Setup](#) zu finden.

```
adrian@adrian-pc:~/ws-template$ colcon build
Starting >>> psaf_configuration
Starting >>> psaf_launch
Finished <<< psaf_configuration [0.50s]
Starting >>> psaf_controller
Starting >>> psaf_lane_detection
Starting >>> psaf_manual_mode
Finished <<< psaf_launch [0.57s]
Starting >>> psaf_object_detection
--- stderr: psaf_manual_mode
CMake Error at CMakeLists.txt:24 (find_package):
  By not providing "Findlibpsaf.cmake" in CMAKE_MODULE_PATH this project has
  asked CMake to find a package configuration file provided by "libpsaf", but
  CMake did not find one.

Could not find a package configuration file provided by "libpsaf" with any
of the following names:

  libpsafConfig.cmake
  libpsaf-config.cmake

Add the installation prefix of "libpsaf" to CMAKE_PREFIX_PATH or set
"libpsaf_DIR" to a directory containing one of the above files.  If
"libpsaf" provides a separate development package or SDK, be sure it has
been installed.

---
Failed <<< psaf_manual_mode [4.45s, exited with code 1]
Aborted <<< psaf_controller [4.52s]
Aborted <<< psaf_object_detection [4.40s]
Aborted <<< psaf_lane_detection [4.50s]
Summary: 2 packages finished [5.31s]
 1 package failed: psaf_manual_mode
 3 packages aborted: psaf_controller psaf_lane_detection psaf_object_detection
 4 packages had stderr output: psaf_controller psaf_lane_detection psaf_manual_mode psaf_object_detection
 7 packages not processed
adrian@adrian-pc:~/ws-template$
```

Figure 7. Fehlermeldung libpsaf

Falls beide Ordner gefunden werden, kann der Fehler durch das fehlende "sourcen" der Pakete versucht werden. In diesem Fall muss man folgende Befehle ausführen:

```
cd ~/library/
source install/local_setup.bash
cd ..
cd ~/psaf_ucbridge/
source install/local_setup.bash
```

Dieser Schritt muss jedes Mal wiederholt werden, wenn ein neues Konsolenfenster geöffnet wird. Um dies zu vermeiden, kann der Befehl auch in die `bashrc` Datei eingefügt werden. Hierfür muss die `bashrc` Datei zunächst geöffnet werden:

```
gedit ~/.bashrc
```

In der Datei die beiden source Befehle einfügen und speichern. Anschließend muss das Terminal neu gestartet werden.

Fehlermeldung: zbar.h - No such file or directory

Die **zbar** Bibliothek wird für die Erkennung des QR-Codes in der Startbox benötigt. Falls der build wegen der fehlenden **zbar.h** Abhängigkeit fehlschlägt, muss folgender Befehl ausgeführt werden.

```
sudo apt-get install libzbar-dev
```

Fehlermeldung: cv_bridge: No such file or directory

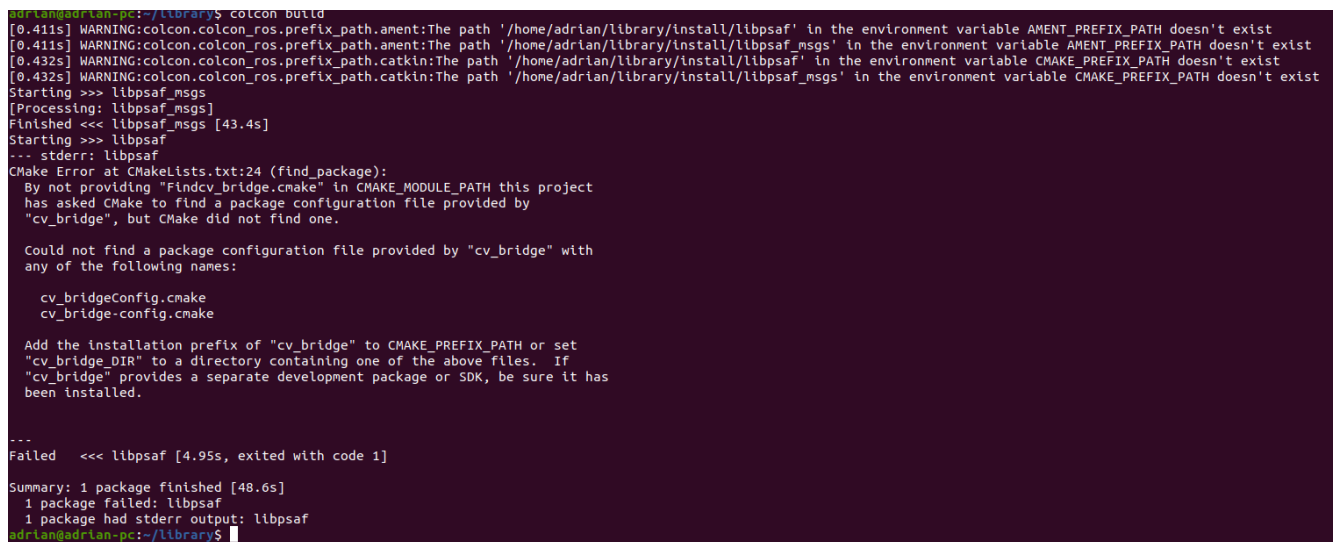
Bei der ROS2 Installation werden unter Umständen nicht alle benötigten Pakete installiert. Um die genannte Fehlermeldung zu beseitigen, können folgende Befehle ausgeführt werden:

```
cd ~/ros2_foxy/src
git clone https://github.com/ros-perception/vision_opencv
cd vision_opencv
git checkout ros2
colcon build --symlink-install
```

Anschließend muss die Installation noch gesourced werden:

```
source install/local_setup.bash
```

Alternativ kann der Befehl auch in die **bashrc** Datei eingefügt und ein neues Terminal geöffnet werden.



```
adrian@adrian-pc:~/library$ colcon build
[0.411s] WARNING:colcon.colcon_ros.prefix_path.ament:The path '/home/adrian/library/install/libpsaf' in the environment variable AMENT_PREFIX_PATH doesn't exist
[0.411s] WARNING:colcon.colcon_ros.prefix_path.ament:The path '/home/adrian/library/install/libpsaf_msgs' in the environment variable AMENT_PREFIX_PATH doesn't exist
[0.432s] WARNING:colcon.colcon_ros.prefix_path.catkin:The path '/home/adrian/library/install/libpsaf' in the environment variable CMAKE_PREFIX_PATH doesn't exist
[0.432s] WARNING:colcon.colcon_ros.prefix_path.catkin:The path '/home/adrian/library/install/libpsaf_msgs' in the environment variable CMAKE_PREFIX_PATH doesn't exist
Starting >>> libpsaf_msgs
[Processing: libpsaf_msgs]
Finished <<< libpsaf_msgs [43.4s]
Starting >>> libpsaf
-- stderr: libpsaf
CMake Error at CMakeLists.txt:24 (find_package):
  By not providing "Findcv_bridge.cmake" in CMAKE_MODULE_PATH this project
  has asked CMake to find a package configuration file provided by
  "cv_bridge", but CMake did not find one.

Could not find a package configuration file provided by "cv_bridge" with
any of the following names:

  cv_bridgeConfig.cmake
  cv_bridge-config.cmake

Add the installation prefix of "cv_bridge" to CMAKE_PREFIX_PATH or set
"cv_bridge_DIR" to a directory containing one of the above files.  If
"cv_bridge" provides a separate development package or SDK, be sure it has
been installed.

---
Failed <<< libpsaf [4.95s, exited with code 1]
Summary: 1 package finished [48.6s]
1 package failed: libpsaf
1 package had stderr output: libpsaf
adrian@adrian-pc:~/library$
```

Figure 8. Fehlermeldung cv_bridge

Fehlermeldung: "permission denied <filename>"

Falls in der CI-Pipeline die Fehlermeldung auftritt, dass der Zugriff auf ein Skript (.sh) oder eine Python-Datei (.py) nicht erlaubt ist, kann der Fehler wie folgt behoben werden. (Die Datei muss sich

bereits im GitLab Repository befinden.)

1. Öffnen eines Terminals
2. Eingabe:

```
git update-index --chmod=+x <filename>  
git commit -m "Changed permission"  
git push
```

Welche Schilder müssen erkannt werden?

Die Schilder sind von den Regularien des Carolo Cups vorgegeben. [Die Abbildung "Schilder"](#) zeigt die im Carolo Cup vorhandenen Schilder.

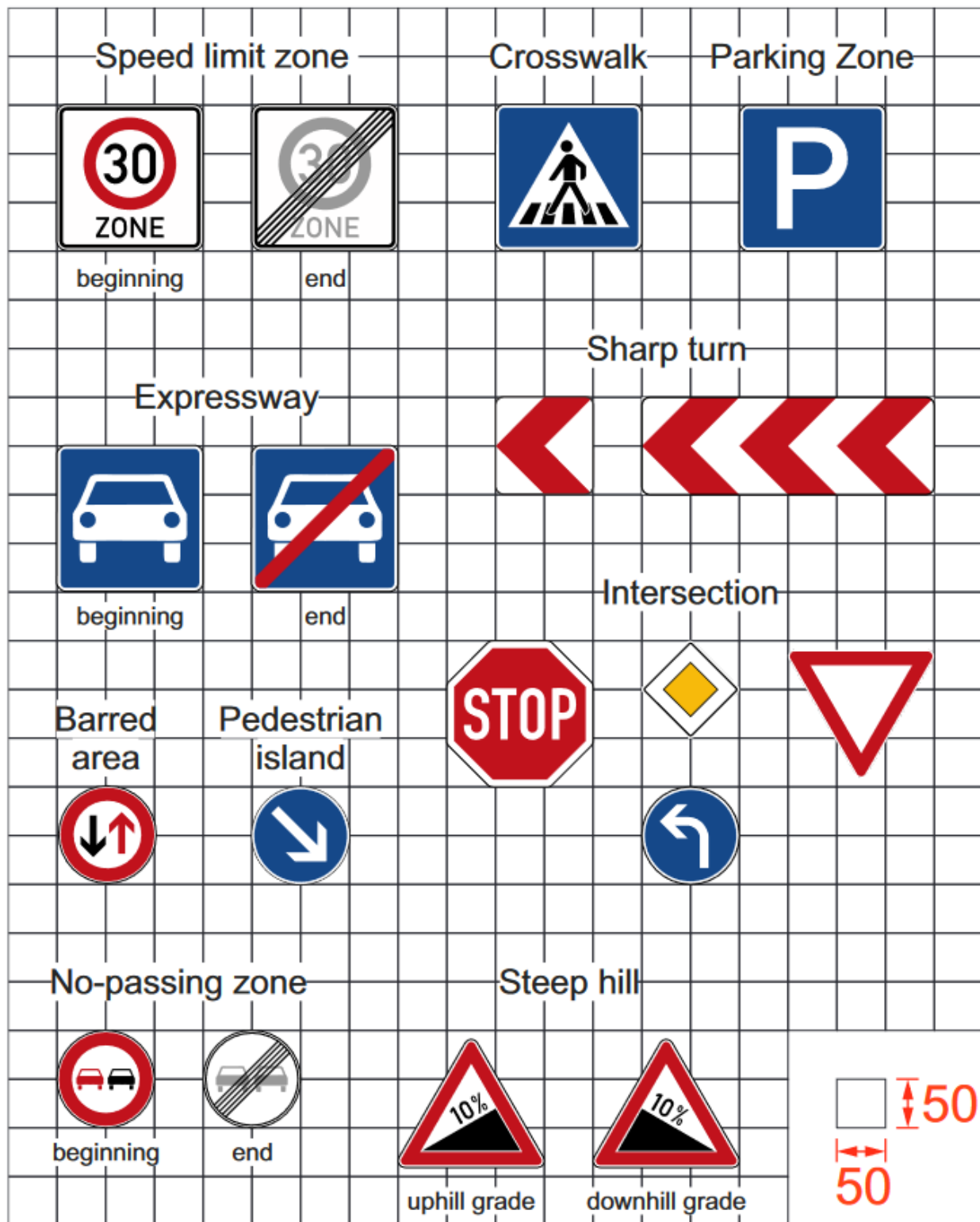


Figure 9. Die Schilder des Carolo Cups