

# Graphische Programmierung & Simulation

## Programmmentwurf ASCET

Nick Dressler (6870655)

Ruben Hartenstein (2746235)

DHBW-Stuttgart Fakultät Technik

Graphische Programmierung

& Simulation im 6. Semester

Dozent: Kai Pinnow

## I. TrafficLight Module (R1, R2, D3)

Um die Ampeln zu realisieren, wurde eine *TrafficLight*-Klasse (siehe Abbildung 1) erstellt. Diese beinhaltet den Zustand ihrer Lichter, sowie ihre Sichtbarkeit, Position und die Entfernung zum Auto als private Attribute (Zugriff auf private Attribute mittels Getter/Setter, um das Geheimnisprinzip zu erfüllen). Hier dargestellt sind die nichttrivialen Methoden, die für die Ausführung des Experiments benötigt werden.

```

1 package environment;
2 import resources.m;
3
4 class TrafficLight {
5     private boolean red = false;
6     private boolean yellow = false;
7     private boolean green = false;
8     private boolean isVisible = false;
9     private m proximity = 0.0[m];
10    private m position = 0.0[m];
11
12    /**
13     * Sets the position of the traffic light on the track in meters.
14     * Initially set.
15     * Also sets the proximity to the car, because it starts at 0 meters.
16     * After the proximity is updated, the function updateVisibility() is called.
17     *
18     * @param pos Position of traffic light in meters
19     */
20    public void setPosition(m pos) {
21        this.position = pos;
22        this.proximity = pos;
23        this.updateVisibility();
24    }
25
26    /**
27     * To comply with R2, a traffic light should only be visible within 100 meters of the car.
28     * The boolean isVisible describes this feature.
29     *
30     * Updates Boolean isVisible.
31     */
32    private void updateVisibility() {
33        this.isVisible = between(this.proximity, 0.0[m], 100.0[m]);
34    }
35
36    /**
37     * Updates the proximity to the car, based on the position of the car and its own position.
38     * After the proximity is updated, the function updateVisibility() is called.
39     *
40     * @param carElapsed Position of the car on the track in meters.
41     */
42    public void updateProximity(m carElapsed) {
43        this.proximity = this.position - carElapsed;
44        this.updateVisibility();
45    }
46
47 }

```

Abbildung 1: Fachliche Klasse *TrafficLight* aus *TrafficLigh.esdl*:

Über die *deltaT*-Message, als globale Zeit, wird in der Datei **Light.bd** eine lokale Variable *elapsedTime* aktualisiert, welche die vergangenen Sekunden hält, damit also stets im Intervall [0s; 59s] liegt (siehe Abbildung 2).

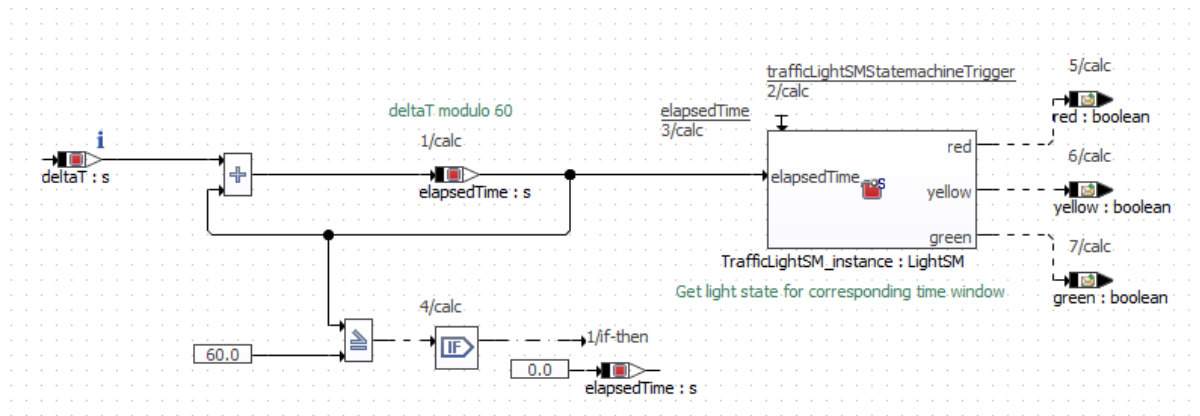


Abbildung 2: Ansteuerung für Light Statemachine mit lokaler Zeit aus *Light.bd*

Die lokale Variable `elapsedTime` wird der Light Statemachine als Parameter übermittelt. Auf Basis dieser werden die Ampelphasen ermittelt und die korrespondierenden Lichtzustände gesetzt und zurückgegeben (siehe Abbildung 3).

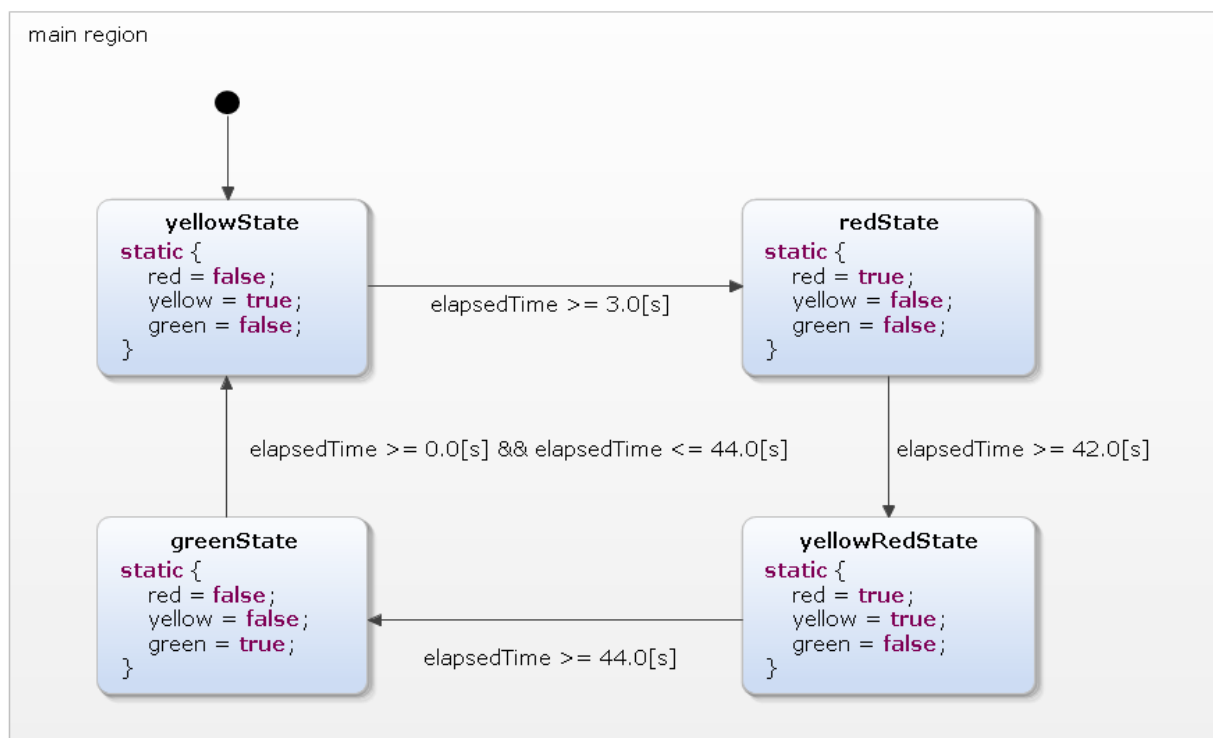


Abbildung 3: Light Statemachine aus *LightSM.sm*

Ein zentraler *TrafficLightController* (siehe Abbildung 4) hält drei Instanzen der *TrafficLight*-Klasse, initialisiert diese mit ihrer entsprechenden Position (300.0 [m], 500.0 [m], 900.0 [m]) und updatet ihre Licht-Zustände sowie die Position des Autos auf der Runde. Immer wenn die Position des Autos geupdatet wird, wird die neue *proximity* der Ampeln berechnet und der Parameter *isVisible* aktualisiert. Sobald die *proximity* zwischen 0m – 100m liegt, ist *isVisible* true, sonst false.

Ist eine der Ampeln sichtbar (Im Modell sowie in der Welt immer nur eine Ampel realistisch) ist es dem *driver* via *TrafficLightMessages* möglich, auf die *proximity* zuzugreifen und die Lichtzustände der Ampeln auszulesen (siehe Abbildung 16).

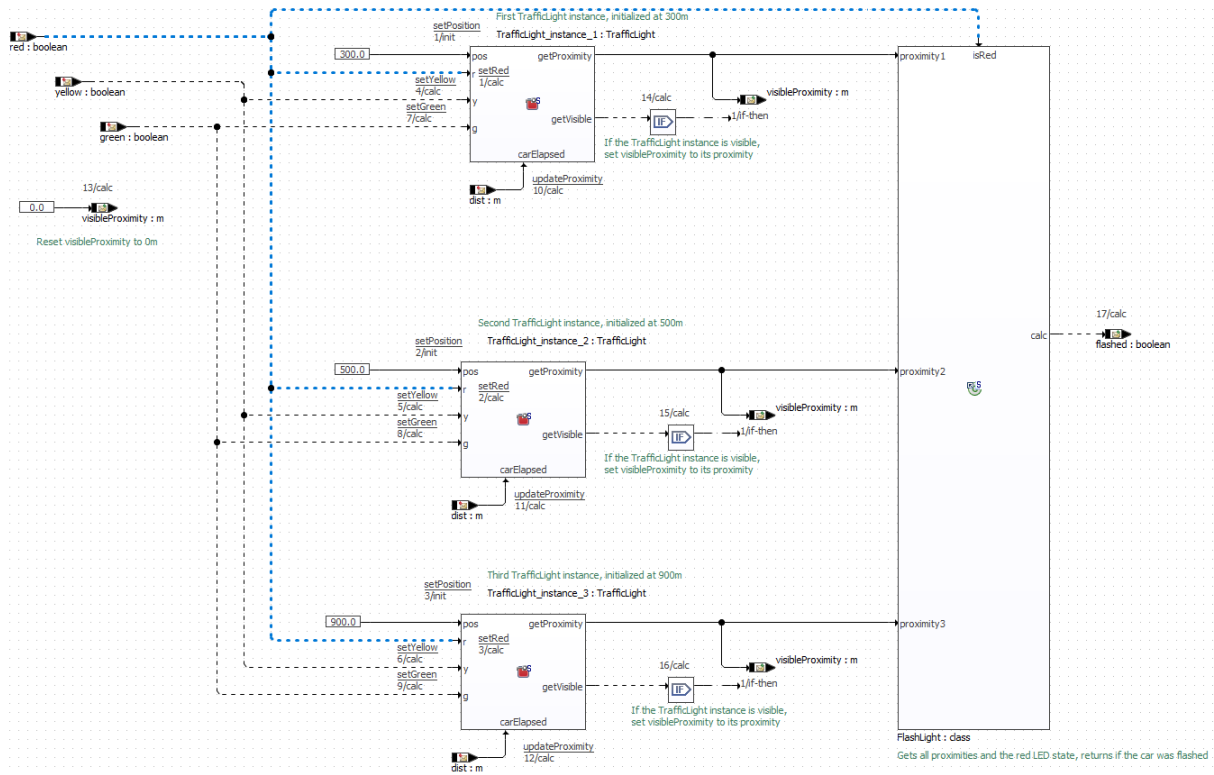


Abbildung 4: Steuerung der Ampeln und des Blitzers aus TrafficLightController.esdl

## II. Car (R3, R1)

Das Auto startet bei Sekunde 0 bei Distanz 0 Meter. Mittels des *drivers* wird verhindert, dass über eine rote Ampel gefahren wird (siehe Kapitel IV für genauere Erläuterungen.)

Ist das Auto über die 1000m Marke gefahren, wird es auf 0m zurückgesetzt. Siehe Zeile 10ff. in Abbildung 17.

## III. FlashLight (R4)

In Abbildung 5 ist die Logik der gekapselten Klasse *FlashLight* dargestellt. Sobald ein Auto über eine Ampel mit rotem Licht (*redState*, *yellowRedState*) fährt, wird ein Boolean impulsartig gesetzt (Im Experiment Environment als LED dargestellt). Als „drüberfahren“ gilt dabei eine Entfernung vom Auto zur nächsten Ampel innerhalb des Intervalls von  $[-2.0[m], 0.0[m]]$ .

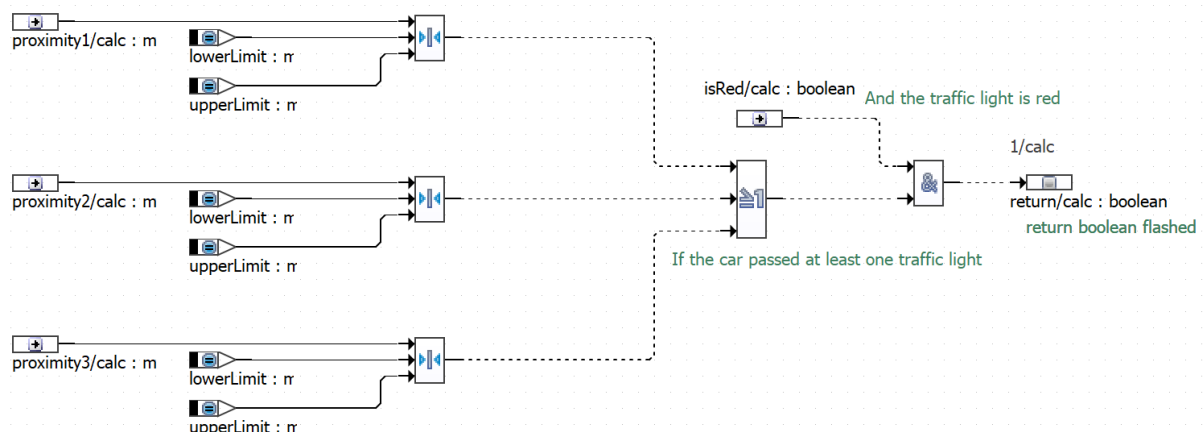


Abbildung 5: Blitzerlogik aus FlashLight.bd

## IV. Driver (R5, D2)

Der *driver* beschleunigt das Auto mittels ACC auf 50km/h. Die Bremskraft liegt dabei bei 70, damit liegt die Verzögerung laut *BrakeMomentum*-Kennlinie bei genau  $-2.5 \text{ [a]}$  und damit noch innerhalb der gegebenen Anforderung R5.

Um bei einer nicht grünen Ampel passend zu bremsen, muss zunächst unterschieden werden, ab wann man bremsen muss, beziehungsweise, bis wann man noch die Ampel passieren sollte.

Für eine obere Abschätzung wurde der Grenzfall von  $-2.5 \frac{\text{m}}{\text{s}^2}$  Beschleunigung betrachtet. Bei einer Geschwindigkeit von 50km/h ergab sich dabei ein Bremsweg von 38,59m. Als untere Abschätzung wurde die Geschwindigkeit von 50km/h und die Dauer der Gelbphase betrachtet. Dabei gab sich innerhalb der 3 Sekunden von Grün bevor Rot eine Strecke von 41,67m. (siehe Abbildung 21)

Daraus lässt sich schließen, dass man beim Ampelphasenwechsel von Grün auf Gelb bei einer ungefähren Distanz von  $\geq 40\text{m}$  eine vollständige Verzögerung bis zum Stillstand durchführen kann und diese auch durchführen sollte. Bei dieser Distanz und Geschwindigkeit lässt sich dies, wie man anhand der Berechnung sieht, mit einer Beschleunigung von  $> -2.5 \frac{\text{m}}{\text{s}^2}$  durchführen.

Im Gegensatz dazu sollte man bei einer Distanz  $< 40\text{m}$  beim Ampelphasenwechsel von Grün auf Gelb, weiterhin mit 50km/h fahren und somit vor der Rotphase hinter der Ampel sein.

Aus diesen Erkenntnissen wurde anschließend eine Entscheidungsmatrix (siehe Tabelle 1) erstellt. Die Matrix unterteilt sich in die unterschiedlichen Ampelzustände, sowie die unterschiedlichen, ausschlaggebenden Abstände des Autos zur nächsten Ampel.

		Proximity		
		> 100m	40m - 100m	0m - 40m
Traffic Light State	Yellow	50km/h	Break until Green	50km/h
	Red	50km/h	Break until Green	(Break until Green)
	YellowRed	50km/h	Break until Green	(Break until Green)
	Green	50km/h	50km/h	50km/h

Tabelle 1: Entscheidungsmatrix Driver

Aus der Entscheidungsmatrix lassen sich 2 Zustände („50km/h“ und „Break until Green“) ablesen. Für den Fall, dass die Ampel mehr als 100m (also für das Auto nicht sichtbar) entfernt ist, fährt der *driver* mittels ACC konstant 50km/h. Sobald sich eine Ampel im sichtbaren Bereich befindet, kann unterschieden werden, ob diese Ampel grün oder nicht grün ist. Im grünen Zustand fährt der *driver* ungestört weiter.

Falls sich die Ampel im Bereich von 40m bis 100m befindet, soll der *driver* das Auto abbremsen, um ein Rotlichtverstoß zu vermeiden und erst bei Grün weiterfahren. Schlägt die Ampel um, während das Auto 0m bis 40m entfernt ist, soll das Auto weiterfahren, da es nicht mehr rechtzeitig angenehm ( $> -2.5 \frac{\text{m}}{\text{s}^2}$ ) vor der Ampel anhalten kann, es vor der Rotphase rechtzeitig aber noch hinter die Ampel schaffen kann.

Die zwei geklammerten Zustände sind nicht erreichbar, da im Bereich von 40m bis 100m, sobald die Ampel nicht mehr Grün ist, abgebremst wird. Dadurch lässt sich die gesamte Entscheidungsmatrix in zwei Zustände (■, ■ in obiger Tabelle) aufteilen, die mittels der *proximity* und des Wertes *green* unterschieden werden.

In Abbildung 6 lässt sich die daraus folgende State machine ableiten.

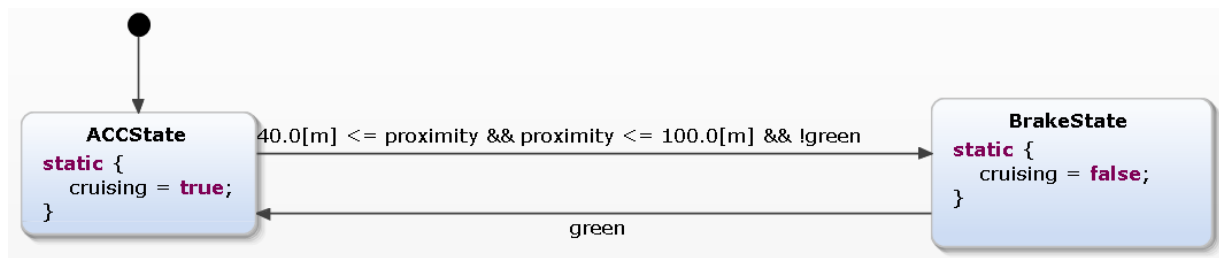


Abbildung 6: Driving State machine aus DrivingSM.sm

Dabei wird *cruising* entweder gesetzt oder nicht gesetzt. Ist *cruising* true, so regelt ACC das Auto auf 50km/h. Andernfalls wird abgebremst, um vor der Ampel stehenzubleiben.

Um nicht unnötig weit entfernt, sondern vor der Ampel stehenzubleiben, soll zum Zeitpunkt des *EdgeFallings* von *cruising* ein *requiredBrake* passend genauso berechnet werden, dass das Auto unmittelbar vor der Ampel zum Stehen kommt (siehe Abbildung 7).

Für die *requiredBrake* wurde zunächst die benötigte negative Beschleunigung für die gegebene Geschwindigkeit von 50 km/h und dem verbleibenden Bremsweg, welche der *proximity* entspricht, berechnet.

Für das Kinematik Modell „Konstante Verzögerung mit Anfangsgeschwindigkeit“ gilt für den Bremsweg  $s_{Br} = \frac{v^2}{2 \cdot a}$ , damit berechnet sich die konstante negative Beschleunigung zu:  $a = \frac{v^2}{2 \cdot s_{Br}}$ .

Um die berechnete Beschleunigung in  $\frac{m}{s^2}$  auf den benötigten Wert zu mappen, wurde die vorhandene Tabelle *BrakeMomentum* invertiert und als *InverseBrakeMomentum* im *driver* implementiert. Dafür wurde ein neuer Datentyp **type curve a real is table a -> real;** definiert (siehe Abbildung 18). Solange *cruising* false ist, wird *power* auf null und *brake* auf *requiredBrake* gesetzt. Das Auto kommt damit ca. 6 m vor der Ampel zum Stehen. Sobald die Ampel grün ist, wird wieder beschleunigt.

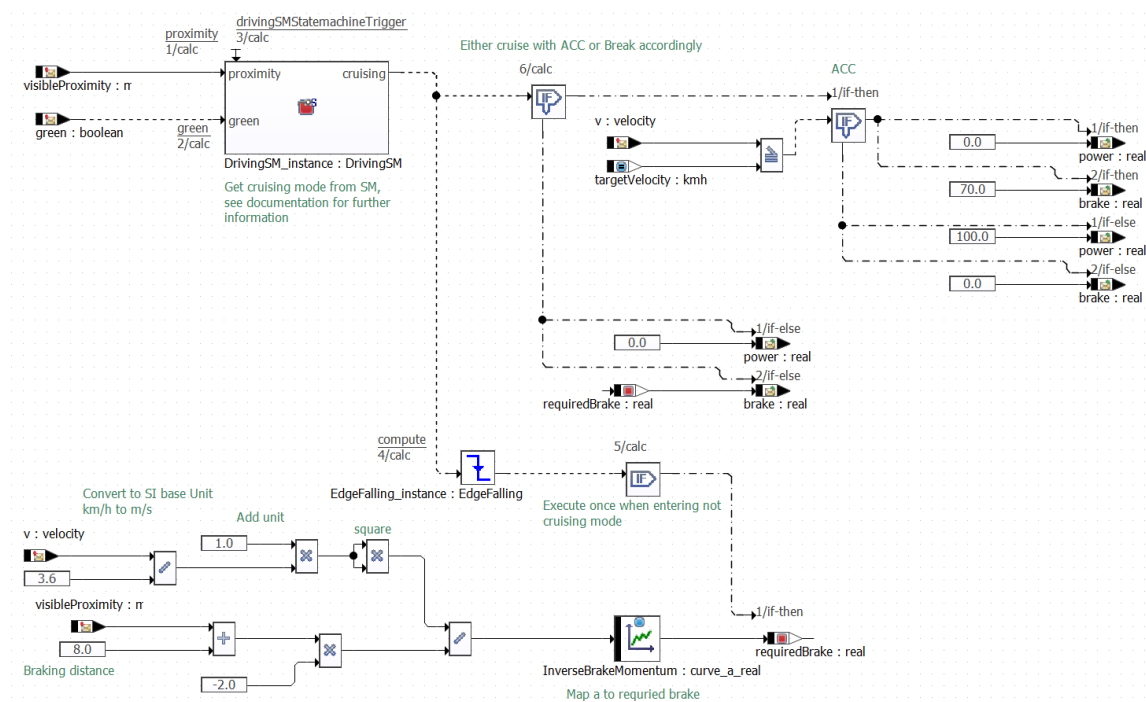


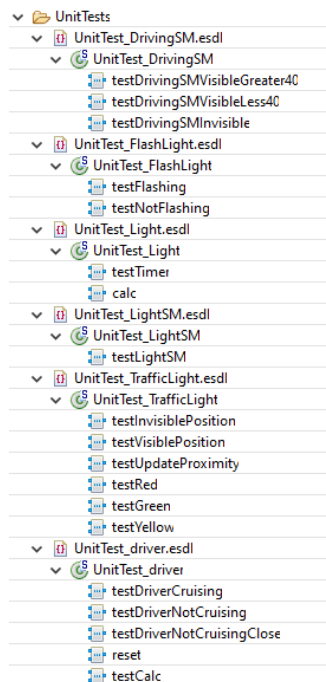
Abbildung 7: Steuerverhalten des autonomen Fahrers aus driver.db

## V. Flashlight and automated Driver (D4)

Siehe: Car (R3, R1), FlashLight (R4) und Driver (R5, D2).

## VI. Unit Tests (D5)

Es wurden zu jeder Klasse entsprechende UnitTests geschrieben (siehe Abbildung 10 bis Abbildung 15). Bis auf 3 Ausnahmen wurde bei den UnitTests eine Codeabdeckung der fachlichen Klassen von 100% erreicht (siehe Abbildung 8).



Element	Ratio	Covered Statements	Total Statements
StopRollGo	100.0 %	206	206
UnitTests	100.0 %	176	176
driver	100.0 %	2	2
DrivingSM.esdl	100.0 %	2	2
DrivingSM	100.0 %	2	2
DrivingSMStateMachine	100.0 %	2	2
ACCState	100.0 %	1	1
BrakeState	100.0 %	1	1
environment	100.0 %	28	28
FlashLight.esdl	100.0 %	1	1
FlashLight	100.0 %	1	1
calc	100.0 %	1	1
LightSM.esdl	100.0 %	12	12
LightSM	100.0 %	12	12
TrafficLightSMStateMachine	100.0 %	12	12
yellowState	100.0 %	3	3
greenState	100.0 %	3	3
redState	100.0 %	3	3
yellowRedState	100.0 %	3	3
TrafficLight.esdl	100.0 %	15	15
TrafficLight	100.0 %	15	15
setPosition	100.0 %	3	3
setRed	100.0 %	1	1
setYellow	100.0 %	1	1
setGreen	100.0 %	1	1
setVisible	100.0 %	1	1
getProximity	100.0 %	1	1
updateVisibility	100.0 %	1	1
updateProximity	100.0 %	2	2
getPosition	100.0 %	1	1
getRed	100.0 %	1	1
getYellow	100.0 %	1	1
getGreen	100.0 %	1	1

Abbildung 9: Übersicht UnitTests

Abbildung 8: Code Coverage bei Ausführung der UnitTests



```

1 package UnitTests;
2 import resources.m;
3 import driver.DrivingSM;
4 import assertLib.Assert;
5
6 static class UnitTest_DrivingSM {
7     private DrivingSM DrivingSM_testInstance;
8
9     @Test
10    public void testDrivingSMVisibleGreater40() {
11        m testProximity = 70.0[m];
12        boolean testGreen = false;
13
14        Assert.assertTrue(DrivingSM_testInstance.cruising);
15        DrivingSM_testInstance.proximity = testProximity;
16        DrivingSM_testInstance.green = testGreen;
17        DrivingSM_testInstance.drivingSMStateMachineTrigger();
18        DrivingSM_testInstance.drivingSMStateMachineTrigger();
19        Assert.assertFalse(DrivingSM_testInstance.cruising);
20
21        testGreen = true;
22        DrivingSM_testInstance.green = testGreen;
23        DrivingSM_testInstance.drivingSMStateMachineTrigger();
24        DrivingSM_testInstance.drivingSMStateMachineTrigger();
25        Assert.assertTrue(DrivingSM_testInstance.cruising);
26    }
27
28    @Test
29    public void testDrivingSMVisibleLess40() {
30        m testProximity = 30.0[m];
31        boolean testGreen = false;
32
33        Assert.assertTrue(DrivingSM_testInstance.cruising);
34        DrivingSM_testInstance.proximity = testProximity;
35        DrivingSM_testInstance.green = testGreen;
36        DrivingSM_testInstance.drivingSMStateMachineTrigger();
37        DrivingSM_testInstance.drivingSMStateMachineTrigger();
38        Assert.assertTrue(DrivingSM_testInstance.cruising);
39
40        testGreen = true;
41        DrivingSM_testInstance.green = testGreen;
42        DrivingSM_testInstance.drivingSMStateMachineTrigger();
43        DrivingSM_testInstance.drivingSMStateMachineTrigger();
44        Assert.assertTrue(DrivingSM_testInstance.cruising);
45    }
46
47    @Test
48    public void testDrivingSMInvisible() {
49        m testProximity = 110.0[m];
50        boolean testGreen = false;
51
52        Assert.assertTrue(DrivingSM_testInstance.cruising);
53        DrivingSM_testInstance.proximity = testProximity;
54        DrivingSM_testInstance.green = testGreen;
55        DrivingSM_testInstance.drivingSMStateMachineTrigger();
56        DrivingSM_testInstance.drivingSMStateMachineTrigger();
57        Assert.assertTrue(DrivingSM_testInstance.cruising);
58
59        testGreen = true;
60        DrivingSM_testInstance.green = testGreen;
61        DrivingSM_testInstance.drivingSMStateMachineTrigger();
62        DrivingSM_testInstance.drivingSMStateMachineTrigger();
63        Assert.assertTrue(DrivingSM_testInstance.cruising);
64    }
65 }

```

Abbildung 10: UnitTest\_DrivingSM.esdl



```

1 package UnitTests;
2 import environment.FlashLight;
3 import resources.m;
4 import assertLib.Assert;
5
6 static class UnitTest_FlashLight {
7     boolean flashed = false;
8
9     @Test
10    public void testFlashing() {
11        boolean red = true;
12        m proximity1 = 0.0[m];
13        m proximity2 = 200.0[m];
14        m proximity3 = 600.0[m];
15
16        flashed = FlashLight.calc(proximity1, proximity2, proximity3, red);
17        Assert.assertTrue(flashed);
18
19        proximity1 = -200.0[m];
20        proximity2 = 0.0[m];
21        proximity3 = 400.0[m];
22
23        flashed = FlashLight.calc(proximity1, proximity2, proximity3, red);
24        Assert.assertTrue(flashed);
25
26        proximity1 = -600.0[m];
27        proximity2 = -400.0[m];
28        proximity3 = 0.0[m];
29
30        flashed = FlashLight.calc(proximity1, proximity2, proximity3, red);
31        Assert.assertTrue(flashed);
32    }
33
34    @Test
35    public void testNotFlashing() {
36        boolean red = false;
37        m proximity1 = 0.0[m];
38        m proximity2 = 200.0[m];
39        m proximity3 = 600.0[m];
40
41        flashed = FlashLight.calc(proximity1, proximity2, proximity3, red);
42        Assert.assertFalse(flashed);
43
44        proximity1 = -200.0[m];
45        proximity2 = 0.0[m];
46        proximity3 = 400.0[m];
47
48        flashed = FlashLight.calc(proximity1, proximity2, proximity3, red);
49        Assert.assertFalse(flashed);
50
51        proximity1 = -600.0[m];
52        proximity2 = -400.0[m];
53        proximity3 = 0.0[m];
54
55        flashed = FlashLight.calc(proximity1, proximity2, proximity3, red);
56        Assert.assertFalse(flashed);
57    }
58 }

```

Abbildung 11: UnitTest\_FlashLight.esdl

```

1 package UnitTests;
2 import resources.s;
3 import environment.LightSM;
4 import assertLib.Assert;
5
6 static class UnitTest_LightSM {
7     private LightSM LightSM_testInstance;
8
9     @Test
10    public void testLightSM() {
11        Assert.assertFalse(LightSM_testInstance.red);
12        Assert.assertFalse(LightSM_testInstance.yellow);
13        Assert.assertFalse(LightSM_testInstance.green);
14
15        LightSM_testInstance.trafficLightSMStateMachineTrigger();
16        Assert.assertFalse(LightSM_testInstance.red);
17        Assert.assertTrue(LightSM_testInstance.yellow);
18        Assert.assertFalse(LightSM_testInstance.green);
19
20        LightSM_testInstance.elapsedTime = 5.0[s];
21        LightSM_testInstance.trafficLightSMStateMachineTrigger();
22        LightSM_testInstance.trafficLightSMStateMachineTrigger();
23        Assert.assertTrue(LightSM_testInstance.red);
24        Assert.assertFalse(LightSM_testInstance.yellow);
25        Assert.assertFalse(LightSM_testInstance.green);
26
27        LightSM_testInstance.elapsedTime = 42.0[s];
28        LightSM_testInstance.trafficLightSMStateMachineTrigger();
29        LightSM_testInstance.trafficLightSMStateMachineTrigger();
30        Assert.assertTrue(LightSM_testInstance.red);
31        Assert.assertTrue(LightSM_testInstance.yellow);
32        Assert.assertFalse(LightSM_testInstance.green);
33
34        LightSM_testInstance.elapsedTime = 45.0[s];
35        LightSM_testInstance.trafficLightSMStateMachineTrigger();
36        LightSM_testInstance.trafficLightSMStateMachineTrigger();
37        Assert.assertFalse(LightSM_testInstance.red);
38        Assert.assertFalse(LightSM_testInstance.yellow);
39        Assert.assertTrue(LightSM_testInstance.green);
40
41        LightSM_testInstance.elapsedTime = 59.0[s];
42        LightSM_testInstance.trafficLightSMStateMachineTrigger();
43        LightSM_testInstance.trafficLightSMStateMachineTrigger();
44        Assert.assertFalse(LightSM_testInstance.red);
45        Assert.assertFalse(LightSM_testInstance.yellow);
46        Assert.assertTrue(LightSM_testInstance.green);
47
48        LightSM_testInstance.elapsedTime = 1.0[s];
49        LightSM_testInstance.trafficLightSMStateMachineTrigger();
50        LightSM_testInstance.trafficLightSMStateMachineTrigger();
51        Assert.assertFalse(LightSM_testInstance.red);
52        Assert.assertTrue(LightSM_testInstance.yellow);
53        Assert.assertFalse(LightSM_testInstance.green);
54    }
55 }

```

Abbildung 12: UnitTest\_LightSM.esdl

```

1 package UnitTests;
2 import resources.m;
3 import environment.TrafficLight;
4 import assertLib.Assert;
5
6 static class UnitTest_TrafficLight {
7     private TrafficLight TrafficLight_testInstance;
8
9     @Test
10    public void testInvisiblePosition() {
11        TrafficLight_testInstance.setPosition(500.0[m]);
12        Assert.assertNear(500.0, TrafficLight_testInstance.getProximity() / 1.0[m], 0.001);
13        Assert.assertNear(500.0, TrafficLight_testInstance.getPosition() / 1.0[m], 0.001);
14        Assert.assertFalse(TrafficLight_testInstance.getVisible());
15    }
16
17    @Test
18    public void testVisiblePosition() {
19        TrafficLight_testInstance.setPosition(50.0[m]);
20        Assert.assertNear(50.0, TrafficLight_testInstance.getProximity() / 1.0[m], 0.001);
21        Assert.assertNear(50.0, TrafficLight_testInstance.getPosition() / 1.0[m], 0.001);
22        Assert.assertTrue(TrafficLight_testInstance.getVisible());
23    }
24
25    @Test
26    public void testUpdateProximity() {
27        TrafficLight_testInstance.setPosition(500.0[m]);
28        Assert.assertNear(500.0, TrafficLight_testInstance.getProximity() / 1.0[m], 0.001);
29        Assert.assertNear(500.0, TrafficLight_testInstance.getPosition() / 1.0[m], 0.001);
30        Assert.assertFalse(TrafficLight_testInstance.getVisible());
31        TrafficLight_testInstance.updateProximity(450.0[m]);
32        Assert.assertNear(50.0, TrafficLight_testInstance.getProximity() / 1.0[m], 0.001);
33        Assert.assertNear(500.0, TrafficLight_testInstance.getPosition() / 1.0[m], 0.001);
34        Assert.assertTrue(TrafficLight_testInstance.getVisible());
35    }
36
37    @Test
38    public void testRed() {
39        Assert.assertFalse(TrafficLight_testInstance.getRed());
40        TrafficLight_testInstance.setRed(true);
41        Assert.assertTrue(TrafficLight_testInstance.getRed());
42        TrafficLight_testInstance.setRed(false);
43        Assert.assertFalse(TrafficLight_testInstance.getRed());
44    }
45
46    @Test
47    public void testGreen() {
48        Assert.assertFalse(TrafficLight_testInstance.getGreen());
49        TrafficLight_testInstance.setGreen(true);
50        Assert.assertTrue(TrafficLight_testInstance.getGreen());
51        TrafficLight_testInstance.setGreen(false);
52        Assert.assertFalse(TrafficLight_testInstance.getGreen());
53    }
54
55    @Test
56    public void testYellow() {
57        Assert.assertFalse(TrafficLight_testInstance.getYellow());
58        TrafficLight_testInstance.setYellow(true);
59        Assert.assertTrue(TrafficLight_testInstance.getYellow());
60        TrafficLight_testInstance.setYellow(false);
61        Assert.assertFalse(TrafficLight_testInstance.getYellow());
62    }
63 }

```

Abbildung 13: UnitTest\_TrafficLight.esdl

## Ausnahmen

### Light

Da es sich dabei um eine statische Klasse handelt, und ihre zu testende Funktion `@Thread` annotiert ist, kann diese nur vom Scheduler aufgerufen werden und nicht einfach mittels `UnitTest` getestet werden.

Da der Großteil der Logik von `UnitTest_Light` sich in einer State Machine befindet, welche bereits separat getestet wird (siehe Abbildung 12), wurde die restliche Logik des Sekundenzählers in eine eigene Funktion innerhalb des UnitTests kopiert, um sie dort zu testen (siehe Abbildung 14).

```

1 package UnitTests;
2 import resources.s;
3 import assertLib.Assert;
4
5 static class UnitTest_Light {
6     s elapsedTime = 0.0[s];
7
8     @Test
9     public void testTimer() {
10         Assert.assertNear(elapsedTime / 1.0[s], 0.0, 0.001);
11
12         testCalc(5.0[s]);
13         Assert.assertNear(elapsedTime / 1.0[s], 5.0, 0.001);
14
15         testCalc(30.0[s]);
16         Assert.assertNear(elapsedTime / 1.0[s], 35.0, 0.001);
17
18         testCalc(20.0[s]);
19         Assert.assertNear(elapsedTime / 1.0[s], 55.0, 0.001);
20
21         testCalc(4.0[s]);
22         Assert.assertNear(elapsedTime / 1.0[s], 59.0, 0.001);
23
24         testCalc(1.0[s]);
25         Assert.assertNear(elapsedTime / 1.0[s], 0.0, 0.001);
26
27         testCalc(60.0[s]);
28         Assert.assertNear(elapsedTime / 1.0[s], 0.0, 0.001);
29
30         testCalc(45.0[s]);
31         Assert.assertNear(elapsedTime / 1.0[s], 45.0, 0.001);
32     }
33
34     public void testCalc(s deltaT) {
35         elapsedTime = (deltaT + elapsedTime);
36         if (elapsedTime >= 60.0[s]) {
37             elapsedTime = 0.0[s];
38         }
39     }
40 }

```

Abbildung 14: `UnitTest_Light.esdl`

## driver

Da es sich dabei um eine statische Klasse handelt, und ihre zu testende Funktion @Thread annotiert ist, kann diese nur vom Scheduler aufgerufen werden und nicht einfach mittels UnitTest getestet werden.

Ein Teil der Logik von *UnitTest\_driver* befindet sich in einer Statemachine, welche bereits separat getestet wird (siehe Abbildung 10). Eine weitere Funktionalität, *EdgeFalling*, wurde aus der *ASCET-SystemLib* importiert. Dabei kann davon ausgegangen werden, dass diese hinreichend getestet wurde. Den Rest der Logik, wie das ACC und die Berechnung des *requiredBrake* Wertes wurde wieder in eine eigene Funktion innerhalb des UnitTests kopiert, um sie dort zu testen (Abbildung 15).

```

1 package UnitTests;
2 import resources.curve_a_real;
3 import resources.a;
4 import resources.ms;
5 import resources.kmh;
6 import resources.m;
7 import assertLib.Assert;
8
9 static class UnitTest_driver {
10   characteristic curve_a_real InverseBrakeMomentum[5] = {{-4.0[a], -3.0[a], -2.0[a], -1.0[a], 0.0[a]}, {100.0, 80.0, 60.0, 40.0, 0.0}};
11   const kmh targetVelocity = 50.0[kmh];
12   real power = 0.0;
13   real brake = 0.0;
14
15   @Test
16   public void testDriverCruising() {
17     boolean testCruising = true;
18     m testProximity = 0.0[m];
19     kmh testV = 30.0[kmh];
20
21     reset();
22
23     Assert.assertNear(power, 0.0, 0.001);
24     Assert.assertNear(brake, 0.0, 0.001);
25
26     testCalc(testV, testProximity, testCruising);
27     Assert.assertNear(power, 100.0, 0.001);
28     Assert.assertNear(brake, 0.0, 0.001);
29
30     testV = 70.0[kmh];
31     testCalc(testV, testProximity, testCruising);
32     Assert.assertNear(power, 0.0, 0.001);
33     Assert.assertNear(brake, 70.0, 0.001);
34   }
35
36   @Test
37   public void testDriverNotCruising() {
38     boolean testCruising = false;
39     m testProximity = 70.0[m];
40     kmh testV = 50.0[kmh];
41
42     reset();
43
44     Assert.assertNear(power, 0.0, 0.001);
45     Assert.assertNear(brake, 0.0, 0.001);
46
47     testCalc(testV, testProximity, testCruising);
48
49     Assert.assertNear(power, 0.0, 0.001);
50     Assert.assertNear(brake, 45.0, 2.0);
51   }
52
53   @Test
54   public void testDriverNotCruisingClose() {
55     boolean testCruising = false;
56     m testProximity = 40.0[m];
57     kmh testV = 50.0[kmh];
58
59     reset();
60
61     Assert.assertNear(power, 0.0, 0.001);
62     Assert.assertNear(brake, 0.0, 0.001);
63
64     testCalc(testV, testProximity, testCruising);
65
66     Assert.assertNear(power, 0.0, 0.001);
67     Assert.assertNear(brake, 60.0, 2.0);
68   }
69
70   public void reset() {
71     power = 0.0;
72     brake = 0.0;
73   }
74
75   public void testCalc(kmh v, m visibleProximity, boolean cruising) {
76     real requiredBrakeMomentum = 0.0;
77
78     requiredBrakeMomentum = InverseBrakeMomentum.getAt((((1.0[ms] * (v / 3.6[kmh]))) * (1.0[ms] * (v / 3.6[kmh])))) / ((visibleProximity + 8.0[m]) * -2.0));
79
80     if (cruising) {
81       if (v >= targetVelocity) {
82         power = 0.0;
83         brake = 70.0;
84       } else {
85         power = 100.0;
86         brake = 0.0;
87       }
88     } else {
89       power = 0.0;
90       brake = requiredBrakeMomentum;
91     }
92   }
93 }

```

Abbildung 15: UnitTest\_driver.esdl

## TrafficLightController

Beim *TrafficLightController* handelt es sich ebenfalls um eine mit `@Thread` annotierte, statische Klasse.

Der Controller ist nur dafür verantwortlich, die Instanzen der *TrafficLights* zu halten und Informationen zu delegieren. Jede Funktionalität, die im *TrafficLightController* beinhaltet ist, wurde bereits separat getestet. *TrafficLightController* benötigt demnach keine eigenen Tests mehr.

## VII. Data Interfaces & Data Types

```

1 package resources;
2
3 // All necessary TrafficLight information
4 data interface TrafficLightMessages {
5     boolean red = false;
6     boolean yellow = false;
7     boolean green = false;
8     boolean flashed = false;
9     m visibleProximity = 0.0[m];
10 }

```

Abbildung 16: Data Interface aus *TrafficLightMessages.esdl*

```

1 package resources;
2
3 data interface CarMessages {
4     @a2l_unit_label("km/h")
5     velocity v = 0.0[kmh];
6     real power = 0.0;
7     real brake = 0.0;
8     real steering = 0.0;
9     real bearing = 0.0;
10    // Use the dist of the car in the lap, because it resets itself
11    @a2l_unit_label("m")
12    m dist = 0.0[m];
13    @a2l_unit_label("m")
14    m x = 0.0[m];
15    @a2l_unit_label("m")
16    m y = 0.0[m];
17 }

```

Abbildung 17: Data Interface aus *CarMessages.esdl*

```

1 package resources;
2
3 type curve_real is table real -> real;
4 // Add curve that maps from a to real for requiredBrake
5 type curve_a_real is table a -> real;
6 type curve_real_a is table real -> a;
7 type curve_m is table m -> m;
8 type curve_s is table s -> real;
9 type curve_kmh is table kmh -> a;
10
11 type map_real_real is table real, real -> real;
12 type map_real_kmh is table real, kmh -> a;

```

Abbildung 18: *CharTableTypes.esdl*



## VIII. Experiment Environment (D6)

Im Experiment Environment befindet sich eine übersichtliche Darstellung aller relevanten Informationen (siehe Abbildung 19).

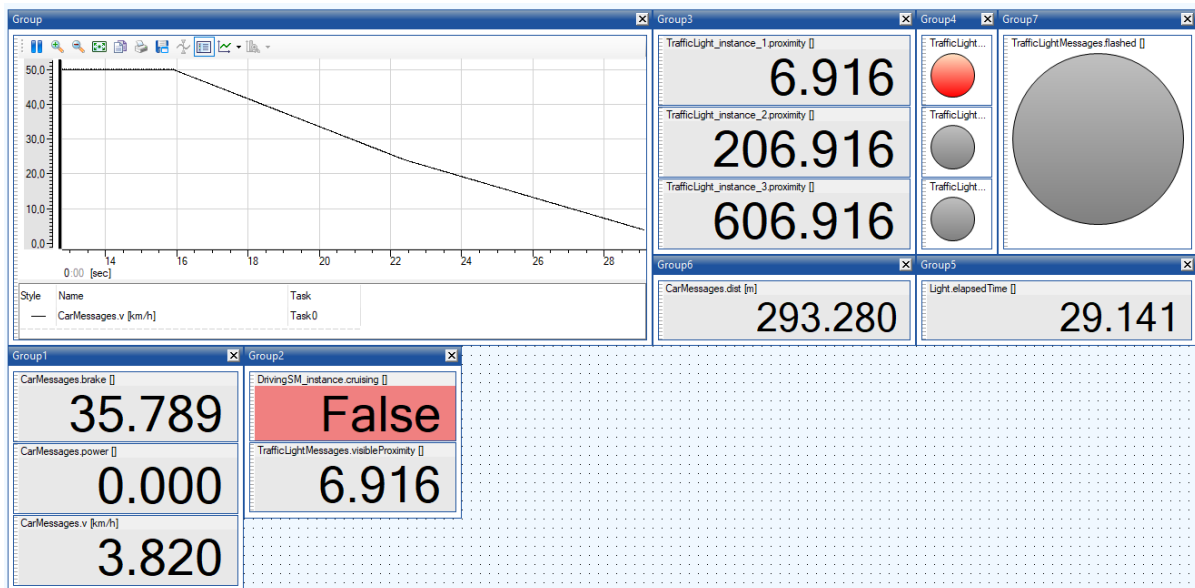


Abbildung 19: Experiment Environment bezüglich des Systemtests

Dargestellt sind:

- CarMessages.v: Geschwindigkeit des Autos, als Oszilloskop und Edit Box
- CarMessages.brake: Bremswert des Autos, als Edit Box
- CarMessages.power: Leistung des Autos, als Edit Box
- DrivingSM\_instance.cruising: Zustand des Autos (ACC oder Ampel abhängig), als Edit Box
- TrafficLightMessages.visibleProximity: Abstand des Autos zu einer sichtbaren Ampel, als Edit Box
- TrafficLight\_instance\_X.proximity: Abstand des Autos zu der entsprechenden Ampel, als Edit Box
- TrafficLightMessages.red/yellow/green: Zustand des Ampellichts, als LED in entsprechender Farbe
- TrafficLightMessages.flashed: Signal für Blitzlicht bei Rotlichtverstoß, als LED
- CarMessages.dist: Entfernung auf der Strecke [0m, 1000m], als Edit Box
- Light.elapsedTime: Vergangene Zeit deltaT modulo 60, als Edit Box

Ein kompletter Durchlauf des Experiments als Video kann hier angeschaut werden:



Abbildung 20: Demo Experiment StopRollGo

Oder als Link: <https://www.youtube.com/watch?v=QHCLL9ebhEQ>



## IX. Zusatzaufgaben

### Wie sollte das Timing bzgl. der Gelbphase für unterschiedliche Geschwindigkeitsbeschränkungen gewählt werden? (D7\*)

Da die Sichtweite zur Ampel konstant bleibt, muss, um dem *driver* die Möglichkeit zu geben zu reagieren, bei zunehmender Geschwindigkeit die Dauer der Gelbphase ebenfalls verlängert werden. Bei unserem momentanen Beispiel, mit einer Geschwindigkeit 50km/h und einer Gelbphasendauer von 3s, steht dem *driver* ein theoretischer Spielraum von  $\Delta s = 3.08m$  zur Verfügung. Diese 3.08m legt er mit seiner Geschwindigkeit in ca.  $\Delta t = 0.222s$  zurück. Um dem *driver* bei anderen Geschwindigkeiten die gleiche Reaktionszeit zu gewährleisten, sollte  $\Delta s$  abhängig von der Geschwindigkeit gemacht werden. Mit ausgewähltem  $\Delta s$ , lässt sich dann für die Richtgeschwindigkeit  $v$  eine entsprechende Gelbphasendauer berechnen (siehe Abbildung 22).

### Wie kann die Verkehrssituation verbessert werden, wenn V2X eingesetzt wird oder R5 flexibler ist? (D9\*)

Bei einer zukünftigen Voraussicht bzgl. Ampelphasen durch V2X kann ein optimaler Bremsvorgang eingeleitet werden, um z.B. im Fall von Elektrofahrzeugen eine maximale Energierückgewinnung in den Akkumulator mittels Rekuperation zu ermöglichen. Ebenso kann z.B. an einer Kreuzung von unterschiedlich stark befahrenen Straßen eine Priorisierung vorgenommen werden, da die Ampelanlage über die Anzahl der in Zukunft zu passierenden Fahrzeugen auf der jeweiligen Straße informiert ist. Damit ist es möglich, unnötige oder unnötig lange Rotphasen zu schalten und den Verkehr auf den Hauptverkehrsstraßen mit verhältnismäßig langen Grünphasen zu priorisieren.

### Begründung über den Einfluss menschlicher Wahrnehmung bzw. Reaktion oder anderer Verzögerungen (D12\*)

Bei einem menschlichen Fahrer muss neben der Bremsdauer auch noch die Reaktionsdauer, also die Zeitspanne zwischen Auslösen eines Ereignisses z.B. Ampel schaltet auf Rot und Eintritt der Fahrzeugverzögerung durch Betätigung des Bremspedales beachtet werden. Diese kann sehr stark variieren. So ist diese bei jungen Fahrern typischerweise geringer und bei älteren Menschen etwas länger. Diese Schwankungsbreite muss berücksichtigt werden. Beispielsweise sollte ein solcher manueller Fahrer von einem vollautonomen System nicht mit einem Referenzwert bzgl. Reaktionszeit, sondern vielmehr mit Worstcase Referenzzeiten angesehen werden.

Reflektion: Welche anderen Beobachtungen oder Kommentare gibt es bezüglich der Planung, der Modellierung, den Anforderungen, den vorgeschriebenen Funktionen oder Ihre Lösung, dem Testen und des gewählten grafischen Ansatzes?

- Keine Konstruktoren in ESDL-Klassen
- Testen von statischen thread-annotierten Methoden nicht einfach möglich. Workaround: Einfügen einer Kopie des Inhalts der entsprechenden Funktion in eine Test-Funktion. Dies führt aber zu Bad (Code-) Smells, da bei jeder Änderung mehrere Stellen im Code angefasst werden müssen.
- Das Einrichten einer Versionsverwaltung mit Git bedarf umfassende Kenntnisse bzgl. gitignore-Dateien. Manche Dateiformate von **ASCET** erzeugen bspw. einen Zeitstempel, obwohl keine inhaltliche bzw. fachliche Änderung am Projekt vorgenommen wurde. In diesem Fall muss die lokale Kopie im Anschluss manuell resettet werden.
- Eine bidirektionale Konvertierung von .esdl zu .bd wäre schön. Aus Informatiker-Sicht ist der Zugang über Code anstelle einer grafischen Modellierung für die Implementierung einer Funktionalität manchmal naheliegender. Hier wäre evtl. der Erhalt des grafischen Layouts nach einer Umwandlung von esdl zu bd wünschenswert.

## X. Anhang

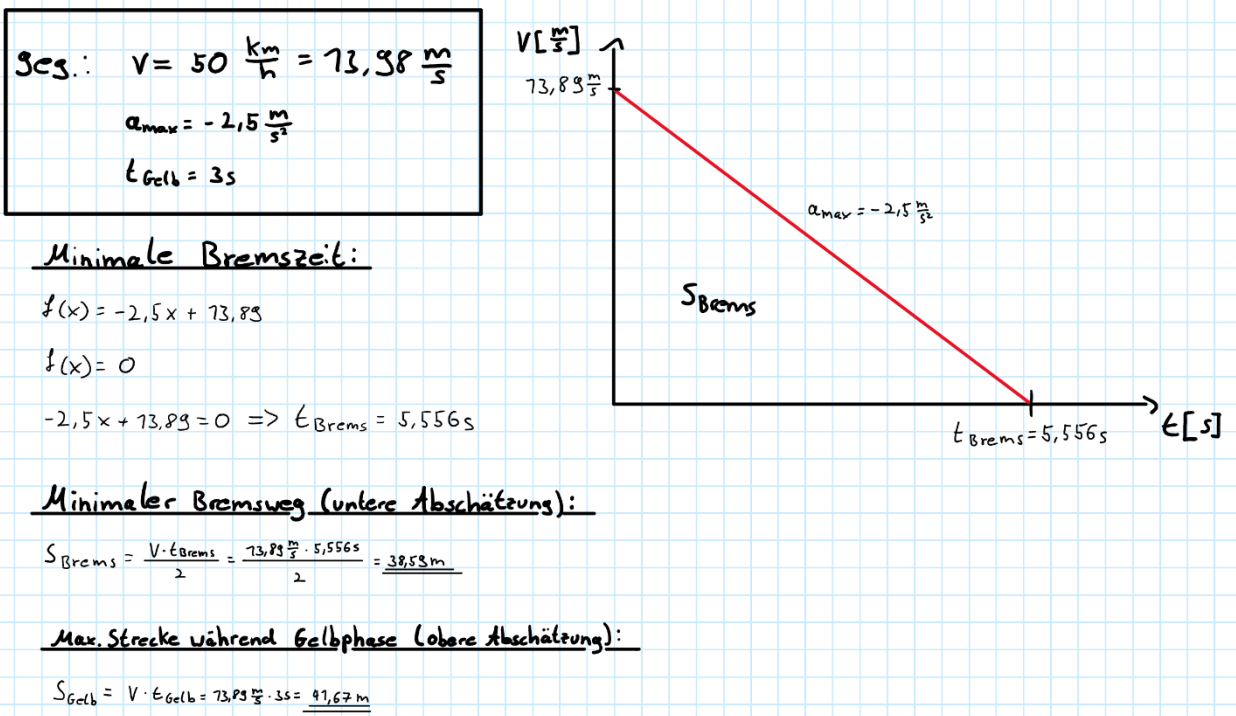


Abbildung 21: Abschätzung bzgl. Machbarkeit Bremsverzögerung

ges.:  $t_{\text{Gelb}}$  für  $v \neq \text{const.}$  und  $\Delta s \geq 0 \text{ m}$        $\Delta t = 0,222 \text{ s}$

$\Rightarrow \Delta s = v_x \cdot \Delta t$

$$\Delta s = S_{\text{Gelb}} - S_{\text{Brems}}$$

$$\Delta s = S_{\text{Gelb}} - \frac{v \cdot t_{\text{Brems}}}{2}$$

$$\Delta s = v \cdot t_{\text{Gelb}} + \frac{v^2}{2a}$$

$$t_{\text{Gelb}} = \frac{\Delta s}{v} - \frac{v}{2a} \quad \text{mit } \Delta s \geq 0 \text{ m}$$

Abbildung 22: Gelbphasendauer für variable Geschwindigkeit