
	<p>Professorship Computer Engineering Automotive Software Engineering Prof. Dr. Dr. h. c. Wolfram Hardt Dr. Batbayar Battseren</p>	
<p>Practical Unit 4</p>	<p>Designing ADAS functions with AUTOSAR</p>	<p>Summer Semester 2025</p>

Contents

1. Introduction	1
2. Development Environment	1
2.1. SystemDesk Template Project	1
2.2. Driving Simulator	2
3. Task Specification	4
4. Solution Approach	8
5. Tasks	9
5.1. Task 1: Setting up the simulation environment	9
5.2. Task 2: Implementing CruiseControl	9
5.2.1. Sub task 1: Basic CC logic with constant target speed	9
5.2.2. Sub task 2: Enable/disable CC	9
5.2.3. Sub task 3: Manual acceleration/braking	10
5.2.4. Sub-task 4: Manual definition of target speed	10
5.3. Task 3: Implementing EmergencyBrake	11

1. Introduction

Based on the knowledge about AUTOSAR and the usage of SystemDesk you gained in unit 3, in this unit you will build 2 Advanced Driver Assistance Systems with AUTOSAR. In addition, you will test your solution in a virtual driving simulator.



Before you begin working on this task, please read the **whole task description carefully**.

2. Development Environment

2.1. SystemDesk Template Project

Your task is to implement a simplified version of **Cruise Control** (German: Tempomat) and **Emergency Brake Assistant** (German: Notbremsassistent). The Cruise Control (CC) will accelerate a car to a target speed set by the driver and will then try to keep this speed (by either accelerating, braking or just letting the car roll). The Emergency Brake Assistant will initiate an emergency brake when the distance to a leading vehicle gets too close and a crash would be otherwise imminent.

You will integrate your solution in a **prepared SystemDesk project** where the **inputs** (i.e. required ports) and **outputs** (i.e. provided ports) for your SWC composition are already set up (cf. Figure 1). The project uses a nested composition where the **MainComposition** already contains

two Software Components which provide the inputs and outputs for the ADAS as shown in Figure 2. The RPorts and PPorts of this composition will be connected to the driving simulator described in Section 2.2 once you execute your solution in VEOS.

Notice, that the MainComposition also contains another SWC composition called AdasComposition which is already wired. Your solution needs to be integrated into this AdasComposition.



Do not change or add anything in the MainComposition.
All of your components should go into AdasComposition.

As with “normal” SWCs you can also add ports to a Composition SWC itself which provides an additional abstraction layer and a unified interface for the composition. These ports will be shown “freely floating” in the Composition diagram and can be placed as needed. If you open the AdasComposition it should look as shown in Figure 3. All SWCs you implement in this Unit should be placed in this AdasComposition and connected to the “virtual” ports. As such, you can focus on implementing the logic of the ADAS function without worrying about configuration and integration into the final system (i. e. everything around your logic).

Table 1 shows the pre-configured ports which can be used by your implementation and Table 2 shows the interfaces which have been pre-defined for you. All interfaces are of type Sender-Receiver and the datatypes are AUTOSAR data types (from the default AUTOSAR_Platform.arxml as used in Unit 3).

2.2. Driving Simulator

As in Unit 3, we will use VEOS as ECU simulator in order to test the AUTOSAR application. This means you need to create a Classic V-ECU which can be loaded in VEOS. When creating the V-ECU please choose the MainComposition as root.

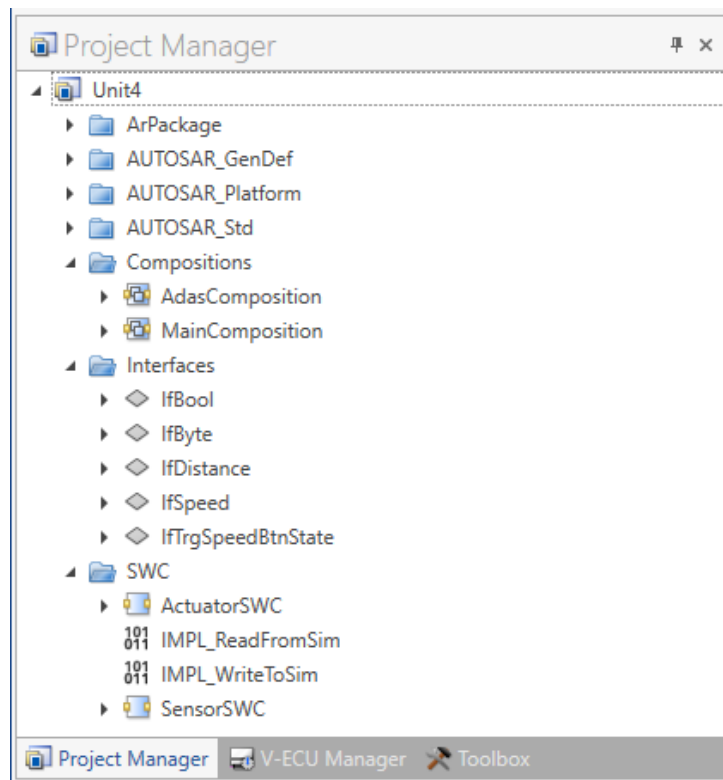


Figure 1: Structure of the template project

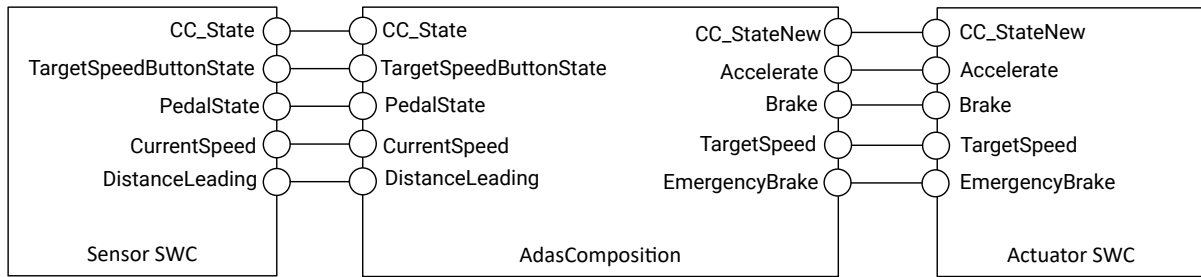


Figure 2: The MainComposition of the target AUTOSAR application which contains AdasComposition as **nested** composition

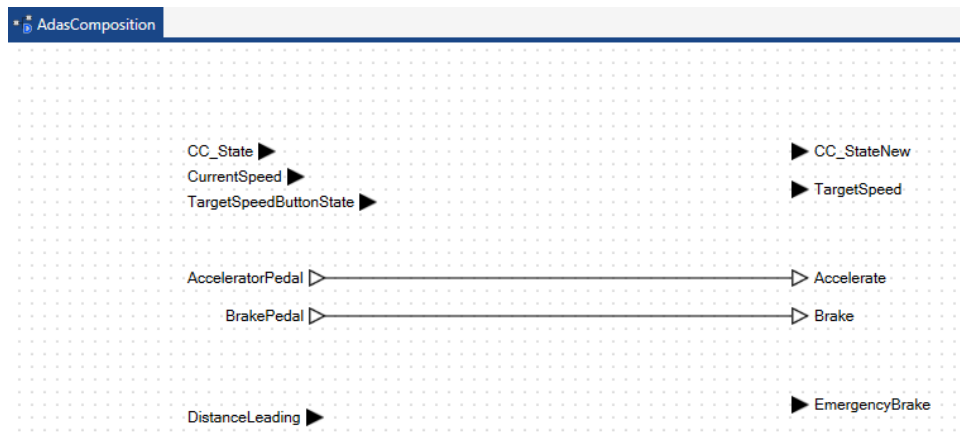


Figure 3: The initial state of the AdasComposition which needs to be implemented by you

In contrast to Unit 3 we will use a 2D virtual driving simulator to visualize the behavior of your solution. For this, make sure the simulator (CE Virtual Car 2D) has been started prior to launching the VEOS simulation. You should see a window similar to Figure 4.

When you launch the VEOS simulation your AUTOSAR composition will be integrated in-the-loop of the driving simulation. Consequently, sensor data will be forwarded to the required ports and outputs will be read from the provided ports of your SWC Composition. You can see the current values of sensors and actuators on the left side of the simulator window.

When the simulator is connected to your AUTOSAR application you can control the signals on the RPorts of the AdasComposition through the following keyboard keys (make sure that the simulator window has focus):

Keyboard Key	Connected to RPort
W	AcceleratorPedal
Space	BrakePedal
C	CC_State
Arrow Up	Sets TargetSpeedButtonState to 1
Arrow Down	Sets TargetSpeedButtonState to 2

In addition, pressing "L" on the keyboard will spawn a single vehicle at a random distance to the ego vehicle. Use it to test the emergency brake assistant to avoid crashes (see Figure 5).

Table 1: Pre-configured ports of the AdasComposition

Port Type	Port Name	Interface	Description
Req.	CC_State	IfBool	The current state of the CC system (= latest output on CC_StateNew)
Prov.	CC_StateNew	IfBool	The new state of the CC system (= passed to CC_State)
Req.	CurrentSpeed	IfSpeed	The vehicle's current speed
Req.	Accelerator-Pedal	IfBool	Indicates the current state of the accelerator pedal (i.e. if the driver wants to accelerate)
Req.	BrakePedal	IfBool	Indicates the current state of the brake pedal (i.e. if the driver wants to brake)
Req.	TargetSpeed-ButtonState	IfTrgSpeedBtnState	Determines the desired driver's action for adjusting the target speed of the CC system
Req.	DistanceLeading	IfDistance	The distance to a leading vehicle or obstacle (if there is any)
Prov.	Accelerate	IfBool	Indicates whether the vehicle should accelerate
Prov.	Brake	IfBool	Indicates whether the vehicle should brake
Prov.	TargetSpeed	IfSpeed	The current target speed as set by the driver (will be displayed in the HUD)
Prov.	Emergency-Brake	IfBool	Indicates whether an emergency brake should be initiated



Please ensure that the simulator shows "CONNECTED to VEOS" after starting the simulation in VEOS.

3. Task Specification

Your task is to implement 2 ADAS functions for the virtual car in the simulator: Cruise Control and Emergency Brake. For this the following requirements have to be met by your implementation:

Cruise Control (CC):

- While CC_State is on:
 - Accelerator and/or Brake should be set in order to achieve the set TargetSpeed
 - If the driver steps on the brake pedal, the CC must be disabled immediately
 - If the driver steps on the accelerator pedal, this overrides the CC (e.g. the car can accelerate above target speed) temporarily (but CC stays on)

Table 2: Pre-defined interfaces of the template project

Interface Name	Data Elements
IfBool	Value: boolean 0 = off / no 1 = on /yes
IfTrgSpeedBtnState	Value: uint8 1= increase target speed by 5 km/h 2 = decrease target speed by 5 km/h all other values = no action
IfSpeed	Value: float32 a speed value in km/h
IfDistance	Value: float32 a distance value in meters

- If the driver releases the accelerator pedal, CC should regulate the speed again
- If CC_State is off, no actions should be taken with regards to reaching TargetSpeed but the driver can accelerate and brake manually
- Based on the value of TargetSpeedButtonState the TargetSpeed should be increased or decreased (or remain unchanged), independently of CC_State
 - The initial TargetSpeed should be **10 km/h**
 - The maximum TargetSpeed value for CC has to be **130 km/h**.

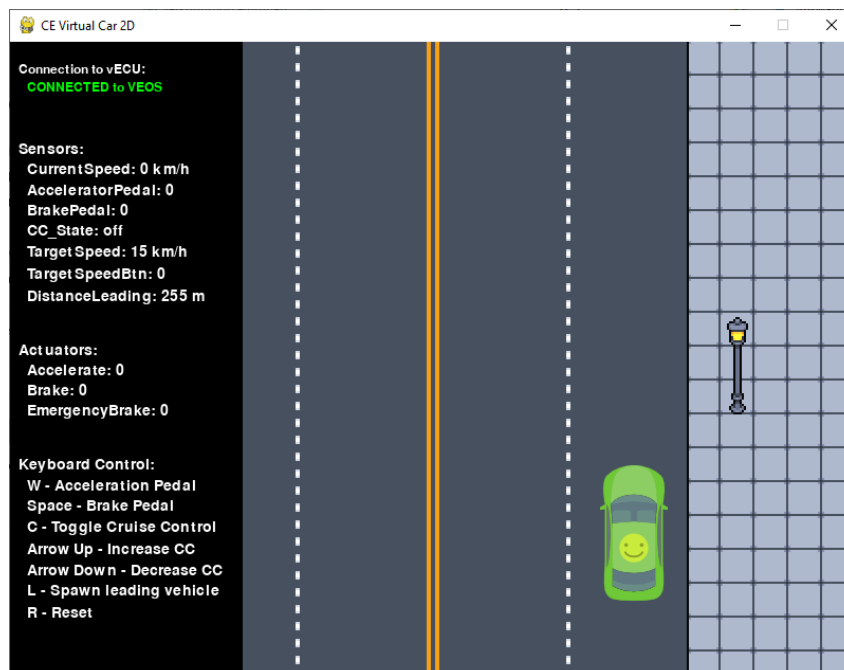


Figure 4: Screenshot of the running driving simulator

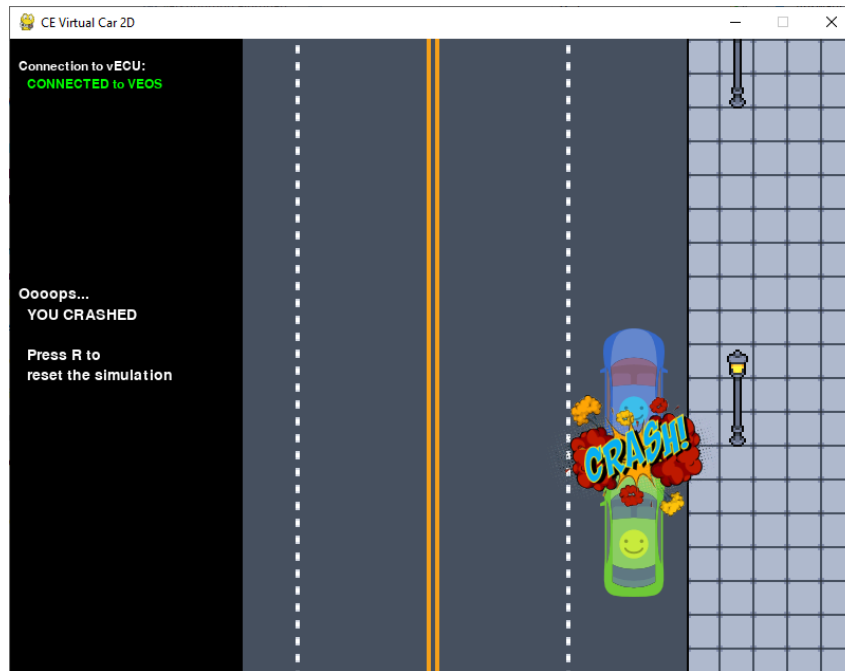


Figure 5: A crash in the simulator



Keep in mind that a car will automatically slow down if neither acceleration nor braking are initiated due to road friction and the engine brake. This behavior is replicated in the simulator.

Emergency Brake:

- If the distance to a leading vehicle is too small to continue driving further, an automatic emergency brake should be initiated (i.e. EmergencyBrake should be set to 1)
- As part of this, if the CC is currently on it should be deactivated immediately
- After an emergency brake, when the car has stopped, the EmergencyBrake signal should revert to 0 again

You will use multiple software components to implement your solution. By splitting functionality into more SWCs the individual logic of each SWC usually becomes easier and more decoupled. For the SWCs, you have to create Ports as required.



Remember, that you can only connect ports which have been assigned the same interface.

The logic of your SWCs should be implemented in single c-file based Runnables (one file and function per SWC). As SystemDesk is not an IDE you will use Visual Studio Code¹ to edit the code files as in Unit 3. Remember, that in order to access ports in code you have to set up the correct Data Access options. Keep in mind, that the naming of the methods, header files and generated RTE functions will depend on the names for SWCs and Ports you choose (cf. Listing 1 as reference).

¹When opening the project's root folder in Visual Studio Code, you gain access to code completion once you have built the V-ECU in SystemDesk

```

1  #include <Rte_MySWC.h>
2  #include <Sab.h>
3
4  void MySWC_MyRunnable()
5  {
6      // SWC logic here
7  }

```

Listing 1: Basic structure of a Runnable code file for a software component called MySWC and the Runnable called MyRunnable

As in Unit 3 you can use the function Sab_SubmitInfo for outputting debugging information in the console of VEOS. For reading and writing ports you should use the implicit read and write methods (i.e. buffered). Please remember the naming conventions of AUTOSAR from the lecture to correctly build the function names:

for reading a port: `Rte_IRead_[RUNNABLE]_[PORT]_[DATA_ELEMENT]()`

for writing a port: `Rte_IWrite_[RUNNABLE]_[PORT]_[DATA_ELEMENT](value)`

In addition, for each runnable you will need to set up one or more RTE events which trigger the execution. Please choose an appropriated type (e.g. time-triggered or triggered by a data-received-event) based on your implemented logic.



A single **Data Received Event** can only listen for data on one port. If you want to trigger a runnable for multiple ports you need to add a separate Data Received Event for each port.



You **should not** try to implement all functionalities at once. By choosing a step-wise approach you can ensure that basic functionality works before you continue.

4. Solution Approach

Your main approach for implementing the **AUTOSAR task of Unit 4** should be following the AUTOSAR workflow known from the lecture (as shown in Figure 1). In its essence, it is a representation of the V-Model development approach. For this you should divide the complete task into sub-tasks which can be completed one after the other where each consecutive task builds on the successful completion of its predecessors. For each sub-task you need a clear requirement when it is finished. Consequently, you should start with designing the architecture of the first sub-task. If you have finished the design, you can continue with implementing the behavior and testing it. As long as your implementation does not fulfill the requirements of your current you need to extend the implementation and re-test it in a continuous cycle. If you are satisfied with the current state you can continue to the next sub-task and repeat the 3 steps design, implementation, test.

In terms of concrete steps when working on an AUTOSAR implementation of an application each of the 3 main steps comprises the following sub-steps:

Step 1 - **Design Phase, SystemDesk**

- Specify a SWC and its name
- Add required ports & interfaces to the SWC, remember to configure Data Access
- Integrate the SWC to the target composition and connect the corresponding ports
- Add Internal Behavior + Runnable definition
- Generate the RTE (by creating a V-ECU)

Step 2 - **Implementation, Visual Studio Code**

- When the RTE has been generated, VS Code will provide you with code completion for the Rte_IRead and Rte_IWrite methods
- Implement the logic in C for your sub-task
- Once completed, click on "Build" in SystemDesk to integrate your SWC's code into the V-ECU

Step 3 - **Test, VEOS & CE VCarSim**

- Import your V-ECU build in Veos Player
- Click on "Run" in the simulation tab to execute your AUTOSAR application
- You can now test the behavior in the **2D driving simulation CE VCarSim**

If your implementation for a sub-step exhibits the correct behavior you can continue to the next sub-task. If not, go back to Step 2 and try to improve your implementation. Should it turn out that changes to the SWC's architecture are necessary, go back to Step 1. Remember that you need to **delete the old V-ECU in SystemDesk** (ArPackage) and create a new one every time you changed the design and want to continue to Step 2.

5. Tasks

5.1. Task 1: Setting up the simulation environment

The first thing to do is to **build and execute** the provided SystemDesk project in VEOS. It has been configured in such a way that you should be able to manually control acceleration and braking with the keyboard keys W and Space since in the initial AdasComposition the pedal states are passed through to the Accelerate and Brake signal (cf. Figure 3). Once you find that the connection of VEOS and the 2D simulator is working you can start with Task 2. In order to implement your custom logic you need to **remove these connections** and rewire the corresponding signals to the SWCs you develop.

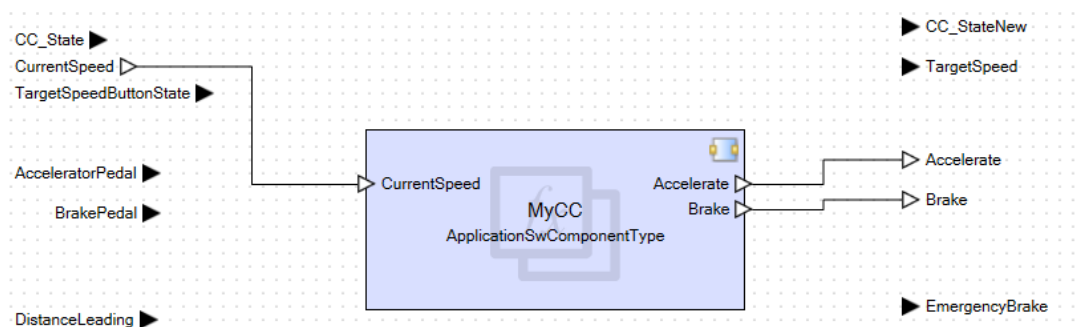
5.2. Task 2: Implementing CruiseControl

As mentioned, in order to implement the complex function of Cruise Control we will split the task into smaller sub tasks. For each of these sub-tasks the AUTOSAR workflow of Design, Implementation and Testing has to be followed individually. You should only move to the next sub-task if the current task was tested successfully.

In terms of Unit 4 a possible (but by far not the only) division into sub-tasks could look as follows. For each step, create the SWC, Ports, RTE events (i.e. triggers), Interface Assignments as well as Internal Behaviors you need for the implementation as necessary.

5.2.1. Sub task 1: Basic CC logic with constant target speed

Add one SWC to AdasComposition which accelerates the simulated car to a constant speed (e.g. 50 km/h) and keeps it there. You need to handle data on the CurrentSpeed port for this and provide the corresponding output on the Accelerator and Brake PPorts of the composition. This behavior should be independent of CC_State (i.e. always on). Start with a component which only has the bare minimum of ports you need to implement the subtask. If you tested it successfully, extend the port definitions according to the next subtask. Consequently, the design for the first sub task could look **similar** to this:



5.2.2. Sub task 2: Enable/disable CC

Add enabling / disabling of the Cruise Control by handling data received on the CC_State port (pressing C in the simulator). This means automatic acceleration / braking should only occur if CC is on.

Make sure to output the current state of the CC to the CC_StateNew port as this will become the next input on CC_State.

5.2.3. Sub task 3: Manual acceleration/braking

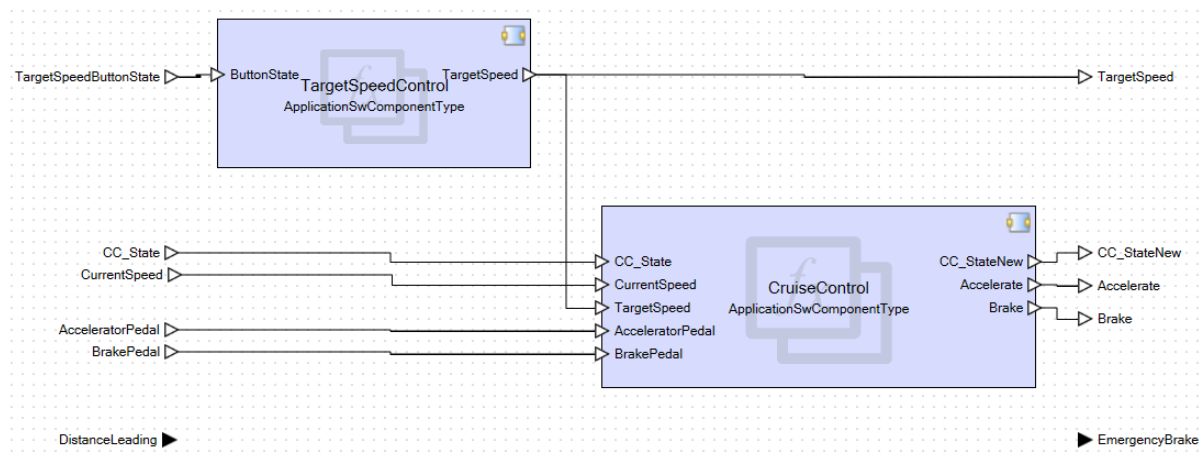
Add the following behavior to your existing solution:

If the CC is disabled, the accelerator and brake pedal should directly control the Accelerate and Brake signals (pressing W or Space in the simulator). If CC is on, pushing the accelerator pedal should accelerate the car over the target speed. If the pedal is released CC should return the car to target speed. If CC is on and the brake pedal is pushed, CC should be disabled immediately.

5.2.4. Sub-task 4: Manual definition of target speed

Add a second SWC which handles increasing / decreasing the CC target speed (pressing arrow up / down in the simulator) and provides this speed to the CC SWC. This means you need to handle the TargetSpeedButtonState port in this SWC and output the new target speed. You can then connect this to a new TargetSpeed port of your CruiseControl SWC to dynamically retrieve the target speed instead of using a constant one.

The final AdasComposition for Task 2 could look similar to the following figure:



5.3. Task 3: Implementing EmergencyBrake

Now, **add a third SWC** which realizes the Emergency Brake behavior. Think about if this SWC needs any connections to the other SWCs and implement these accordingly.

Tip: You can calculate the distance a car needs to brake using the following equation:

$$d_{\text{brake}} = \frac{\text{current_speed}^2}{2 * \text{deceleration}}$$

For the simulated car the following values apply:

$$\text{deceleration}_{\text{brake}} = 10 \frac{m}{s^2}$$

$$\text{deceleration}_{\text{emergencybrake}} = 14 \frac{m}{s^2}$$

Consequently, an emergency brake should occur if there would not be enough distance left for the driver to brake manually.