technische universität
dortmund

Abschlussbericht

PG 613: ACC++

Domänenspezifische Sprachen für Autonomes Fahren

Wintersemester 2017/2018
und Sommersemester 2018

PG-Teilnehmer:
Nils Bergmann,
Natascha Bomm,
Hendrik Eckardt,
Hendrik Grewe,
David Hüttner,
Brian-Frederik Jahnke,
Daniel Lenzen,
Benedikt Maus,
Hendrik Schröder,
Tim Tannert,
Timo Walter,
Daniel Wisniewski

Betreuer:
Falk Howar,
Bernhard Steffen,
Stefan Naujokat

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Programmiersysteme LS5
http://ls5-www.cs.tu-dortmund.de

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Software Engineering LS14
http://ls14-www.cs.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

In modernen Automobilsystemen bekommen automatisierte Fahrassistenzsysteme eine immer größere Bedeutung zugeschrieben. Sie bieten den Fahrern verschiedene Vorteile, von einfachen Komfortfunktionen bis hin zu wertvollen Sicherheitsfunktionen. Die Entwicklung dieser Systeme ist allerdings traditionell mit einem enormen Aufwand verbunden. Der heutige Standard ist die manuelle Umsetzung der Anforderungen in Blockschaltbildern. Dabei können durch Unklarheiten in der Spezifikation oder Fehlern bei der Umsetzung der Funktionalitäten schnell Sicherheitsrisiken entstehen. Daraus entsteht ein großer manueller Aufwand für die Absicherung und das Sicherstellen des erwarteten Verhaltens des Gesamtsystems.

Man spricht auch davon, dass bei der klassischen Entwicklungsmethode eine *semantische Lücke* zwischen den Anforderungen und der Umsetzung besteht. Abbildung 1.1 stellt diese Lücke schematisch dar. Wenn zum Beispiel die Anforderungen in Form eines Zustandsdiagramms spezifiziert werden, wird zunächst die Zustandslogik in ein Blockschaltbild integriert, und später wird auf dessen Grundlage manuell C-Code entwickelt. Viel sinnvoller wäre es, aus dem Zustandsdiagramm direkt eine entsprechende Implementierung zu erzeugen. Dies vermeidet Fehler, da der Entwicklungsprozess bei Verfügbarkeit eines entsprechenden Werkzeugs durch Domänenexperten (z.B. Ingenieure) auf einer hohen Abstraktionsebene vorgenommen werden kann. Die Implementierung des automatischen Umsetzungsschritts – auch *Codegenerator* genannt – muss dabei nur einmalig erfolgen und ist von spezifischen Assistenzsystemen unabhängig.

Die Projektgruppe behandelt den Ansatz, sich vom klassischen Vorgehen der Automobilindustrie zu entfernen und stattdessen eine abstrahierte Modellierung nach dem obigen Muster zu ermöglichen. Durch die Verwendung von domänenspezifischen Sprachen soll die Distanz zwischen Spezifikation und Umsetzung reduziert werden, um letztendlich die semantische Lücke zu schließen. Aus Modellen in diesen spezifischen Sprachen soll dann

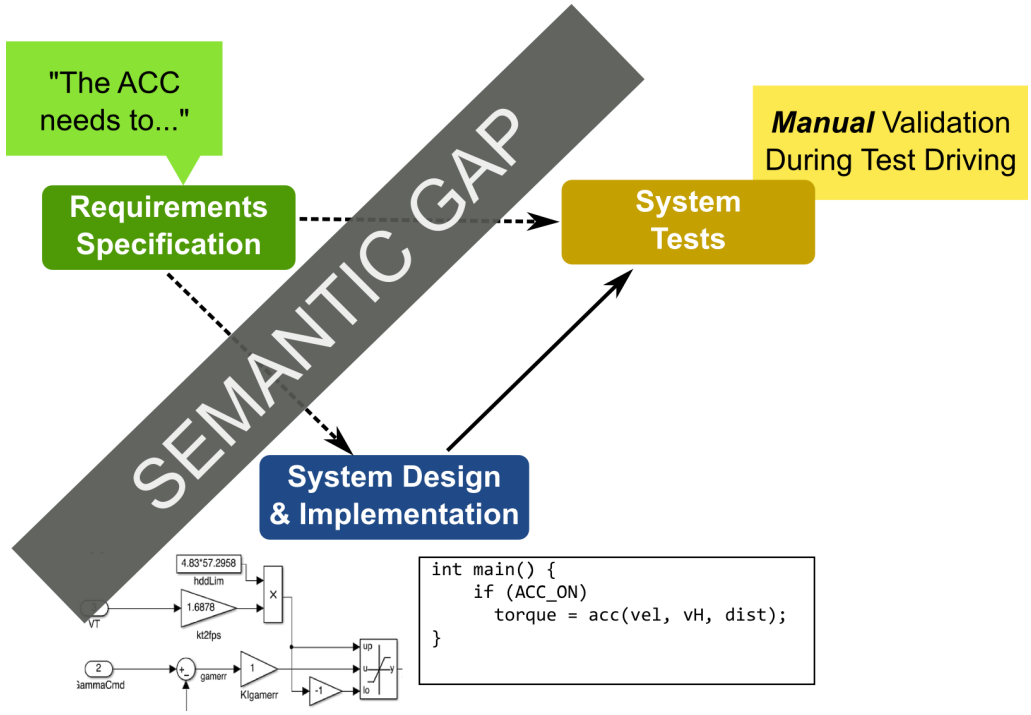


Abbildung 1.1: Semantische Lücke bei der Implementierung von Fahrassistenzsystemen

automatisch die tatsächliche Umsetzung mit dazugehörigen Tests und Absicherungen generiert werden.

1.1 Zielsetzung

Ziel der Projektgruppe ist es, mit dem oben beschriebenen Ansatz die Modellierung von Fahrassistenzsystemen zu vereinfachen. Als spezifisches Beispiel wird dafür *Adaptive Cruise Control* (ACC) verwendet. ACC arbeitet mit zwei Zuständen, in dem einen hält es eine eingestellte Geschwindigkeit, in dem anderen den Abstand zu einem vorausfahrenden Fahrzeug. Eine Modellierungsumgebung soll mithilfe der CINCO Meta Tooling Suite [2] erstellt werden. In dieser Umgebung sollen Modelle entwickelt werden können, aus denen durch Generatoren lauffähige Implementierungen und dazugehörige Zusatzkomponenten erzeugt werden. Diese Generatoren werden ebenfalls von der Projektgruppe entwickelt. Die Arbeit der Projektgruppe soll mithilfe von Modellfahrzeugen im Maßstab 1:8 und durch die Simulationsumgebung *Virtual Test Drive* (VTD) [11] evaluiert werden. Auf den Modellfahrzeugen bietet das *Automotive Data and Time-Triggered Framework* (ADTF) [10] eine Schnittstelle für das generierte System. Während der Evaluation müssen Verhalten und Zustand des Testfahrzeuges überwacht werden. Zusätzlich soll es möglich sein komplette Testszenarien zu beschreiben und zu erzeugen. Ein Überblick über die verwendeten Technologien und Konzepte ist in Abbildung 1.2.

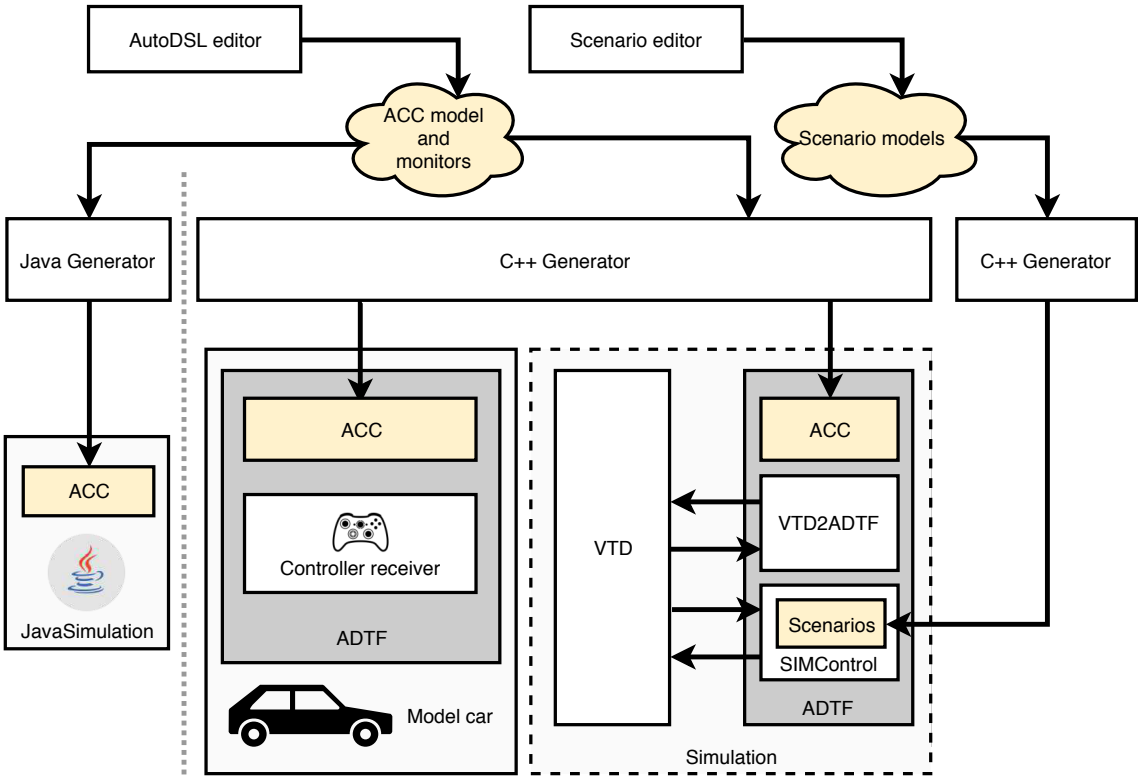


Abbildung 1.2: Gesamtüberblick über die Projektstruktur und die verwendeten Technologien

1.2 Aufbau der Arbeit

Zu Beginn wird in Teil I auf die Grundlagen der verwendeten Technologien und Methoden eingegangen, wie z.B. grundlegendes zu Fahrassistenzsystemen und domänenspezifischen Sprachen. Als Nächstes folgt die konzeptionelle Umsetzung der entwickelten domänenspezifischen Sprachen (Teil II). Im Anschluss wird auf die technische Umsetzung eingegangen (Teil III). Dazu gehören u.A. die Umsetzung des Zustandsmodells in CINCO und selbst entwickelte Komponenten in ADTF. In Teil IV wird eine Fallstudie zu dem entwickelten ACCs vorgestellt, in der diese durch eine Simulationsumgebung und Realfahrten bzgl. der gesetzten Ziele evaluiert wird. Abschließend wir in Teil V die Arbeit zusammengefasst und ein kurzer Ausblick geboten.

Teil I

Grundlagen

Kapitel 2

Fahrassistenzsysteme

In diesem Kapitel wird zunächst grundlegend erklärt, was unter dem Begriff Fahrassistenzsystem zu verstehen ist. Anschließend wird die für solche Systeme häufig verwendete Regelungstechnik anhand des PID-Reglers näher erläutert. Weiter wird die Funktionsweise von *Adaptive Cruise Control* (ACC) als Beispiel für ein solches Fahrassistenzsystem anhand der entsprechenden *ISO-Norm 15622* dargestellt.

2.1 Einleitung

Die Entwicklung von Autos beschränkt sich bereits seit mehreren Jahren nicht mehr lediglich auf die mechanischen Aspekte. Bereits seit geraumer Zeit existiert eine Vielzahl an Fahrassistenzsystemen in Fahrzeugen, die das Fahrerlebnis zum einen sicherer und zum anderen komfortabler machen. Dabei existieren auf der einen Seite Systeme wie z.B. *Antiblockiersystem (ABS)*, *Antriebsschlupfregelung (ASR)* oder *Elektronisches Stabilitätsprogramm (ESP)*, die im Hintergrund passiv in das Fahrverhalten des Fahrzeugs eingreifen. Dabei merkt der Fahrer oft gar nicht, dass diese Systeme arbeiten. Auf der anderen Seite existieren Systeme, die aktiv vom Fahrer aktiviert werden müssen und nur unter bestimmten Bedingungen laufen können. Dazu gehören *Adaptive Cruise Control (ACC)* (vgl. Abschnitt 2.3), Einparksysteme, Spurhalteassistenten, Tempomaten und weitere. Viele Systeme geben dem Fahrer lediglich Informationen und greifen nicht aktiv ein und belassen somit die Verantwortung beim Fahrer. Mit den Bemühungen der Automobilhersteller zum autonomen Fahren wird dem Fahrer jedoch immer mehr Arbeit abgenommen.

Diese Systeme benötigen Wissen über die aktuelle Fahrsituation. An dieses gelangen sie über den *Controller Area Network (CAN)*-Bus, an dem entsprechende Sensoren angebunden sind. Zu diesen Sensoren gehören unter anderem Ultraschallsensoren,*light detection and ranging (Lidar)* bzw. Radar, diverse Kameraarten, GPS-Empfänger und Gyroskope. Mit Hilfe dieser Sensoren können die Fahrassistenzsysteme an Informationen über die ak-

Abbildung 2.1: Bereiche, die ein 7er BMW durch Sensoren abdeckt. [1]

tuelle Geschwindigkeit, die aktuelle Position und Abstände zum nächsten Gegenstand vor und hinter dem Fahrzeug gelangen. Diese sind wichtig, um die aktuelle Fahrsituation einzuordnen. In Abbildung 2.1 werden die Bereiche abgebildet, die von einem 7er BMW durch Sensorik abgedeckt werden. Dabei wird häufig eine viel größere Fläche abgedeckt, als der Fahrer selbst gleichzeitig überschauen kann.

Ein wichtiger Bestandteil für die technische Umsetzung einiger Fahrassistenzsysteme sind sogenannte Regler. Diese werden im folgenden Abschnitt genauer betrachtet.

2.2 Basis Regelungstechnik

Die Hauptaufgabe eines Reglers ist es, einen voreingestellten Soll-Wert in einem System zu erreichen und zu halten. Der Ist-Wert kann in diesem System durch diverse Störgrößen verändert werden, welche der Regler ausgleichen muss. Ein klassisches Beispiel dabei wäre die Innentemperatur eines Fahrzeugs. Die Klimaanlage muss dabei die Ist-Innentemperatur auf die Soll-Innentemperatur anheben. Dabei nehmen Außentemperatur und Sonneneinstrahlung allerdings einen Einfluss auf die Innentemperatur, weshalb es nicht reicht, lediglich Luft auf die Soll-Temperatur zu erwärmen und in das Fahrzeug zu leiten. Im Folgenden werden drei Arten dieser Regler betrachtet, deren Konstruktionsidee zusammengesetzt einen vierten Regler, den sogenannten PID-Regler, ergibt.

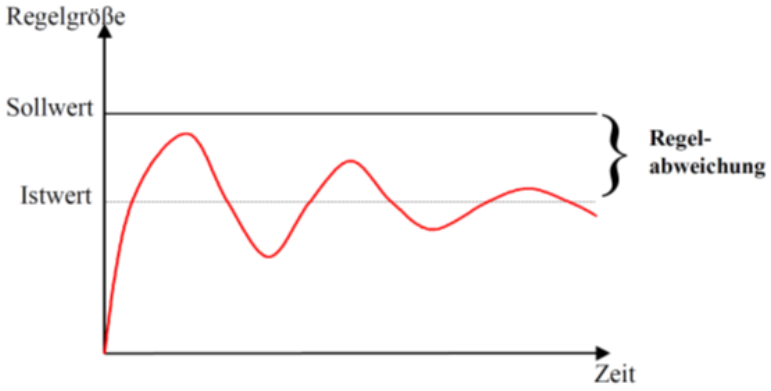


Abbildung 2.2: Diagramm eines P-Reglers [15].

2.2.1 Proportionalregler

Die Wirkungsweise eines Proportionalreglers (P-Regler) ist in Abbildung 2.2 dargestellt. Der P-Regler erhält die zeitabhängigen Eingangsgrößen $y, w, u \in \mathbb{R}$. Beim P-Regler ist die Stellgröße $u(t)$ direkt proportional zur Regelabweichung $e(t) = w(t) - y(t)$. Die Reglergleichung ist dann

$$u = u_0 + K_P * e \tag{2.1}$$

$$\Leftrightarrow e = (u - u_0)/K_P, \tag{2.2}$$

wobei K_P eine Konstante ist. Man kann erkennen, dass die Abweichung e klein wird, wenn K_P groß gewählt wird. Das bedeutet, dass bei einer groß gewählten Konstante K_P der Regler sehr schnell reagiert und sich die Regelgröße schnell dem Führungswert anpasst. Da der Regelkreis in der Rückführung eine Zeitverzögerung aufweist und der Regler dadurch nicht sofort reagieren kann, birgt eine große Konstante K_P die Gefahr des Überschwingens.

Der P-Regler hat den Vorteil, dass er einfach durch einen Widerstand zu realisieren ist. Außerdem reagiert er im Vergleich zu anderen Reglertypen schnell. Der größte Nachteil ist, dass der Führungswert auch dauerhaft nicht erreicht wird. Dies lässt sich auch nicht durch eine groß gewählte Konstante K_P vermeiden. Dieser Versuch würde nur zum Überschwingen des Reglers führen. Im schlechtesten Fall könnte es zu einer dauerhaften Schwingung kommen. Die bleibende Abweichung von der Führungsgröße wird am besten durch den Integralregler ausgeglichen.

2.2.2 Integralregler

Die Wirkungsweise eines Integralreglers (I-Regler) ist in Abbildung 2.3 dargestellt. Der I-Regler erhält ebenfalls die zeitabhängigen Eingangsgrößen $y, w, u \in \mathbb{R}$. Es wird eine

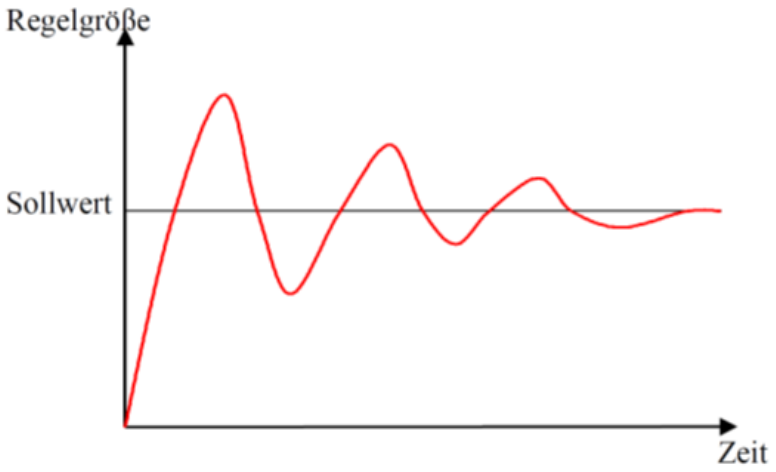


Abbildung 2.3: Diagramm eines I-Reglers [15].

Konstante K_I gewählt und die Regelabweichung $e(t) = w(t) - y(t)$ über einen Zeitraum integriert. Die Reglergleichung ist dann

$$u = K_I \int_0^t e. \tag{2.3}$$

Durch das Integral sind selbst dauerhafte Abweichungen im Zeitverlauf sichtbar, was zur Anpassung der Regelgröße führt und eine dauerhafte Abweichung unmöglich macht.

Da über eine bestimmte kurze Zeit integriert wird, erfolgt die Anpassung der Regelgröße sehr schwerfällig. Dies senkt zwar die Wahrscheinlichkeit, dass es zum Überschwingen oder gar dauerhaften Schwingungen kommt, senkt jedoch auch die Reaktionsgeschwindigkeit des Reglers. Dies macht den I-Regler alleine unbrauchbar für zeitkritische Anwendungen. Deshalb wird der I-Regler selten alleine verwendet, sondern mit dem P-Regler zum PI-Regler kombiniert. Da der P-Regler die schnellere Reaktion zeigt, gleicht dieser den Großteil der Abweichung aus. Da der P-Regler die Abweichung nicht vollständig ausgleichen kann, gleicht der langsamere I-Regler die dauerhafte Ungenauigkeit aus. Die Reaktion des PI-Reglers sieht man im Vergleich in Abbildung 2.5. Man kann erkennen, dass der PI-Regler deutlich genauer und schneller reagiert als der P- und I-Regler. Viele zeitkritische Systeme erfordern jedoch eine noch schnellere Reaktion, als der PI-Regler hat. Im Folgenden wird deshalb die Funktionsweise des Differentialreglers erläutert.

2.2.3 Differentialregler

Der Differentialregler (D-Regler) kommt dann zum Einsatz, wenn die Reaktionsgeschwindigkeit des P-Reglers noch nicht ausreicht. Der D-Regler reagiert auf eine plötzliche Regelabweichung durch die Ableitung dieser. Er erhält ebenfalls die zeitabhängigen Eingangs-

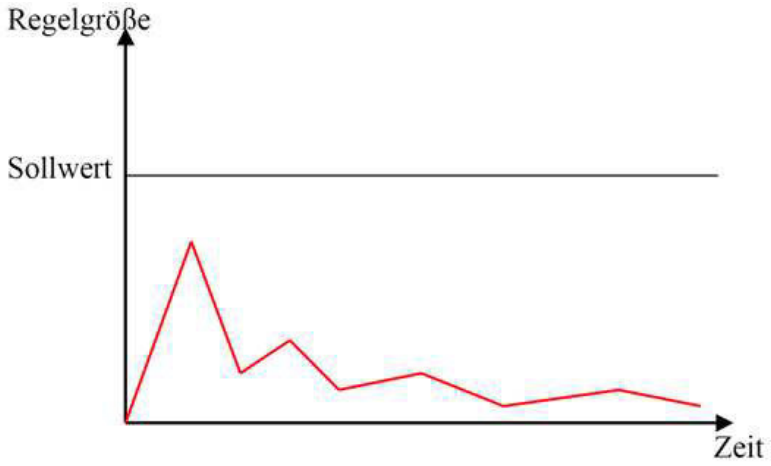


Abbildung 2.4: Diagramm eines D-Reglers [15].

größen $y, w, u \in \mathbb{R}$. Außerdem wird eine Konstante K_D gewählt. Die Reglergleichung ergibt sich dann als

$$u = K_D * de/dt. \tag{2.4}$$

In Abbildung 2.4 sieht man die Reaktion eines D-Reglers. Zunächst erfolgt eine sprunghafte Annäherung an die Führungsgröße durch eine große Ableitung. Durch die schnelle Verringerung der Regeldifferenz führt dies zu einer etwas kleineren Ableitung mit umgekehrtem Vorzeichen. Diese macht sich bemerkbar, sobald die neu eingestellte Regelgröße durch die Rückführung dem D-Regler gemeldet wird. Die Regeldifferenz wird dann wieder größer. Die Änderungen der Richtung der Kurve in Abbildung 2.4 finden immer dann statt, wenn die neu eingestellte Regelgröße dem Regler in der Rückführung gemeldet wurde.

Man kann erkennen, dass eine dauerhafte Regeldifferenz bestehen bleibt. Deshalb ist der Regler alleine unbrauchbar und wird nur in Kombination mit anderen Reglern verwendet, um eine schnellere Reaktion des Reglers zu ermöglichen.

2.2.4 PID-Regler

Es ist möglich, den P-, I- und D-Regler zu einem Regler zu kombinieren, da die Reaktion der einzelnen Regler zeitversetzt eintritt. Dazu werden die einzelnen Reglergleichungen wie folgt aufaddiert:

$$u = (K_P * e) + (K_I \int_0^t e) + (K_D * de/dt). \tag{2.5}$$

Um den PID-Regler zu realisieren, werden die einzelnen Komponenten parallel geschaltet. Dazu ist es wichtig, dass die Konstanten K_P, K_I und K_D geeignet gewählt werden. Falls diese zu klein gewählt werden, reagiert der Regler langsam. Bei zu groß gewählten Konstanten ist es möglich, dass der Regler zum Überschwingen neigt. Bei geeigneten Konstanten vereint der PID-Regler die Vorteile der einzelnen Komponenten. Zuerst rea-

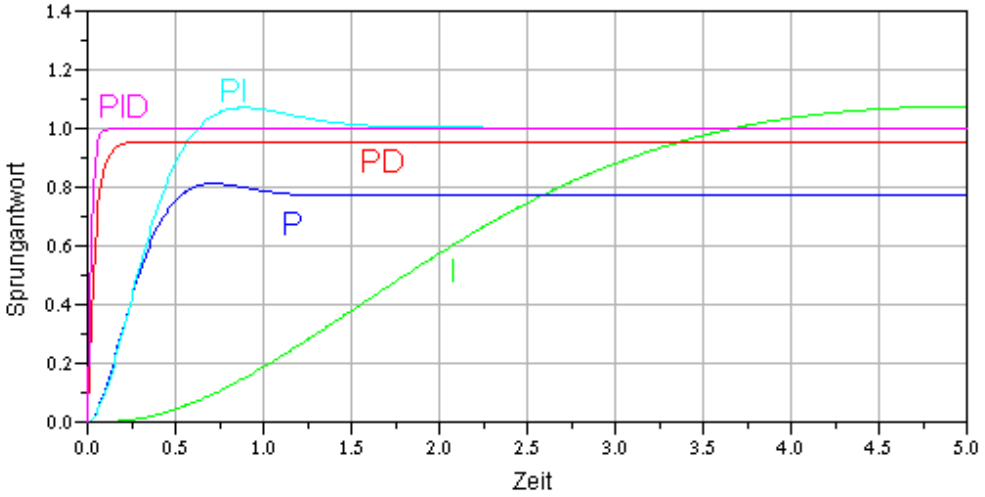


Abbildung 2.5: Vergleich der Reglertypen [22].

giert der D-Anteil, der zu einer schnellen ersten Anpassung an eine plötzliche Abweichung führt. Anschließend reagiert der P-Anteil und gleicht die übrige Abweichung in kurzer Zeit aus, behält jedoch eine dauerhafte Abweichung bei, die durch den I-Anteil ausgeglichen wird.

Die Reaktion eines PID-Reglers im Vergleich zu den anderen Reglertypen ist in Abbildung 2.5 zu sehen. Man kann erkennen, dass der PID-Regler am schnellsten und genauesten reagiert. Natürlich sind jegliche anderen Kombinationen der Reglertypen möglich. Der PID-Regler ist jedoch aufgrund dessen auch ein Bestandteil des von der Projektgruppe modellierten ACCs, das in Kapitel 13 vorgestellt wird. Im nächsten Abschnitt wird dazu erst einmal in die Grundlagen und die allgemeine Funktionsweise eines ACC eingeführt.

2.3 Adaptive Cruise Control

Adaptive Cruise Control (ACC) ist eine der vielen technischen Innovationen, die im letzten Jahrzehnt Einzug in die neu entwickelten Modelle der Automobilindustrie gefunden haben. Es handelt sich dabei um eine Weiterentwicklung der ursprünglichen Fahrgeschwindigkeitskontrolle (Tempomat), die für mehr Sicherheit und Komfort zusätzlich zum Halten einer bestimmten Geschwindigkeit auch noch äußere Faktoren miteinbezieht.

Der Name ACC für diese Technik wird größtenteils im VW-Konzern verwendet. Andere Autohersteller haben oft andere Namen für ihre *ACC-äquivalenten* Systeme. Jedes System, das den Namen ACC trägt, muss die *ISO-Norm 15622* (im folgenden *ACC-ISO-Norm* genannt) einhalten [13].

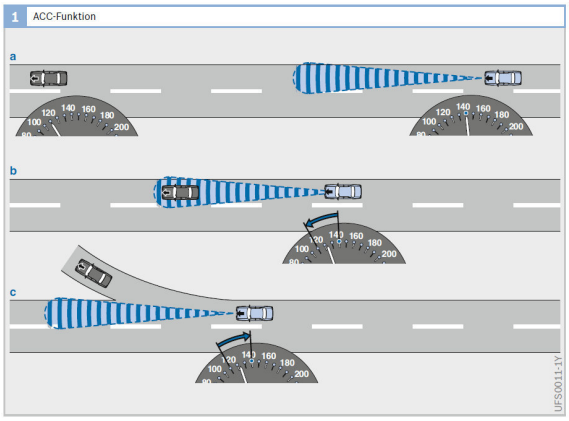


Abbildung 2.6: ACC-Anwendungsszenario: Fahrzeug mit freier Fahrbahn (oben), Fahrzeug mit eingeschränkter Fahrbahn (durch anderes Fahrzeug) (mitte), Fahrzeug mit nun freier, aber zuvor belegter Fahrbahn (unten). [19]

2.3.1 Funktionsweise

ACC kontrolliert mit Hilfe von Radarsensoren den Bereich vor dem Fahrzeug (ca. 200 m), um mögliche (fahrende) Hindernisse frühzeitig zu erkennen. Dafür steht den Herstellern das Frequenzband zwischen 76 und 77 GHz (Wellenlänge ca. 4 mm) zur Verfügung. Die Wunschgeschwindigkeit wird mittels Schalter oder Drehrädchen (meist am Lenker) eingestellt. Falls ein Hindernis erkannt wurde, soll das Fahrzeug angemessen durch Verzögerung oder Beschleunigung reagieren, um Unfälle zu vermeiden. Im Vergleich zu einer klassischen Geschwindigkeitsregelanlage muss vermehrt in andere Systeme des Autos eingegriffen werden.

Abbildung 2.6 zeigt ein mögliches Szenario. Im oberen Teil der Abbildung fährt das rechte Auto mit eingeschaltetem ACC mit einer Geschwindigkeit von 140 km/h und das linke vorausfahrende Fahrzeug mit nur 120 km/h. Da sich das linke Fahrzeug noch weit genug entfernt befindet, greift das ACC noch nicht ein. Im mittleren Teil der Grafik ist das rechte Fahrzeug auf das linke aufgefahren und die Abstandssensoren haben das vorausfahrende Fahrzeug als zu nah erkannt. Daher greift das ACC in das Bremssystem des Fahrzeugs ein und regelt die Geschwindigkeit von 140 km/h so weit herunter, dass der Sicherheitsabstand noch eingehalten wird. In der unteren Grafik verlässt das vorausfahrende Fahrzeug die Spur. Das ACC bemerkt, dass das Hindernis nicht mehr da ist und erhöht die Geschwindigkeit wieder auf die eingestellten 140 km/h.

Damit der Fahrer ACC ein- und ausschalten kann, gibt es meist am Lenkrad einige Bedienelemente. Außerdem gibt es neben Tachometer und Drehzahlmesser für gewöhnlich eine Anzeigemöglichkeit, damit der Fahrer seine Eingaben sieht und ggf. ändern kann. Die Bedienelemente unterscheiden sich von Modell zu Modell, umfassen aber für gewöhnlich (vgl. Abbildung 2.7) eine Ein/Aus-Taste (3), eine Aktivierungstaste (1), und meistens auch

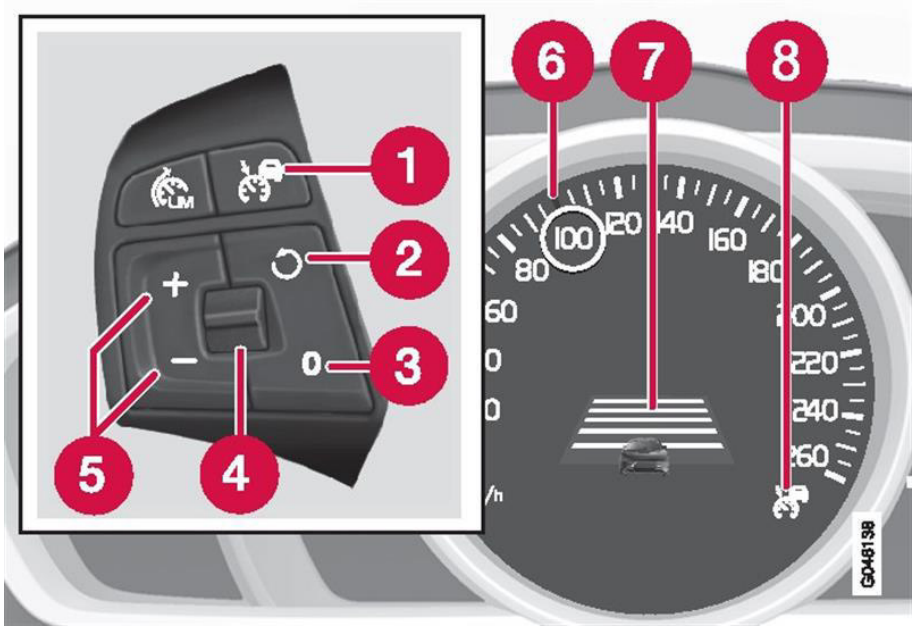


Abbildung 2.7: Beispiel für Bedienelemente an einem Lenkrad [24]. Dabei sind 7 Schalter erkennbar, über die ACC gesteuert werden kann.

Type	Manual clutch operation required	Active brake control
1a	Yes	No
1b	No	No
2a	Yes	Yes
2b	No	Yes

Abbildung 2.8: ACC Systemtypen nach *ACC-ISO-Norm* [13].

ein oder mehrere Paare von Tasten (4,5), mit denen die eingestellte Geschwindigkeit um 1, 5 oder 10km/h erhöht oder verringert werden kann. Einige Fahrzeuge haben zusätzlich eine Taste, die es ermöglicht, zum Beispiel nach einem Bremsvorgang die zuletzt eingestellte Geschwindigkeit zu reaktivieren (2). Manche Tasten sind je nach aktuellem Zustand auch doppelt belegt.

2.3.2 Klassifizierung und Kontrollstrategien

ACC-Systeme lassen sich in mehrere Typen einordnen. Die Typen hängen davon ab, ob es sich bei dem Fahrzeug um eines mit Schalt- oder Automatikgetriebe handelt und ob das Fahrzeug *Active Brake Control* unterstützt, eine Systemfunktion, die dem System auch aktiven Bremseingriff erlaubt. Daraus lassen sich die 4 Typen ableiten, die in Abbildung 2.8 zu sehen sind.

Außerdem können ACC-Systeme für unterschiedliche Kurvenradien ausgelegt sein (vgl. Abbildung 2.9). Ein kleinerer Kurvenradius bedeutet dabei eine engere Kurve. Da die Sensoren

Performance class	Curve radius capability
I	No performance capability claimed
II	≥ 500
III	≥ 250
IV	≥ 125

Abbildung 2.9: Performance-Klassen nach *ACC-ISO-Norm* [13]. Die Angaben der Kurvenradien sind in Metern.

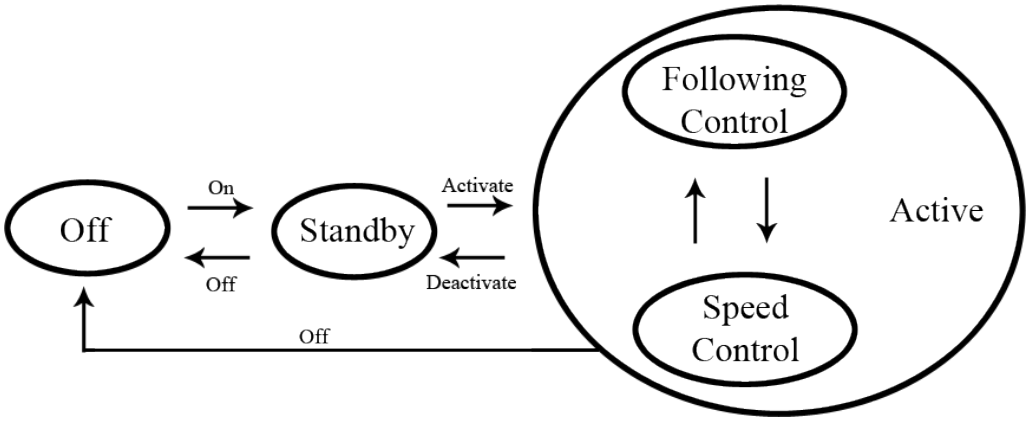


Abbildung 2.10: ACC Automat nach *ACC-ISO-Norm* [13].

geradeaus ausgerichtet sind, ist Objekterkennung und Zuordnung bei kleineren Kurvenradien schwieriger. Eine höhere Performance-Klasse stellt also höhere Anforderungen dar. ACC-Systeme mit Performance-Klasse 1 sind nicht für Kurvenfahrten ausgelegt.

Zusätzlich spezifiziert die *ACC-ISO-Norm* die Funktionsweise von ACC mit einem Zustandsautomaten (vgl. Abbildung 2.10). Es gibt drei Zustände: *ACC-off*, *ACC-standby* und *ACC-active*. Der Zustand *ACC-off* bedeutet, dass ACC vom Fahrer noch gar nicht eingeschaltet wurde. Wurde ACC über einen Schalter am Lenkrad (vgl. Unterabschnitt 2.3.1) eingeschaltet, wechselt der Automat in den Zustand *ACC-standby*. Das bedeutet, dass noch keine Geschwindigkeits- oder Abstandskontrolle durchgeführt wird, der Fahrer diese aber jederzeit direkt einschalten oder ACC wieder in den Zustand *ACC-off* zurückversetzen kann. Wenn der Fahrer eine Wunschgeschwindigkeit festlegt, die eine bestimmte Mindestgeschwindigkeit v_{low} übersteigen muss, wechselt der Automat in den Zustand *ACC-active*. v_{low} ist von der *ACC-ISO-Norm* auf mindestens 5 m/s, also 18 km/h festgelegt.

ACC-active besteht aus zwei Unterzuständen *ACC-speed-control* und *ACC-following-control*. Falls die Sensoren im Bereich vor dem Fahrzeug kein anderes Fahrzeug detektieren, wird *ACC-speed-control* angenommen. Dieser Zustand beschreibt die Basisfunktion des Tempomaten, die nur auf das Halten einer eingestellten Geschwindigkeit ausgerichtet ist. Wurde jedoch ein anderes Fahrzeug detektiert, das langsamer ist als die eingestellte Geschwindig-

keit, wird zu *ACC-following-control* gewechselt. Dieser Zustandswechsel kann nicht vom Fahrer initiiert werden, sondern wird vom System automatisch vorgenommen. Gibt es mehrere vorausfahrende Fahrzeuge, wählt das System selbstständig das relevante Fahrzeug aus. In dem Fall, dass *ACC-following-control* aktiviert ist, folgt das Fahrzeug dem vorausfahrenden Fahrzeug mit konstantem Abstand. Wenn sich das Hindernis entfernt, zum Beispiel durch einen Spurwechsel (vgl. Abbildung 2.6), nimmt das System wieder den Zustand *ACC-speed-control* an, bis der Fahrer diese deaktiviert oder ACC komplett ausschaltet. Fällt die Fahrzeuggeschwindigkeit im Zustand *ACC-following-control* unter die Mindestgeschwindigkeit v_{low}, so ist ein erneuter Beschleunigungsvorgang ausgeschlossen. Wahlweise kann das System in diesem Fall auch in den Zustand *ACC-standby* wechseln. Sollte neben ACC noch ein konventioneller Tempomat eingebaut sein, darf das Fahrzeug nicht selbstständig zwischen den beiden Systemen wechseln und muss dem Fahrer vor allem immer deutlich machen, welches System aktiviert ist und derzeit verwendet wird.

Die Deaktivierung von ACC erfolgt in der Regel durch einen Bremsvorgang des Fahrers oder durch Ausschalten (*ACC-off*) von ACC, im Falle eines Bremsvorgangs jedoch nur, wenn die in dem Moment durch den Fahrer ausgeübte Bremskraft die des ACC übersteigt. Eine Geschwindigkeitserhöhung durch Betätigung des Gaspedals führt nicht zur Deaktivierung. Wird das Gaspedal wieder losgelassen, tritt die zuletzt eingestellte Geschwindigkeit wieder in Kraft. Wenn durch einen Bremsvorgang der Einsatz von Stabilitätsprogrammen wie ABS oder ESP notwendig wurde, wird daraufhin bei manchen Fahrzeugen ACC automatisch abgeschaltet. Des Weiteren ist eine automatische Deaktivierung möglich, wenn die Geschwindigkeit während eines automatischen Bremsvorgangs unter $v_{low} = 5\,\mathrm{m/s}$ fällt. In diesem Fall darf das System allerdings nicht plötzlich die Bremskraft verringern.

2.3.3 Fehlerverhalten

In Fehlerfällen gibt erneut die *ACC-ISO-Norm* Aufschluss über das erwartete Verhalten. Abbildung 2.11 und Abbildung 2.12 zeigen mögliche Fehler und die Systemreaktionen. Wichtig ist, dass der Fahrer sofort informiert wird und die Fehlermeldung erst mit Behebung des Fehlers verschwindet. Nach einem Fehlerfall muss nach *ACC-ISO-Norm* ein vollständiger Selbsttest ausgeführt werden. Im Fehlerfall startet ACC nicht.

Failure in subsystem		Failure occurs whilst acc is applying: Deceleration control	Engine control
1	Engine	ACC engine control mode shall be relinquished.	ACC engine control mode shall be relinquished.
2	Gearbox	ACC control mode shall be relinquished.	ACC engine control mode shall be relinquished.
3	Detecting and ranging sensor	Shall maintain the same strategy as the time before before fault at least as long as $v > v_{low}$ The system shall be switched off immediately after driver intervention by brake or accelerator pedal or ACC off switch.	ACC engine control mode shall be relinquished.
4	ACC controller	ACC control mode shall be relinquished.	ACC control mode shall be relinquished.

Abbildung 2.11: Fehlerreaktionen für ACC Typ 1 nach *ACC-ISO-Norm* [13].

Failure in subsystem		Failure occurs whilst acc is applying: Brake control	Engine control
1	Engine	Should maintain braking as required at least for the actual/current braking manoeuvre.	ACC engine control shall be relinquished.
2	Brake system[a]	ACC control shall be relinquished. If the brake system failure is not total during an active brake manoeuvre, the system may finish the current braking manoeuvre before the ACC control is relinquished completely.	ACC engine control shall be relinquished.
3	Detecting and ranging sensor	Should initiate a controller strategy starting with the last valid braking command. Braking shall not be released suddenly in this case. The system shall be switched off immediately after driver intervention by brake or accelerator pedal or ACC off switch.	ACC control shall be relinquished.
4	ACC controller	ACC control shall be relinquished.	ACC control mode shall be relinquished.
[a] If a malfunction within the gear controller occurs, the brake will be able to handle the deceleration function.			

Abbildung 2.12: Fehlerreaktionen für ACC Typ 2 nach *ACC-ISO-Norm* [13].

2.3.4 Einschränkungen

ACC hat einige Einschränkungen in der Funktionsweise. Der *Long-Range-Radarsensor* tastet zwar einen Bereich von 200 m vor dem Fahrzeug ab, erfasst jedoch die komplette Breite der eigenen Fahrspur erst in 40 m Entfernung. Demnach ist der Gebrauch von ACC aktuell nur für einen Geschwindigkeitsbereich von 30 km/h bis 160 km/h, beziehungsweise je nach Hersteller 200 km/h, verwendbar.

Auch die Beschleunigung, sowohl positiv als auch negativ (Verzögerung), ist begrenzt, um in dem verwendeten Bereich eine einwandfreie Funktion garantieren zu können. Die maximale Beschleunigung liegt zwischen 0,6 und 1 m/s^2(lt. *ACC-ISO-Norm* maximal 2 m/s^2), während die maximale Verzögerung meist bei 2,5 m/s^2 liegt (lt. *ACC-ISO-Norm* maximal 3,5 m/s^2). Wenn eine Verzögerung über 2,5 m/s^2 benötigt wird, muss der Fahrer selbst

übernehmen. Da die maximal mögliche Verzögerung auf offener Straße ca. vier Mal so groß ist, ist eine maximale Verzögerung von 2,5 m/s^2 eine relativ große Funktionseinschränkung und bietet noch deutlichen Spielraum für zukünftige ACC-Generationen. Des Weiteren darf bei automatischer Verzögerung die Verzögerungsänderung nicht mehr als 2,5 m/s^2 betragen.

Aus den beiden oben genannten Punkten, dem auf 200 m begrenzten Radarsensor und der auf 2,5 m/s^2 begrenzten maximalen Verzögerung, ergibt sich außerdem eine maximale Geschwindigkeitsdifferenz zwischen dem eigenen und dem vorausfahrenden Fahrzeug. Dies ist eine weitere Funktionseinschränkung, die jedoch nicht so einfach behoben werden kann. Um diese Differenzgeschwindigkeit zu erhöhen, müsste entweder die Reichweite des Radarsensors erhöht werden oder/und die maximale Verzögerung erhöht werden.

Da ACC in erster Linie eine Komfort- und keine Sicherheitsfunktion ist, wäre die Erhöhung der maximalen Verzögerung mit einem Komfortverlust verbunden und daher nicht ratsam. Eine Erhöhung der Reichweite des Radarsensors ist jedoch ebenfalls mit Problemen verbunden. Da in 40 m die volle Breite der eigenen Fahrspur erfasst wird, ist der erfasste Bereich in 200 m bereits sehr breit. Mit zunehmender Breite erhöht sich auch die Schwierigkeit, Fahrzeuge einer bestimmten Fahrspur zuzuordnen. Dasselbe Problem würde sich bei Kurven mit engerem Radius (unter 1000 m) ergeben, da der Erfassungsbereich des Radarsensors durch Randbebauungen, Bäume o.Ä. eingeschränkt sein kann. Des Weiteren steigt bei steigender Geschwindigkeitsdifferenz zwischen eigenem und vorausfahrendem Fahrzeug die Wahrscheinlichkeit eines Überholmanövers. Um diese Differenz auszugleichen, müsste ACC früher die Geschwindigkeit reduzieren, da der Fahrer aber mit großer Wahrscheinlichkeit überholen möchte, würde er diese Geschwindigkeitsverringerung als negativ empfinden.

Für das Folgeverhalten in Kurvenfahrten gibt es auch einige Einschränkungen. Generell muss ein Fahrzeug mit einem ACC-System der Performance-Klassen II, III oder IV in der Lage sein, einem vorausfahrenden Fahrzeug mit der gleichmäßigen Geschwindigkeit v_{circle} zu folgen. Dabei gibt v_{circle} die maximale Geschwindigkeit in einer Kurve, bei gegebener maximaler Querbeschleunigung $a_{lateral\_max}$, an. Die folgende Tabelle 2.1 soll die vorgeschriebenen Fähigkeiten von ACC in Kurvenfahrten demonstrieren. Die Querbeschleunigung a ist von der *ACC-ISO-Norm* vorgegeben, genauso wie die Performance-Klassen und die dazugehörigen minimalen Radien (vgl. Unterabschnitt 2.3.2). Aus der Wurzel des Produkts der beiden Werte ergibt sich dann für den jeweiligen Radius die minimale Geschwindigkeit des Zielfahrzeugs, dem das eigene Fahrzeug mit ACC folgen können muss.

Des Weiteren ist ACC auf die Erfassung von fahrenden Objekten ausgelegt, stehende werden ignoriert. Die ACC-Einheit errechnet die Relativgeschwindigkeit des erfassten Objekts zum eigenen Fahrzeug. Wenn das erfasste Objekt langsamer ist, ist diese Relativgeschwindig-

Tabelle 2.1: Fähigkeiten von ACC in Kurvenfahrten [13]

Performance-Klasse	R_{min} [m]	$a_{lateral\_max}$ [m/s^2]	v_{circle} [km/h]
II	500	2,0	114
III	250	2,3	86
IV	125	2,3	61

keit negativ. Die Summe aus Relativgeschwindigkeit und eigener Geschwindigkeit ergibt die Geschwindigkeit des erkannten Objekts. Ist diese kleiner als 5 km/h, wird dieses als stehendes Objekt ausgeschlossen. Der Ausschluss erfolgt hauptsächlich aus zwei Gründen. Zum einen reicht die relativ geringe maximale Verzögerung durch die Auslegung von ACC als Komfortsystem nicht aus, um das Fahrzeug rechtzeitig zu stoppen, zum anderen ist die Objekterkennung hochkomplex und die Wahrscheinlichkeit, dass stehende Objekte z.B. am Straßenrand fälschlicherweise der eigenen Fahrspur zugeordnet werden würde, ist daher zu groß.

Daher werden stehende Objekte bislang nur bei niedrigen Geschwindigkeiten analysiert. Wenn ACC ein stehendes Objekt in der eigenen Fahrspur erkennt, wird nicht automatisch beschleunigt. Außerdem werden für die Folgeregelung (*ACC following control*) nur fahrende Fahrzeuge berücksichtigt oder Fahrzeuge, die zwar angehalten haben, aber vorher bereits als fahrend erfasst wurden. Diese Regelung verringert die Wahrscheinlichkeit auf ein Minimum, dass Objekte am Fahrbahnrand fälschlicherweise eine Reaktion des ACC hervorrufen.

2.3.5 Sicherheit

Wie für andere Systeme gibt es auch für ACC ein Sicherheitskonzept. Ziel eines solchen Konzepts ist es, die Robustheit des gesamten Systems sicherzustellen, indem durch ACC hervorgerufene kritische Fahrsituationen vermieden werden. Gleichzeitig sind die Funktionseinbußen durch Sicherheitseinschränkungen zu minimieren. Wichtig hierbei ist, dass im Falle einer Fehlfunktion von ACC keines der anderen Subsysteme wie Antrieb, Bremssystem und Ähnliches beeinträchtigt wird. Außerdem muss ACC die Radarstrahlung einstellen und darf nicht mehr in die Antriebssteuerung eingreifen, sodass dem Fahrer wieder die aktive komplette Kontrolle über das Fahrzeug übertragen wird.

Das ACC-Sicherheitskonzept basiert hauptsächlich auf Redundanz in der Hard- und Software. Das ACC-Steuergerät beinhaltet zwei Prozessorkerne und ist somit gut geeignet für ein redundanzbasiertes Sicherheitskonzept. Dieses besteht aus drei Ebenen: In der ersten Ebene werden ausschließlich die Peripherieeinheiten des ACC-Systems untersucht. Komponenten wie Radartransceiver, CAN-Datenbus und Sensoren werden unabhängig voneinander von beiden Controllern des Steuergeräts betrachtet.

Auf der zweiten Ebene werden die in den Controllern ausgeführten (Berechnungs-) Funktionen überprüft. Auch hier arbeiten beide Controller voneinander unabhängig. Die Redundanz ist somit wie bei Ebene 1 gegeben. Außerdem werden auf der zweiten Ebene auch Steuergeräte von den vernetzten anderen Subsystemen verwendet, um die von den beiden ACC-Controllern über den CAN-Bus gesendeten Nachrichten mittels Checksummenprüfung etc. auf Plausibilität zu überprüfen.

Die dritte Ebene stellt die gegenseitige Kontrolle der beiden Controller sicher. Kontrollberechnungen, die auf einem Controller ausgeführt werden, werden vom anderen Controller analysiert und auf Korrektheit geprüft und umgekehrt. Außerdem werden Checksummenprüfungen und Timing-Überwachungen der gegenseitigen Kommunikation durchgeführt.

Kapitel 3

Domänenspezifische Sprachen

Dieses Kapitel behandelt die zum Verständnis der folgenden Kapitel notwendigen Grundlagen zum Thema der domänenspezifischen Sprachen. In der Projektgruppe werden drei Sprachen erstellt, mit der die im vorigen Kapitel erklärten Fahrassistenzsysteme modelliert werden können. Zur Verwendung dieser soll kein Entwicklerhintergrund nötig sein.

3.1 Einleitung

Eine domänenspezifische Sprache (*Domain Specific Language* (DSL)) ist eine formale Sprache, welche auf ein bestimmtes Problemfeld spezialisiert ist. DSLs stehen damit im Kontrast zu klassischen universellen Programmiersprachen wie Java oder C. DSLs werden im Informatik-Umfeld oft benutzt, auch wenn dem Nutzer nicht klar ist, dass es sich um eine DSL handelt. So sind zum Beispiel *Structured Query Language* (SQL) , LaTeX, *Hypertext Markup Language* (HTML) oder die Backus-Naur-Form domänenspezifische Sprachen. Jede dieser Sprachen ist auf eine bestimmte Domäne zugeschnitten. In den folgenden Kapiteln wird zuerst genau definiert, worum es sich bei einer DSL handelt und welche Vor- und Nachteile diese bieten kann.

3.1.1 Definition

Domänenspezifische Sprachen sind Programmiersprachen, welche eine begrenzte Ausdrucksstärke haben und deren Anwendung sich auf eine Domäne beschränkt. Vier Kernelemente können die Merkmale von DSLs besonders herausstellen[9]:

1. Das erste Kernelement ist, dass die Sprache eine Programmiersprache ist. Dies bedeutet, dass die DSL von Menschen benutzt wird, um einer Maschine Befehle zu geben. Dabei sollte diese von Menschen verstanden werden und vom Computer ausgeführt werden können.

2. Zweites Kernelement besagt, dass die DSL nicht nur aus einzelnen Ausdrücken besteht, sondern diese auch zusammengesetzt sein können und sich die Sprache somit fließend anfühlt.

3. Das dritte Kernelement bezieht sich auf die begrenzte Ausdrucksstärke der Sprache. Eine DSL sollte nur Features unterstützen, welche benötigt werden, um eine Domäne abzubilden. Wenn zu viele Sprachkonstrukte unterstützt werden, besteht die Gefahr, dass die Sprache keine DSL mehr ist, sondern eine *General Purpose Language* (GPL) , wie zum Beispiel Java.

4. Das letzte Kernelement ist der Fokus auf eine Domäne. Durch diesen Fokus wird die begrenzte Ausdrucksstärke gerechtfertigt. Weiterhin wird so genug Mehrwert geschaffen, dass die DSL einen Anwendungszweck hat. Andernfalls würden GPLs eine bessere Alternative darstellen.

Semantisches Modell

Weiterhin haben viele DSLs ein sogenanntes Semantisches Modell [9]. Dabei handelt es sich um ein unterliegendes Modell der DSL. Wenn Code, welcher in der DSL geschrieben wurde, gelesen wird, wird das Modell mit der Semantik des Codes befüllt. Wird zum Beispiel ein endlicher Automat durch eine DSL beschrieben, dann besteht das semantische Modell aus Zuständen, Zustandsübergängen und Aktionen. Beschreibt nun die DSL einen endlichen Automaten mit den Zuständen A und B, werden im semantischen Modell zwei Instanzen der Zustandsklasse erzeugt, jeweils eine Instanz für Zustand A und B. Dasselbe gilt für Zustandsübergänge und Aktionen.

Ein solches Modell hilft dabei, auf der von der DSL abgedeckten Domäne zu arbeiten. Im besten Fall sollte das Modell auch ohne die DSL verwendbar sein, da so gewährleistet werden kann, dass die Domäne gerecht abgebildet ist und nicht falsch abstrahiert wurde. Des Weiteren wird durch die Unabhängigkeit ermöglicht, dass Testen auch ohne vorliegende DSL möglich ist. Zum einen kann das Modell auf semantische Korrektheit getestet werden, ohne dass Code einer DSL benötigt wird. Zum anderen kann der Parser getestet werden, ob bei gegebenem Code einer DSL das Modell wie erwartet befüllt wird.

Das semantische Modell macht es weiterhin einfacher, Code zu generieren. In dem Modell sind alle benötigten Informationen hinterlegt und es ist möglich, eine semantische Validation vor der Generation auf dem Modell durchzuführen.

Ein weiterer wichtiger Vorteil eines semantischen Modells ist es, dass die Semantik und das Parsen getrennt werden. DSLs können schnell sehr komplex werden und die Unterteilung hilft dabei, die Problematik in zwei separate Teile zu zerlegen.

Durch die vielen Vorteile sollten DSLs immer ein semantisches Modell als Grundlage haben.

Interne und Externe DSLs

DSLs können in zwei Gruppen aufgeteilt werden[9]. Zum einen gibt es die externen DSLs. Diese sind Sprachen, welche eine andere Syntax als die der Hauptsprache der Anwendung verwenden. In den meisten Fällen wird die externe DSL dann in der Hauptsprache geparsed und ausgewertet. Zu den bekanntesten Vertretern der externen DSLs gehören SQL, welches Datenbankeninteraktion als Domäne hat, sowie HTML.

Wichtig ist hier die Unterscheidung zwischen einer GPL und einer externen DSL. Es gibt GPLs, die einen Fokus auf eine Domäne haben, aber trotzdem eine GPL sind. Dazu gehört unter anderem die Sprache R, welche sich auf die Domäne der Statistik fokussiert, aber trotzdem die Ausdrucksstärke einer GPL hat. Da somit das Paradigma der begrenzten Ausdrucksstärke der Sprache verletzt ist, ist R keine externe DSL. Ein wichtiger Indikator für die Unterscheidung kann die Turing-Vollständigkeit sein. Sobald eine Sprache Turing-vollständig ist, kann sie für Probleme benutzt werden, welche nicht durch die Domäne abgedeckt werden.

Die andere Gruppe ist die der internen DSLs. Im Gegensatz zu den externen DSLs bewegt man sich hier innerhalb einer GPL, wobei man sich jedoch auf eine kleine Teilmenge der Sprache beschränkt. Diese Teilmenge soll sich wie eine eigene Sprache anfühlen. Ein bekanntes Beispiel für eine interne DSL ist Ruby on Rails unter Ruby [21].

Hier ist der Unterschied zwischen einer internen DSL und einem Framework oder API wichtig. Eine Eigenschaft, welche beim Differenzieren hilft, ist, dass interne DSLs versuchen eine Domäne so abzubilden, dass auch Nicht-Entwickler diese verstehen können. Außerdem kann ein Indikator sein, dass eine nicht technische Domäne abgebildet wird.

Weiterhin kann zwischen textuellen und graphischen DSLs unterschieden werden. Die DSLs, welche bis jetzt genannt wurden, sind allesamt textuell. Im Gegensatz dazu gibt es die graphischen DSLs. Diese DSLs werden nicht als Code repräsentiert, sondern haben eine graphische Repräsentation. Die Sprachelemente sind dabei graphische Objekte, die verwendet werden, um die gewünschte Logik zu beschreiben. Bekannte graphische DSLs sind *Business Process Model and Notation (BPMN)* 2.0, welches die Domäne der Geschäftsprozesse abdeckt, oder *Unified Modeling Language (UML)*.

3.1.2 Language Workbenches

Neben den DSLs gibt es auch die sogenannten Language Workbenches [9]. Bei diesen handelt es sich um Werkzeuge, die beim Entwickeln von DSLs unterstützen. Außerdem wollen die Workbenches dem Problem der fehlenden Unterstützung für externe DSLs entgegenwirken.

Wenn DSLs von Hand ohne Werkzeugunterstützung erstellt werden, ist es ein nicht zu unterschätzender Aufwand, auch entsprechende Entwicklungswerkzeuge für die Sprache bereitzustellen. Language Workbenches helfen sowohl den Parser zu entwickeln, als auch entsprechende Plug-Ins für Entwicklungsumgebungen bereitzustellen.

Ohne Language Workbenches ist es ein erheblicher Aufwand, graphische DSLs zu erstellen. Neben dem Parser muss ein eigener Editor erstellt werden, der die graphische Repräsentation für den Anwender verständlich konstruierbar macht sowie eine Validierung ermöglicht und dem Benutzer direkt Feedback über die Strukturen gibt.

Language Workbenches vereinfachen das Entwickeln von Sprachen und erstellen Hilfsmittel, die bei der Verwendung von neuen Sprachen helfen. So ist es möglich, viel Zeit bei der Implementierung einer DSL zu sparen. Weiterhin sind die daraus entstandenen Sprachen auch von Nicht-Entwicklern einfacher zu verwenden, da genug Hilfsmittel bereitstehen. Zwei bekannte Language Workbenches sind Xtext [26] für textuelle DSLss und CINCO [2] für graphische DSLss. Die Grundlagen beider Werkzeuge werden jeweils in Abschnitt 3.3 und Abschnitt 3.2 erläutert.

3.1.3 Vorteile von domänenspezifischen Sprachen

Es gibt mehrere Gründe, warum man DSLs für spezielle Anwendungsfälle verwenden sollte. Zwei wichtige Vorteile sind, dass die Produktivität von Entwicklern und die Kommunikation mit Domänenexperten verbessert werden kann [9].

Die Produktivitätssteigerung wird durch die vereinfachte Darstellung der Problematik als DSL gewährleistet. Es ist einfacher, den Code beim Lesen zu verstehen und semantische Fehler zu finden. Da DSLs, im Gegensatz zu GPLs, eine verringerte Ausdruckskraft haben, die auf eine Domäne spezifiziert ist, nimmt auch die Anfälligkeit für Fehler ab.

Ein weiterer Vorteil ist die Kommunikation mit Domänenexperten. In vielen Softwareprojekten kann es Probleme in der Kommunikation geben. Durch eine DSL kann die Implementierung in einer für den Kunden verständlichen Sprache umgesetzt werden. Der Kunde kann dann auf mögliche Fehler in der Implementierung hinweisen, da dieser den Prozess des zu erzeugenden Produkts besser verstehen kann als ein Entwickler. Diesen Vorteil haben aber nicht alle DSLs, da es Sprachen gibt, die es dem Entwickler einfacher machen sollen, bestimmte Aufgaben zu erfüllen. Diese wird ein Kunde nicht zu Gesicht bekommen. Darunter können zum Beispiel reguläre Ausdrücke fallen. Weiterhin ermöglicht eine gut zugeschnittene und einfache DSL einem Domänenexperten selbst Code zu schreiben. Dadurch kann garantiert werden, dass es für diesen Teil weniger semantische Fehler gibt. Selbst wenn es noch syntaktische Fehler in dem Code vom Domänenexperten gibt, können diese schnell von einem Entwickler beseitigt werden.

DSLs können außerdem genutzt werden, um Definitionen oder Logik aus der Compile-Zeit in die Laufzeit zu bewegen. So werden DSLs benutzt, um Konfigurationen zur Laufzeit zu laden. Dies ist übersichtlicher und schneller zu definieren, als XML-Konfigurationsdateien zu erstellen. Des Weiteren können DSLs in eine andere Sprache wie zum Beispiel JAVA übersetzt werden, um dann von einem Programm ausgeführt zu werden. Dadurch können Domänenexperten ihre Anforderungen als DSL schreiben und diese werden dann vom Programm ausgewertet und entsprechend ausgeführt.

3.1.4 Nachteile von domänenspezifischen Sprachen

Leider gibt es auch Nachteile bei der Benutzung von DSLs.

Ein Nachteil ist, dass jede DSL erst einmal neu erlernt werden muss. Die Sprachen haben zwar nicht denselben Umfang wie die General Purpose Language, aber trotzdem ist es ein Aufwand, der von jedem Entwickler, der mit der DSL in Berührung kommt, betrieben werden muss. Dies erschwert das Einführen von neuen Entwicklern in das Projekt.

Ein weiterer Nachteil sind Kosten, welche beim Erstellen der Sprache entstehen. Damit ist nicht nur die initiale Arbeit an der Sprache gemeint, sondern auch die weiterführende. So kann es nötig sein, die DSL anzupassen, wenn sie einen Bereich der Domäne nicht korrekt abbildet. Wenn nun eine DSL nur eine sehr kleine Domäne abbildet, kann das Erstellen und Warten der Sprache erheblich mehr Zeit kosten, als durch die Anwendung und verringerte Fehleranfälligkeit der Sprache eingespart werden kann.

Außerdem kann es zum Problem werden, wenn die Domäne nicht verständlich abstrahiert wird. So kann die Verständlichkeit der Sprache verloren gehen, wenn falsch abstrahiert wird. Weiterhin besteht die Gefahr, dass Domänenexperten die DSL nicht mehr verstehen und damit nicht mehr auf Fehler hinweisen können.

3.2 CINCO - Funktionsweise und Nutzung

Die CINCO Meta Tooling Suite [2] ist eine Modellierungsumgebung, die seit 2013 am Lehrstuhl für Programmiersysteme der Fakultät Informatik an der TU Dortmund entwickelt wird. Modelliert werden graphische domänenspezifische Sprachen – genauer gesagt sind es Graphenstrukturen, die sich aus verschiedenen Knoten und Kanten zusammensetzen. Das Hauptaugenmerk von CINCO liegt nicht darauf, möglichst generisch zu sein, sondern es sehr einfach zu machen, graphische DSLs zu entwickeln: Der Entwickler spezifiziert einmalig, welche Knoten- und Kantentypen es gibt und in welchen Konstellationen sie auftreten dürfen. Daraufhin wird ein entsprechender Editor (das sog. CINCO-Produkt) erzeugt, in dem man mit der entwickelten Sprache Modelle konstruieren kann.

Es ist dabei möglich, nicht nur Aussagen über die Syntax zu machen, der die Modelle entsprechen sollen, sondern auch über die Semantik. Einerseits können *Checks* spezifiziert werden, die stets überprüfen, ob ein gebautes Modell sinnvoll ist (z.B. keine Zyklen enthält). Andererseits ist es möglich, einen Generator anzubinden, der das Modell interpretiert und daraus Code in einer anderen Sprache erzeugt (z.B. C++ oder JAVA, aber theoretisch sind bezüglich der Zielsprache keine Grenzen gesetzt). CINCO verfolgt dabei das Prinzip der *Full Code Generation* – sämtlicher Code wird automatisch erzeugt und es ist nicht vorgesehen, dass der Entwickler den erzeugten Code manipuliert.

Der letzte wichtige Aspekt von CINCO ist, dass es einfach ist, eigene bzw. andere Bibliotheken anzubinden. Dadurch ist es beispielsweise möglich, im Modell Teile auch durch eine textbasierte DSL abzubilden. CINCO basiert auf der Eclipse *Rich Client Platform (RCP)*. Dies bedeutet, dass im Prinzip alle Funktionalitäten in Form von Plugins für Eclipse realisiert werden. Der große Vorteil ist, dass dadurch viele bestehende Technologien wiederverwendet werden können. So wird das Eclipse Modeling Framework verwendet, um die Modelle intern objektorientiert darzustellen, und es wird vom Graphiti Framework Gebrauch gemacht, um den Grapheditor anzuzeigen. Weiterhin lassen sich praktisch alle anderen Features von Eclipse, wie z.B. der Projektexplorer, die JAVA Development Tools und Versionskontrollsysteme direkt nutzen. Auch die CINCO-Produkte basieren auf Eclipse RCP – beim Generierungsschritt werden neue Plugins erzeugt, die dann in einer separaten Eclipse-Instanz geladen werden.

Innerhalb der Projektgruppe nimmt CINCO eine zentrale Rolle ein, da sowohl die Metamodelle der AutoDSL (siehe Kapitel 7) als auch der SzenarienDSL (siehe Kapitel 9) darin verfasst sind.

3.2.1 Modellierung mit CINCO

CINCO verwendet drei domänenspezifische Sprachen, die bei der Erstellung von Projekten eine Rolle spielen: Die Definition von Graphen in der *Meta Graph Language* (MGL), das Aussehen der Elemente im Editor in der *Meta Style Language* (MSL) und eine Produktdefinition *Cinco Product Definition* (CPD), die angibt, welche Graphmodelle und Styles es im Projekt gibt und gewisse andere Eigenschaften des Produkts konfigurierbar macht.

In der MGL werden Graphen anhand von Knoten, Kanten und Containern definiert. Container sind spezielle Knoten, die wiederum andere Knoten enthalten können. Bei sämtlichen Knotentypen kann mittels outgoingEdges bzw. incomingEdges bestimmt werden, welche Kantentypen vom Knoten ausgehen bzw. in ihn eingehen dürfen. Durch ein Anfügen von [min, max] an den Namen des Kantentyps kann über die minimale- bzw. maximale Anzahl von Kanten verfügt werden, wobei „∗“ beliebig viele bedeutet.

Schlüsselwort	**Funktion**
graphModel	Wird zur Definierung eines eigenen Graphtypen verwendet. Im folgenden Block werden die Graphkomponenten definiert.
package	Definiert den Java-Paketnamen.
nsURI	Eindeutiger Identifier für das zu erzeugende Ecore-Modell.
iconPath	Pfad zum Icon, das zu dem zu definierenden Graphen gehört.
diagramExtension	Dateiendung der erzeugten Diagramme.
node	Leitet die Definition eines neuen Knotentyps ein.
edge	Leitet die Definition eines neuen Kantentyps ein.
container	Leitet die Definition eines neuen Containertyps ein.
attr *Typ* **as** *Name*	Attributdefinition.
Typ1 **extends** *Typ2*	Vererbung von Graphelementen.
@style	Weist dem folgenden Element (Knoten, Kante oder Container) den angegebenen Style zu.

Tabelle 3.1: Auflistung der wichtigsten Schlüsselwörter der MGL

Weiterhin können Knoten und Kanten Attribute (eingeleitet durch attr) enthalten, welche die Datentypen *EInt*, *EDouble*, *EString* oder Enumeration besitzen können. Im erzeugten CINCO-Produkt erscheinen für jedes Attribut – nach Anklicken des entsprechenden Elements – zusätzliche Textfelder, um den Attributen Werte zuzuordnen.

Jedem Graphelement kann durch eine Annotation ein Style zugewiesen werden, welcher dann das Aussehen des jeweiligen Elements bestimmt. Es ist möglich, in der Annotation mittels "${Attributname}" Attribute an den Style zu übergeben, sodass z.B. vom Nutzer eingegebene Bezeichnungen angezeigt werden können. Die Style-Annotation des graphModel enthält den Namen der .style-Datei, welche alle referenzierten MSL-Definitionen enthalten muss.

Neben den eigentlichen Graphelementen sind im Graphmodell noch einige Metadaten anzugeben, z.B. ein Packagename, Iconpfad und die Dateiextension für die Diagrammdateien.

Tabelle 3.1 zeigt eine Reihe von Schlüsselwörtern, die in MGL definiert sind und Code-Ausschnitt 3.1 zeigt am Beispiel eines Petrinetzgraphen, wie diese in der Praxis eingesetzt werden können.

Als Nächstes soll kurz die Meta Style Language betrachtet werden. Wie oben schon erwähnt, wird diese dazu verwendet, Graphelementen ein eigenes Aussehen zu geben. Dazu sind in der *Meta Style Language (MSL)* drei Dinge definierbar: appearance, nodeStyle und edgeStyle.

```
1  @style("model/PetriNet.style")
2  graphModel PetriNet {
3    package info.scce.cinco.product.petrinetpaper
4    nsURI "http://cinco.scce.info/product/petrinetpaper"
5    iconPath "icons/petri-icon.png"
6    diagramExtension "pn"
7
8    @style(place, "${token}")
9    node Place {
10     outgoingEdges (PlaceTransitionArc)
11     incomingEdges (TransitionPlaceArc)
12     attr EInt as token
13   }
14
15   @style(transition)
16   node Transition {
17     outgoingEdges (TransitionPlaceArc)
18     incomingEdges (PlaceTransitionArc)
19   }
20
21   @style(arc)
22   edge PlaceTransitionArc { }
23   @style(arc)
24   edge TransitionPlaceArc { }
25 }
26
```

Code-Ausschnitt 3.1: MGL: PetriNet Beispiel

Eine *Appearance* bekommt bestimmte Eigenschaften zugewiesen, die das Aussehen von Elementen beeinflussen. Diese Werte sind universell und passen zu jedem Elementtyp (Kanten, Knoten und Container). Dazu gehören zum Beispiel die Hintergrundfarbe und Liniendicke. Werden anschließend Styles definiert, dann können zuvor definierte Appearances verwendet werden, oder innerhalb der Styledefinition Appearances definiert werden. In Code-Ausschnitt 3.2 wird die Appearance *default* definiert, die bei allen Elementen, die diese verwenden, die Standardliniendicke und Standardhintergrundfarbe festlegt.

Tatsächlich in einer *Meta Graph Language (MGL)*-Datei verwendet werden können nur nodeStyle- und edgeStyle-Blöcke (jeweils für Knoten und Kanten). In beiden Styledefinitionen können Appearances definiert werden oder zuvor definierte Appearances referenziert werden. Weiterhin gibt es bestimmte Eigenschaften, die teils bei beiden, teils nur bei einem der beiden Blocktypen verwendet werden können. In einer nodeStyle-Definition können unter anderem die Position, die Ecken, ein Fülltext, die Größe usw. festgelegt werden.

Die edgeStyle-Definition funktioniert ähnlich, allerdings können hier zusätzlich *Dekoratoren* definiert werden und diese an der Kante z.B. als Pfeilspitzen positioniert werden. Die Positionierung von Dekoratoren funktioniert nicht absolut, sondern relativ zur Länge des Pfeiles. Die Position 0.5 positioniert den Dekorator genau in der Mitte der Kante.

```
1  appearance default {
2    lineWidth 2
3    background (144,207,238)
4  }
5
```

Code-Ausschnitt 3.2: MSL: Definition einer Appearance

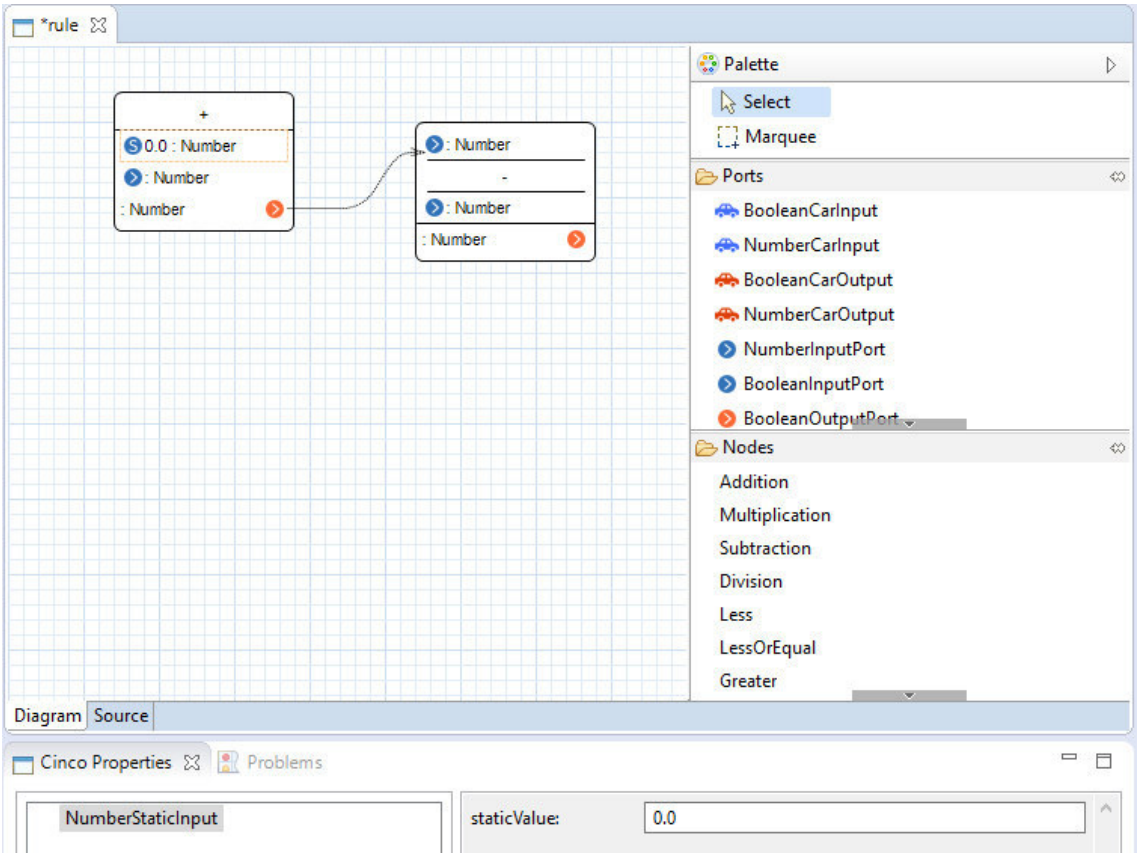


Abbildung 3.1: Editorteil der Benutzeroberfläche eines CINCO-Produkts

```
nodeStyle blueTextRectangle(1) {
  roundedRectangle {
    appearance default
    position (0,0)
    size (96,32)
    corner (8,8)
    text {
      position ( CENTER, MIDDLE )
      value "%s"
    }
  }
}

edgeStyle arc {
  appearance default
  decorator {
    location (1)
    ARROW
    appearance {
      foreground (128,128,128)
      lineWidth 2
    }
  }
}
```

Code-Ausschnitt 3.3: MSL: Definition von Styles

Wie in Code-Ausschnitt 3.3 zu sehen ist, kann hinter den Namen einer Styledefinition eine Zahl in Klammern geschrieben werden. Dies ist die Anzahl der Parameter, die an den Style übergeben werden müssen, wenn er in einem MGL-Modell referenziert wird. Durch (ggf. mehrmaliges) Schreiben von %s können die Parameter dann in Text eingebunden werden.

3.2.2 CINCO-Metaplugins

In diesem Unterkapitel werden wir die Funktionsweise von Hooks, Checks und Generatoren in CINCO betrachten. Diese werden durch sog. *Metaplugins* realisiert, die der Entwickler in Form von Annotationen ansteuert.

Beim Erstellen von graphischen Editoren reicht es oft nicht aus, dass der Nutzer selber Knoten und Kanten in sein Modell einfügen kann. Oftmals ist es wünschenswert, dass bestimmte Graphelemente automatisch erzeugt werden, z.B. ein Skelett oder Template beim Anlegen eines neuen Modells. Dazu bietet CINCO die Möglichkeit, Hook-Annotationen an Elemente in der MGL-Datei zu schreiben (u.a. @postCreate, @postResize und @preDelete).

Die Annotation bekommt als Parameter den (vollständig qualifizierten) Namen der Klasse, die Code für die Behandlung des jeweiligen Ereignisses enthält. Code-Ausschnitt 3.4 zeigt am Beispiel eines PostCreate-Hooks die Struktur einer solchen Klasse. Im Beispiel gibt es den Containertyp `Addition` und den Knotentyp `Port`. Beim Erstellen eines `Addition`-Containers werden darin automatisch zwei `Port`-Knoten erzeugt. Für alle Graphelemente werden im Generierungsschritt Klassen erzeugt, mit denen sich der Graph manipulieren lässt. So ist beispielsweise das Lesen und Setzen von Attributen möglich, das Erstellen und Löschen von Knoten in Containern, das Verschieben von Elementen, und vieles mehr.

```
1 public class CreateAddition extends CincoPostCreateHook<Addition> {
2     @Override
3     public void postCreate(Addition createdNode) {
4         createdNode.newPort(4, 24).setName("A");
5         createdNode.newPort(4, 42).setName("B");
6     }
7 }
```

Code-Ausschnitt 3.4: Beispiel für PostCreate-Hook

Nahezu identisch zu Hooks können auch @contextMenuAction- und @doubleClickAction-Annotationen hinzugefügt werden, die jeweils bestimmen, was geschieht, wenn man auf ein Element einen Rechtsklick bzw. Doppelklick ausführt. Die generischen Klassen, von denen dafür geerbt werden muss, heißen CincoCustomAction<T> bzw. CincoDoubleClickAction<T>.

Als Nächstes soll die Modellverifikation, welche durch das MCaM-Framework [25] realisiert wird, vorgestellt werden. Vom Entwickler können *Checks* definiert werden, die dann zu bestimmten Zeitpunkten zur Laufzeit auf dem Modell ausgeführt werden. Diese verhalten sich prinzipiell wie Unit-Tests in klassischen Programmiersprachen. Die Annotationen für Checks werden nicht an einzelne Elemente geschrieben, sondern an das graphModel. Zunächst ist @mcam("check") erforderlich, dann können mit @mcam_checkmodule("Klassenname") verschiedene Checks zum Modell hinzugefügt werden.

```
1 public class StartMarkerPresent extends WebStoryCheck {
2     @Override
3     public void check(WebStory story) {
4         if (story.getStartMarkers().isEmpty()) {
5             story.addError("A StartMarker is required.");
6         }
7     }
8 }
```

Code-Ausschnitt 3.5: Beispiel für Check

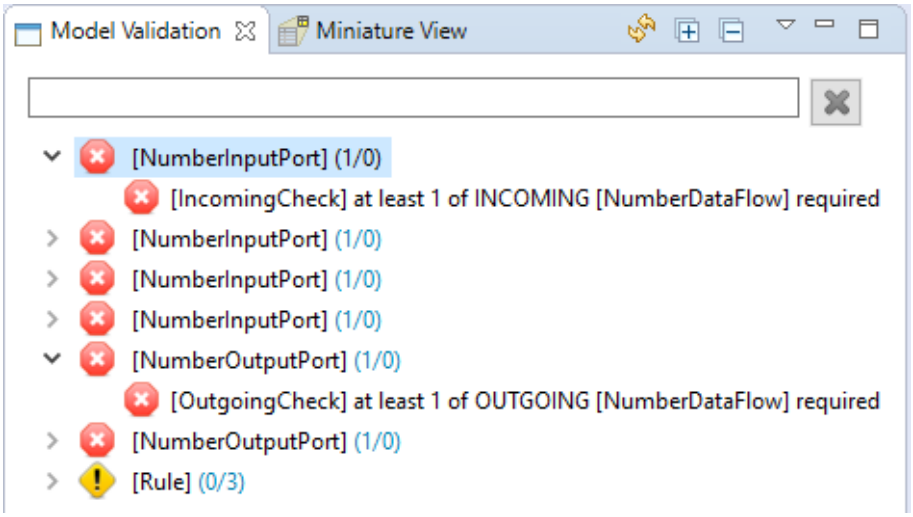


Abbildung 3.2: Benutzersicht auf Modellvalidierung

Code-Ausschnitt 3.5 zeigt beispielhaft einen einfachen Check. Die zu erbende Klasse trägt immer den Namen <Name des Graphmodells>Check und der an die check-Methode übergebene Parameter ist ein Objekt, über das auf alle Unterelemente des Graphen zugegriffen werden kann. Durch einen Aufruf von addError kann der Nutzer über erkannte Fehler informiert werden. Checks werden beim Speichern, beim Öffnen des Editors oder manuell (siehe Button oben rechts in Abbildung 3.2) ausgeführt.

Die Anbindung eines Generators erfolgt ebenfalls durch Annotieren des Graphmodells, und zwar mit @generatable("Klassenname"). Im generierten CINCO-Produkt existiert anschließend eine Schaltfläche, die den Generator anstößt. Der Generator ist eine Klasse, die von IGenerator<T> erbt (wobei T der Name des Graphmodells ist) und eine generate-Methode besitzt, die den zu überführenden Graphen übergeben bekommt. Daraufhin wird i.d.R. eine Ordnerstruktur für das generierte Projekt erzeugt, in die dann in der Zielsprache generierte Quelltextdateien gespeichert werden.

Hinweis: Die hier gezeigten Codebeispiele sind alle in Java verfasst, es ist jedoch stets auch möglich, Xtend (siehe Kapitel 4) zu verwenden, welches durch seine Fähigkeit, Template-Ausdrücke zu verarbeiten, insbesondere das Schreiben von Generatoren enorm vereinfacht.

3.3 Xtext

Xtext [26] ist ein Framework zur Entwicklung von textuellen DSLs. Es ist ein Teil des *Eclipse Modeling Frameworks (EMF)* [3] und ist dementsprechend sehr gut in die Entwicklungsumgebung Eclipse eingebunden. Generiert man eine in Xtext definierte Programmiersprache, wird automatisch ein dazugehöriger Parser, Linker, Typechecker und Compiler erzeugt.

Außerdem wird automatisch ein in Eclipse integrierter Editor erzeugt, welcher nützliche Zusatzfunktionen wie eine Autovervollständigungsfunktion besitzt.

In der Projektgruppe wird Xtext zur Implementierung einer Monitoring-Sprache für die entwickelten Fahrassistenzsysteme verwendet (siehe Kapitel 8).

3.3.1 Syntax und Funktionen

Zentral basiert die Syntax von Xtext auf der erweiterten Backus-Naur-Form (EBNF) und teilt große Teile der Syntax mit dieser. Eine in Xtext definierte Syntax besteht aus einer Menge an Nichtterminalsymbolen, auch Regeln genannt, welche Produktionsregeln einer formalen Grammatik darstellen. Dabei steht auf der linken Seite der Bezeichner der Regel, während rechts ein Ausdruck in der erweiterten Backus-Naur-Form (EBNF) (mit einigen Erweiterungen) steht. Die Syntax hierfür hat also die Form Bezeichner:<Erweiterte Expression>;. Zeichenketten, die von "" eingeschlossen werden, wie zum Beispiel "def", sind einfache Terminalsymbole. Es ist auch möglich komplexere Terminalsymbole zu definieren, so sind im Voraus schon verschiedene Terminalsymbole wie INT definiert. INT entspricht Zeichenketten der Form ('0'..'9')+ . Dabei gibt '0'..'9' an, dass an dieser Stelle eine Ziffer steht. + gibt an, dass der vorherige Teil beliebig oft wiederholt werden kann, aber mindestens einmal vorkommt. Analog dazu gibt es noch $*$ (wie +, aber erlaubt auch kein Vorkommen) und ? (der vorherige Ausdruck ist optional). $*$ entspricht somit +?.

Letztendlich erzeugt Xtext aus in der DSL definiertem Code einen Syntaxbaum, mit dem später gearbeitet werden kann. Der Syntaxbaum enthält Objekte, die aus den Nichtterminalsymbolen der Sprache instanziiert worden sind. Um auf die Eigenschaften eines Nichtterminals zugreifen zu können, ist es möglich, Bezeichner für einzelne Teilausdrücke zu vergeben. Definiert man zum Beispiel eine simple Regel wie Variable: "def" name=ID "=" value=INT; wird ein Objekt vom Typ Variable erzeugt, welches die Attribute name und value hat. Diese Attribute verweisen auf ID, ein vordefiniertes Terminal für Bezeichner, und auf eine Zahl. Sie werden auch *Features* genannt. Besitzt eine Regel keine Featurezuweisung, wird auch kein Objekt erzeugt, da dieses keine abrufbaren Attribute hätte. Zusätzlich zu der einfachen Zuweisung gibt es auch die Möglichkeit mit += sogenannte *multi-valued features* zu definieren. So könnte man zum Beispiel mit Definitions: defs+=Variable; Objekte vom Typ Definitions erzeugen, die eine Liste von Objekten vom Typ Variable haben. Um nun an anderer Stelle auf vorher definierte Variablen verweisen zu können, kann eine Crossreference verwendet werden. Die Regel Addition: left=[Variable] "+" right=[Variable]; beschreibt beispielsweise eine Addition, bei der zwei Variablen referenziert werden. Bei Referenzen wird standardmäßig das name-Feature verwendet, welches vom Typ ID oder STRING sein muss.

```
Addition returns Expression:
  Multiplication ({Addition.left=current} '+' right=Multiplication)*;

Multiplication returns Expression:
  Primary ({Multiplication.left=current} '*' right=Primary)*;

Primary returns Expression:
  NumberLiteral |
  '(' Addition ')';

NumberLiteral:
  value=INT;
```

Code-Ausschnitt 3.6: Simple arithmetische Ausdrücke in Xtext

```
Prefix returns Prefix:
    {ValPrefix} "val" |
    {VarPrefix} "var";
```

Code-Ausschnitt 3.7: Explizite Typisierung

Mithilfe des Schlüsselworts returns <Typname> kann der Obertyp der für eine Regel erzeugten Klasse festgelegt werden. Code-Ausschnitt 3.6 zeigt eine Beispielsprache für geklammerte arithmetische Ausdrücke mit den Operatoren Addition und Multiplikation. Die drei Hauptregeln werden dabei alle von der Oberklasse Expression abgeleitet. Eine weitere Besonderheit in dem Beispiel sind die Ausdrücke in geschweiften Klammern, welche eine weitergehende Manipulation des Syntaxbaums erlauben. Bei der Addition wird das Feature left auf current gesetzt – current ist ein Spezialbezeichner für das aktuelle Syntaxbaum-Objekt, das innerhalb der Regel bereits erzeugt wurde. Es wird immer erst die Multiplication-Regel aufgerufen, welche im simpelsten Fall einfach über Primary ein NumberLiteral zurückgibt. In dem Falle wäre current dann das NumberLiteral, welches schließlich an Addition.left zugewiesen wird.

Auch der Typ von Regeln selbst kann bestimmt werden – dies dient entweder dazu, die erzeugte Klasse anders zu benennen, als die Regel, oder aber um die Erzeugung eines Objekts zu erzwingen (falls es keine Attribute hat). Code-Ausschnitt 3.7 zeigt dies anhand des Beispiels von möglichen Arten von Variablen- oder Felddeklarationen in Xtend (siehe Unterabschnitt 4.2.2).

3.3.2 Generatoren, Validatoren und weitere Zusatzfunktionen

Es gibt viele Möglichkeiten, mit der erzeugten DSL zu arbeiten und erweiterte Funktionen zu entwickeln. Zum Beispiel ist es möglich, über sog. *Scope Provider* und *Proposal Provider* eine Xtext-DSL mit beliebigen anderen EMF-basierten Metamodellen zu verknüpfen. Der Scope Provider kann für einen beliebigen Kontext der DSL festlegen, welche Eingaben als gültig akzeptiert werden (im häufigsten Fall sind dies Bezeichner aus anderen Modellen). Hingegen ist der Proposal Provider lediglich dafür zuständig, dem Eclipse-Editor Vorschläge für die Autovervollständigung zu liefern.

Neben jenen Providern kann auch noch Validierung eingebaut werden, die dazu dient, über einfache Syntaxfehler hinausgehende Fehler kenntlich zu machen.

Weiterhin erlaubt es Xtext, einen Codegenerator anzubinden. Dieser ermöglicht es, aufbauend auf dem Modell der DSL Code für eine beliebige andere Sprache (z.B. C++ oder Java) zu erzeugen.

Für jede der genannten Funktionen werden beim Erzeugen des Xtext-Projekts leere Klassenrahmen erzeugt, wo eigene Funktionalität ergänzt werden kann. Informationen über die technischen Details dieser Funktionen lassen sich in der offiziellen Xtext-Dokumentation finden[1].

[1] `https://www.eclipse.org/Xtext/documentation/index.html`

Kapitel 4

Xtend

Dieses Kapitel führt die im Rahmen der entwickelten DSL und des Generators verwendete Programmiersprache Xtend ein und klärt dabei den grundlegenden Aufbau und verschiedene Funktionalitäten. Dabei werden hier jedoch nicht alle Strukturelemente und Funktionalitäten aufgeführt. Für weitere Informationen ist in jedem Fall ein Blick in die Dokumentation ratsam.

4.1 Einleitung

Bei Xtend handelt es sich um eine Programmiersprache, die in ihrer Struktur Java sehr ähnelt, jedoch bei weitem flexibler ist und damit insbesondere kompakteren Code ermöglicht und in Teilen auch mehr Funktionalität bietet. Welche Zugewinne hier vorliegen, wird im weiteren Verlauf näher beleuchtet.

Eine weitere wichtige Eigenschaft der Sprache ist allerdings auch die Tatsache, dass sie nicht zu herkömmlichem Bytecode übersetzt wird, sondern zu Java-Quelltext kompiliert. Aufgrund dessen ist eine vollständige Interoperabilität zwischen den beiden Sprachen gewährleistet und es ist mühelos möglich, in einem Projekt beide zu verwenden und wechselseitige Referenzen aufzubauen. Einen ersten Eindruck hiervon erhält man im Rahmen des Hello-World-Beispiels im nächsten Unterabschnitt 4.1.1.

Xtend ist statisch typisiert, unterstützt das Java-Typsystem – also angefangen von den primitiven Typen `int` und `boolean` über Java-Klassen, Interfaces, generische Typen, etc. – vollständig, weshalb es nicht zu problematischen Typ-Zuordnungen kommen kann. Die ausnahmslose Kompatibilität ermöglicht einen reibungslosen Übergang von einer Sprache in die andere, ohne zuvor die Weltsicht ändern zu müssen [7, 8].

4.1.1 Hello World

Als Ausgangspunkt für die Arbeit mit einer zunächst noch fremden Programmiersprache bietet sich häufig das Erstellen einer kleinen Hello-World-Anwendung an. Der nachfolgende Code-Ausschnitt 4.1 zeigt, wie diese im Falle von Xtend aussehen kann.

```
1 class HelloWorld {
2   def static void main(String[] args) {
3     println("Hello World")
4   }
5 }
```

Code-Ausschnitt 4.1: Hello World in Xtend

Während einem Entwickler mit grundlegenden Kenntnissen in Java die meisten Aspekte bereits vertraut vorkommen dürften, werden auch in einem so kleinen Beispiel schon erste Unterschiede deutlich. Da der Xtend-Code, wie zuvor erwähnt, nicht in Bytecode, sondern Java-Quelltext überführt wird, kann man bereits im direkten Vergleich einige dieser Unterschiede und ebenso der Gemeinsamkeiten betrachten.

```
1 // Generated Java Source Code
2 import org.eclipse.xtext.xbase.lib.InputOutput;
3
4 public class HelloWorld {
5   public static void main(final String[] args) {
6     InputOutput.<String>println("Hello World");
7   }
8 }
```

Code-Ausschnitt 4.2: Java-Generat

Wie in Java ist es für eine ausführbare Anwendung notwendig, eine Klasse mit main-Methode zu deklarieren. In einem Hello-World-Beispiel ist die Komplexität natürlich nicht besonders groß, weshalb der spätere Gewinn durch Xtend zunächst noch nicht sehr deutlich wird, sondern nur sehr gering wirkt. Was man bereits feststellen kann, sind Einsparungen bezüglich der Sichtbarkeitsangabe `public` oder das Entfallen von Semikola.

Auffällig ist darüber hinaus das neue Schlüsselwort `def` zu Beginn der Methode, welches im Kapitel Unterabschnitt 4.2.3 erklärt werden wird. Außerdem wird zusätzlich eine Xtend-Standardbibliothek importiert und anstelle des möglicherweise zu erwartenden `System.out` verwendet, um zusätzliche Eigenschaften der Xtend-Sprache zu gewährleisten [6].

4.2 Struktur

Grund dafür, dass eine Klasse oder Methode nicht explizit als `public` gekennzeichnet werden muss, ist die Tatsache, dass dies in Xtend der angenommene Default ist, da dieser häufiger Verwendung findet. Wer tatsächlich aber Javas standardmäßige "package private"-Sichtbarkeit wünscht, kann dies wiederum in Xtend durch das Schlüsselwort `package` erzielen [4].

4.2.1 Klassen

In Xtend ist es im Gegensatz zu Java möglich, mehrere Klassen auf oberster Ebene innerhalb einer einzigen Datei zu deklarieren, generiert wird für jede Klasse aber notwendigerweise eine eigene. Als Resultat dessen sind in Bezug auf die Dateinamenswahl von Xtend-Code auch keine starken Einschränkungen gegeben [4].

Konstruktor

Die Konstruktoren in einer Xtend-Klasse unterscheiden sich bereits sehr deutlich von ihrem Java-Pendant. Für sie ist es nicht nötig, den bereits spezifizierten Klassennamen erneut zu wiederholen. Anstelle dessen findet das Schlüsselwort `new` gemäß der Veranschaulichung in Code-Ausschnitt 4.3 Verwendung. Wie ebenfalls in dem Beispiel zu sehen, kann mittels `this` samt Argumenten auch an einen anderen Konstruktor derselben Klasse delegiert werden [4].

```
1 class MyClass extends AnotherClass {
2   new(String s) {
3     super(s)
4   }
5
6   new() {
7     this("default")
8   }
9 }
```

Code-Ausschnitt 4.3: Xtend-Konstruktoren

4.2.2 Felder

Felder sind in Xtend defaultmäßig `private`. Ihre Deklaration wird durch die neuen Schlüsselwörter `val` (für Value) – was synonym zu `final` verwendet werden kann – und `var` (für Variable) für nicht-finale Felder eingeleitet. Die Verwendung von `var` ist dabei zusätzlich optional, kann also weggelassen werden. Darauf folgt die Angabe des jeweiligen Typs und danach der Bezeichner [4, 5].

```
class MyClass {
  int count = 1
  static boolean debug = false
  var name = 'Foo'          // type String is inferred
  val UNIVERSAL_ANSWER = 42 /* final field with inferred
  type int */
  //...
}
```

Code-Ausschnitt 4.4: Verschiedene Felder

Typinferenz

Ähnlich der vergleichbaren Problematik, welche bereits eben bei den Konstruktoren angesprochen wurde, so ist man im Java-Umfeld auch zum vielfachen Wiederholen von Typsignaturen gezwungen. Dies ist zum einen zwar auch dem statischen Typsystem geschuldet, doch kann innerhalb von Xtend zu großen Teilen darauf verzichtet werden, auch wenn hier ebenfalls statische Typen verwendet werden.

```
final LinkedList<String> list = new LinkedList<String>();
```

Code-Ausschnitt 4.5: Erstellen einer Liste in Java

```
val list = new LinkedList<String>
```

Code-Ausschnitt 4.6: Erstellen einer Liste in Xtend

Die Schreibweise in Java ist sehr lang und enthält redundante Informationen, da der Typ aus dem Kontext ableitbar wäre. Gerade unter dieser Voraussetzung kann eine Typangabe im Rahmen von Xtend ausgespart werden. Möchte man, dass nur eine allgemeinere Typinformation genutzt wird – hier im Code-Ausschnitt 4.6 etwa `List<String>` – so ist die explizite Deklaration nötig, da nur genau der Typ auf der rechten Seite inferiert werden kann [5, 8].

4.2.3 Methoden

Die Signatur von Xtend-Methoden wird mit einer Ausnahme eins zu eins bei der Übersetzung zu Java übernommen, was erneut der einfachen Interoperabilität sehr zuträglich ist. Von der Notation her gibt es hier aber erneut Unterschiede festzustellen. Methoden werden mit entweder **`def`** oder **`override`** eingeleitet, abhängig davon, ob es sich um eine erstmalige

Definition handelt oder eine bestehende Funktion einer Oberklasse beziehungsweise eines Interfaces überschrieben wird.

Die Angabe des Rückgabetyps kann des Weiteren üblicherweise entfallen, da dieser aus dem Methodenrumpf ermittelt werden kann. Ausnahmen bilden rekursive Methoden, da der Typ hier bereits innerhalb des Rumpfes statisch bekannt sein muss, sowie abstrakte Methoden, da hier der Rumpf noch überhaupt nicht vorliegt. Ebenfalls für Methoden mit dem Rückgabetyp `void`, wie die, welche bereits exemplarisch im einleitenden Hello-World-Beispiel in Code-Ausschnitt 4.1 zu sehen war, ist die Typangabe empfehlenswert, wenn auch nicht verpflichtend. Der Grund hierfür wird in Abschnitt 4.3 deutlich [4].

4.2.4 Automatische Konvertierung

Aus Java ist das Konzept der Wrapper-Klassen bekannt, welches ermöglicht, die primitiven Typen wahlweise auch als Klasse/Objekt zu betrachten und mittels (Un-)Boxing zwischen den beiden Repräsentationen zu wechseln. In Xtend wird dieses Konzept erweitert und lässt ebenfalls eine automatische Konvertierung von Arrays und Listen ineinander zu, mit Hinblick auf die Richtung hin zu Arrays gilt dies sogar für alle Implementierungen von `Iterable`. Code-Ausschnitt 4.7 zeigt, dass eine einfache Zuweisung genügt. [5, 8]

```
1 def toList(String[] array) {
2   val List<String> asList = array
3   return asList
4 }
```

Code-Ausschnitt 4.7: Automatische Konvertierung von Array zu Liste

4.2.5 Extension Methods

Extension Methods sind der Kern und der Namensgeber der Sprache Xtend. Sie erhöhen die Flexibilität, wie ein Methodenaufruf aussehen kann, und ermöglichen es auf einfache Weise, einer Klasse zusätzliche Funktionalität zu geben, ohne diese direkt zu manipulieren.

Ermöglicht wird dies mittels Zulassens einer abweichenden Syntax für einen Methodenaufruf, bei der das erste Argument einer Methode nach vorne gezogen wird. Dies ist im Folgenden beispielhaft dargestellt.

Es gibt verschiedene Möglichkeiten, derartige Schreibweisen für Methoden zugänglich zu machen. Zunächst bieten alle nicht-statischen lokalen Methoden diese Zugriffsweise bereits automatisch an. Code wie im Beispiel in Code-Ausschnitt 4.9 ist damit von vornherein möglich. Für andere Fälle muss ein geringer zusätzlicher Aufwand betrieben werden.

```
"hello".toFirstUpper
// calls StringExtensions.toFirstUpper(String)

listOfStrings.map[ toUpperCase ]
/* calls ListExtensions.<T, R>map(List<T> list,
   Function<? super T, ? extends R> mapFunction) */
```

Code-Ausschnitt 4.8: Methodenaufrufe mit Extension Methods

```
class MyClass {
  def doSomething(Object obj) {
    // do something with obj
  }
  def extensionCall(Object obj) {
    obj.doSomething() // calls this.doSomething(obj)
  }
}
```

Code-Ausschnitt 4.9: Extension Methods: nicht-statische lokale Methoden

Static Imports

Um eine Methode als Extension nutzen zu können, muss sie entsprechend importiert werden. Hierfür wird eine um das Schlüsselwort **extension** ergänzte Form des Static Imports genutzt, wie dies Code-Ausschnitt 4.10 veranschaulicht.

Die in Code-Ausschnitt 4.8 genutzten Erweiterungs-Klassen werden als fester Bestandteil von Xtend implizit automatisch entsprechend bereitgestellt [4].

```
import static extension java.util.Collections.singletonList
// allows for later use of singletonList methods like this:
new MyClass().singletonList()
// calls Collections.singletonList(new MyClass())
```

Code-Ausschnitt 4.10: Extension Methods: Static Import

4.3 Expressions

In Xtend ist alles eine Expression, alles hat einen Rückgabewert. Dies ermöglicht neben vielem anderen beispielsweise die Verwendung eines Try-Catch-Ausdrucks auf der rechten Seite einer Zuweisung, wie nachfolgend dargestellt.

```
val data = try {
  fileContentsToString('data.txt')
} catch (IOException e) {
  'dummy data'
}
```

Code-Ausschnitt 4.11: Try-Catch-Zuweisung

Läuft die Methode `fileContentsToString` ohne Probleme durch, so wird `data` das Methodenergebnis als Wert zugewiesen. In dem Fall, dass anstelle dessen jedoch eine `IOException` geworfen wird, so wird dies abgefangen und `data` erhält das Resultat des Catch-Blockes, in diesem Fall `'dummy data'`.

Allgemeiner betrachtet ist festzuhalten, der Rückgabewert eines Code-Blocks ist der Wert des letzten Ausdrucks in diesem Block. Aufgrund dessen ist in Methoden auch nicht länger ein **`return`** erforderlich, sofern eine Methode nicht frühzeitig verlassen werden soll. Zur deutlichen Veranschaulichung kann ein explizites **`return`** dienen, wie etwa in Code-Ausschnitt 4.7. Die zusätzliche Zeile Code ist dort außerdem nötig, da als letzte Anweisung innerhalb eines Blockes keine Variablendeklaration stehen darf.

Schleifen haben grundsätzlich den Typ **`void`** [5].

4.3.1 Strings

Strings sind in Xtend erneut flexibler bezüglich ihrer Notation. Während Java ausschließlich Anführungszeichen zu deren Kenntlichmachung verwendet, können in Xtend zusätzlich auch Hochkommata genutzt werden. Diese werden aufgrund der geringeren Störwirkung beim Lesen gegenüber der Alternative auch empfohlen, wobei darauf hingewiesen wird, dass die Häufigkeit möglicherweise zu escapender Zeichen bei der Wahl insbesondere berücksichtigt werden sollte.

In dem Fall, dass die Notwendigkeit eines Zeichen-Escapes im String weiterhin bestehen bleibt, kann dieses wie üblich über einen Backslash erreicht werden. Eine weitere Eigenschaft von Strings ist, dass diese sich hier auch über mehrere Zeilen erstrecken dürfen [5].

```
'Hello World !'
"Hello World !"
'Hello "World" !'
"Hello \"World\" !"
```

Code-Ausschnitt 4.12: verschiedene zulässige String-Notationen

4.3.2 Setter & Getter

Werden Feldern Werte zugewiesen, so gibt der Ausdruck dieser Zuweisung ebenfalls den entsprechenden Wert zurück. Felder sind nicht aus jedem Kontext heraus direkt zugänglich, weshalb in jenen Fällen auf eine Setter-Methode zurückgegriffen werden muss. Xtend vereinheitlicht die Schreibweise von regulären Zuweisungen und Settern so, dass bei notierter Zuweisung und mangelndem Direktzugriff eine Abbildung auf die Setter-Methode stattfindet. Der Rückgabewert eines Setters kann je nach Methodendefinition abweichen.

```
1 myObj.myProperty = 'foo' // calls myObj.setMyProperty("foo")
2 myObj.myProperty /* calls myObj.getMyProperty()
3                     in case myObj.myProperty is not visible */
```

Code-Ausschnitt 4.13: vermeintlicher Feld-Zugriff

Gleiches gilt für das Referenzieren von Feldern beim Auslesen des Wertes. Ist ein Feld aus dem Kontext nicht zugänglich, wird, wenn verfügbar, auf eine dem Bezeichner zugehörige Getter-Methode ausgewichen. Voraussetzung ist in allen Fällen, dass wenn das Attribut – im Code-Ausschnitt 4.13 wäre dies `myObj.myProperty` – nicht sichtbar ist, auch keine parameterlose Methode `myObj.myProperty()` vorhanden sein darf, da diese sonst Vorzug hätte. Klammern sind bei parameterlosen Methoden nicht notwendigerweise zu setzen [5].

4.3.3 Nicht Null

Arbeitet man in einem Programm mit Objektreferenzen, die auch hin und wieder `null` enthalten können, so müssen dahingehend immer wieder Überprüfungen stattfinden, um zu verhindern, dass eine `NullPointerException` geworfen wird. Der resultierende Code kann dabei sehr schnell schlecht lesbar werden, weshalb in Xtend eine kompaktere aber äquivalente Schreibweise zur Absicherung gegen derartige Aufrufe zur Verfügung steht.

```
1 if (myRef != null) myRef.doStuff()
2 // can be simplified to
3 myRef?.doStuff
```

Code-Ausschnitt 4.14: Gegen Null abgesicherter Methodenaufruf

Wie der Code-Ausschnitt 4.14 zeigt, ist es nur nötig, ein Fragezeichen an die möglicherweise problematische Objektreferenz anzuhängen beziehungsweise dem Methodenaufruf voranzustellen. Ist der alternative Fall, dass tatsächlich `null` vorliegt, gesondert zu behandeln, hilft die kurze Schreibweise zwar nicht mehr, doch für sonstige Situationen kann Code so sehr viel übersichtliche gestaltet werden [5].

4.3.4 Lambda-Ausdrücke

In Xtend wurden Lambda-Ausdrücke bereits eingeführt, bevor Java seine eigene Umsetzung nachreichte. Je nach Compiler-Wahl werden Xtend-Lambdas daher zu Java-Lambda-Ausdrücken aus Java 8 oder sonst zu Anonymen Klassen überführt.

Ein Lambda-Ausdruck wird in Xtend von eckigen Klammern umschlossen und besteht aus einer Aufzählung der Parameter gefolgt von einem `|` und dem eigentlichen Ausdruck.

```
1 Collections.sort(someStrings) [ a, b | a.length - b.length ]
```

Code-Ausschnitt 4.15: Methode mit Lambda als Argument

Die Typangabe für Parameter kann hier erneut ausgelassen werden, da diese aus dem Kontext inferiert werden kann. Eine Auflistung der Parameter ist ebenfalls optional. Wird diese eingespart, so werden die Parameter mit `$1`, `$2`, `...`, `$n` je nach Anzahl `n` bezeichnet.

Wie Code-Ausschnitt 4.15 verdeutlicht, kann ein Lambda-Ausdruck, der letztes Argument einer Methode ist, außerhalb der Klammern des Methodenaufrufs notiert werden. Dies ist insbesondere von Bedeutung, wenn es sich um das einzige Argument handelt, da dann die runden Klammern trotz vorhandenem Argument vollständig entfallen können [5, 8].

4.3.5 Templates

Template-Methoden stellen eine weitere sehr große Funktionalität von Xtend dar. Sie werden von dreifachen Hochkommata umschlossen und dienen zum Erzeugen größerer Texteinheiten. Der reine Text wird für das Einbinden von Programmlogik durch Guillemets verlassen, welche innerhalb der Eclipse-Umgebung leicht durch das Drücken der Strg-Taste gefolgt von der Eingabe von Kleiner- beziehungsweise Größerzeichen eingefügt werden können.

```
1 def someHTML(String content) '''
2 <html>
3   <body>
4     «content»
5   </body>
6 </html>
7 '''
```

Code-Ausschnitt 4.16: Eine einfache Template-Methode

Der Template-Anteil muss sich dabei nicht über die ganze Methode erstrecken, sondern kann auch innerhalb einer Methode wo angebracht für einen kleineren Ausschnitt erfolgen.

Innerhalb eines Templates sind zeilenweise Kommentare durch Eingabe von «««« möglich. Für Bedingungen und Schleifen innerhalb des Templates stehen eigene Ausdrücke zur Verfügung, welche in den nachfolgenden Beispielen dargestellt werden.

```
'''
«IF condition»
content dependent on condition
«ENDIF»
'''
```

Code-Ausschnitt 4.17: Bedingung innerhalb eines Templates

```
'''
«FOR p : paragraphs BEFORE '<p>' SEPARATOR '</p><p>' AFTER '</p>'»
  «p.text»
«ENDFOR»
'''
```

Code-Ausschnitt 4.18: Schleife innerhalb eines Templates

Die zusätzlichen Schlüsselwörter der Schleife fügen abhängig davon, ob es keinen, einen oder mehrere Schleifendurchläufe gibt, Text zu Beginn und am Ende aller Iterationen beziehungsweise zwischen den einzelnen Iterationen ein [5].

Kapitel 5

Automotive Data and Time-Triggered Framework

Zunächst wird Grundlegendes zu Automotive Data and Time-Triggered Framework erläutert. Anschließend wird die grundlegende Hardwareplattform mit ihren zusammenwirkenden Subkomponenten dargestellt.

5.1 Einleitung

Das Automotive Data and Time-Triggered Framework ist ein Framework zur Entwicklung von Assistenzsystemen in Kraftfahrzeugen. Moderne Kraftfahrzeuge bieten eine Vielzahl an Assistenzsystemen. Um die Entwicklung und das Testen dieser zu vereinfachen, wurde das *Automotive Data and Time-Triggered Framework (ADTF)* entwickelt. Es bietet Tools für die Entwicklung, Visualisierung, Validierung und das Testen von Automobilsoftware.

Im „Configuration Editor“ lassen sich Datenflussmodelle modellieren. Die einzelnen Knoten in dem Modell repräsentieren ADTF-Plugins, die Basisfunktionalitäten zur Verfügung stellen. Durch baukastenartiges Verbinden von unterschiedlichen Plugin-Knoten lässt sich eine Modelllogik entwerfen. Die Verbindung der einzelnen Plugin-Knoten erfolgt datenflussbasiert. Dazu stellen die Plugins Output- bzw. Input-Ports bereit. Im ADTF-Kontext wird eine Menge von verbundenen Filtern als Konfiguration bezeichnet. Die innerhalb der Projektgruppe modellierten Basis-Konfigurationen werden detailliert in Abschnitt 12.2 vorgestellt. Eine genauere Erläuterung der Funktionsweise der Filter findet sich im anschließenden Abschnitt 12.1 und in Abschnitt B.

5.2 Hardwareplattform

Abbildung 5.1 zeigt den das Modellauto betreffenden Teil der Projektarchitektur.

Der mit „META“ beschriebene Block stellt die Meta-Ebene in Form von modellierten Fahrassistenzsystemen dar. Im Rahmen der Implementierung erfolgt dies mit dem in Abschnitt 3.2 vorgestellten CINCO-Produkt. Aus den Meta-Modellen wird durch Full-Code-Generation eine konkrete Instanz eines Fahrassistenzsystems in Form eines ADTF-Plugins generiert. ADTF-Plugins werden in einer auf dem Modellauto installierten ADTF-Instanz ausgeführt. Der sogenannte Arduino-Controller stellt dabei ein Spezial-Plugin dar, mit dem die Sensorik und Aktuatorik des realen Modellautos gesteuert werden können. Diese werden in Unterabschnitt 5.2.2 näher vorgestellt. Die Steuerung des Modellautos erfolgt über eine dezidierte Funkfernsteuerung, mithilfe eines Xbox-Controllers, tiefergehend erläutert in Unterabschnitt 12.1.4, kann aber auch teilweise oder vollständig von einem *Advanced Driver Assistance Systems (ADAS)* übernommen werden. Die manuellen Steuerungsmethoden sind zusammengefasst in dem mit „Inputs“ beschriebenen Block. Im Rahmen der Projektgruppe erfolgt die Simulation mit *Virtual Test Drive* (VTD) . Die Kommunikation zwischen Simulationsumgebung und ADTF-Instanz erfolgt über spezifische ADTF-Plugins. Eine nähere Beschreibung befindet sich in Kapitel 6. Aus der ADTF-Umgebung heraus kann mit einem Plugin zudem eine Verbindung zu einem MQTT-Broker hergestellt werden, siehe auch Unterabschnitt 12.1.6.

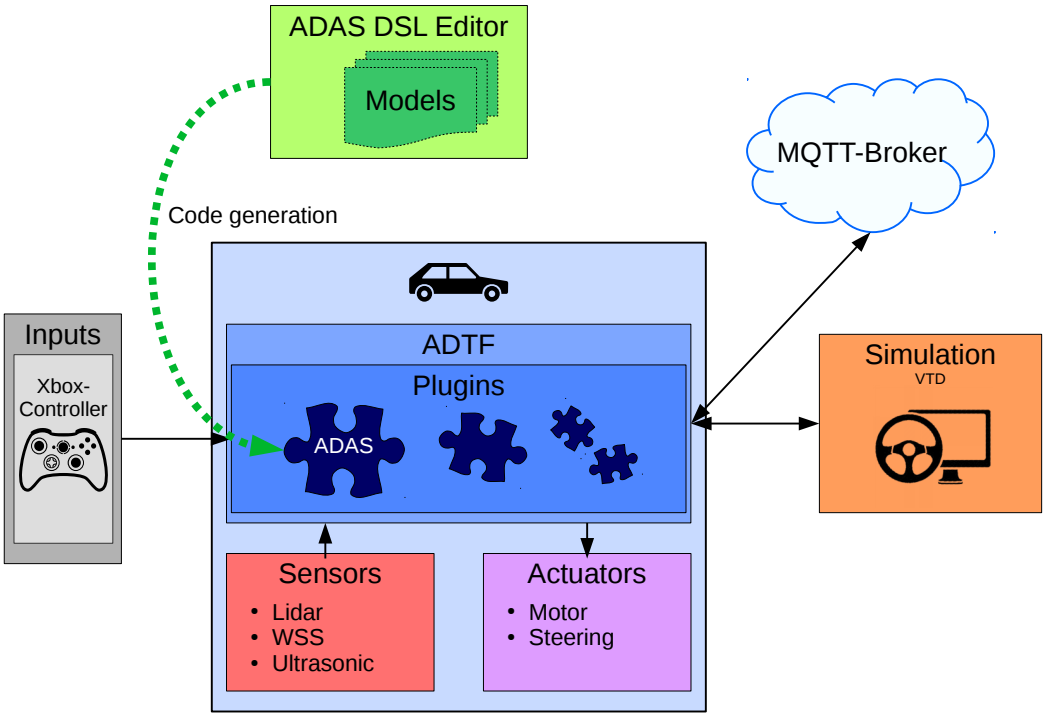


Abbildung 5.1: Architekturbild

5.2.1 Hard- und Software

Auf dem Modellauto ist ein vollwertiger PC montiert, welcher insbesondere zum Ausführen der installierten ADTF-Umgebung dient. Dank vorinstallierten SDKs ist auch das Entwickeln von ADTF-Plugins auf diesem Rechner möglich. Die Eckdaten des Systems sind:

- Intel® Core™ i3-6100T CPU
- 8 GB DDR4 PC-2133 RAM
- NVIDIA GeForce GTX 1050 Ti GPU
- XUbuntu Linux 16.04.2 LTS

Die Software sowie Sensorik und Aktuatorik des Fahrzeugs waren bereits weitestgehend vom Hersteller konfiguriert. Während der Projektlautzeit wurden ADTF-Plugins nahezu ausschließlich auf diesem System entwickelt, wodurch das Fahrzeug eine begrenzte Ressource darstellte.

5.2.2 Sensorik und Aktuatorik

Verbaute Sensorik, am Fahrzeug angebracht wie in Abbildung 5.2 dargestellt:

- *Ultraschallsensoren (USS)* hinten links, mittig und rechts sowie jeweils seitlich am Fahrzeug
- Inertiale Messeinheit (englisch *inertial measurement unit*, IMU)
- Lidar (Abkürzung für englisch *light detection and ranging*) an Fahrzeugfront
- Zusätzliche Raddrehzahlsensoren an jedem Rad

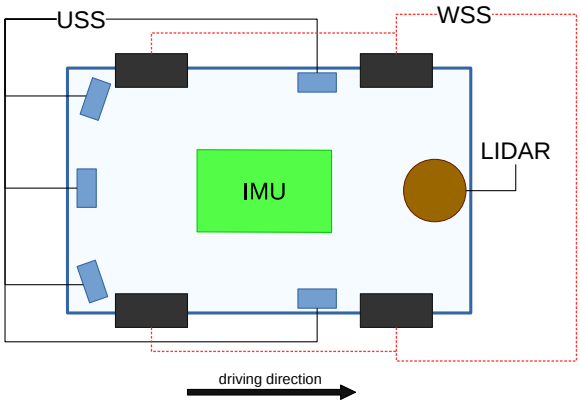


Abbildung 5.2: Schematische Darstellung der verbauten Sensorik

Mithilfe von Ultraschallsensoren lassen sich Abstände zu Hindernissen messen. Die verbauten Sensoren decken einen Erfassungsbereich zwischen 2 cm und 4 m ab und haben

eine Auflösung von 3 mm [27]. Im Gegensatz zu den Ultraschallsensoren liefert das Lidar nicht jeweils einen einzelnen Abstandswert, sondern kann prinzipiell mehrere Hindernisse in einem 360°-Bereich um sich selbst erfassen. Dabei ist die zeitliche Auflösung der ermittelten Hindernis-Daten abhängig von der Baudrate, also der Drehgeschwindigkeit des Lasers. Je höher die Drehgeschwindigkeit, desto höher ist auch die zeitliche Auflösung. Eine Visualisierung der vom Lidar erkannten Hindernisse findet sich in Unterabschnitt 12.1.1. Die inertiale Messeinheit misst auftretende Beschleunigungen. Auslesbar sind die von der verbauten Sensorik gemessenen Werte zerlegt in drei Dimensionen eines kartesischen Koordinatensystems. Die Raddrehzahlsensoren ermöglichen das Ermitteln der Fahrzeuggeschwindigkeit.

Zusätzlich angebracht sind eine Frontkamera, eine Front-Tiefenkamera sowie eine Rückfahrkamera.

Zur Aktuatorik zählen die Lenkung sowie der Allradantrieb der Räder. Die Lenkung wird von einem Servomotor gesteuert. Der Hauptantrieb erfolgt über einen bürstenlosen Elektromotor.

5.2.3 Autonomer Fahrmodus und RC-Modus

Am Fahrzeug befindet sich ein Schalter, der das Umschalten zwischen dem autonomen Fahrmodus und dem RC-Modus ermöglicht. Im RC-Modus lässt sich das Fahrzeug ausschließlich mit der RC-Fernsteuerung steuern und es sind keine Steuerungseingriffe via Software möglich. Im autonomen Fahrmodus wird das Fahrzeug ausschließlich über die ADTF-Umgebung per Software gesteuert. Eine gemischte Steuerung, die Steuerungsbefehle vom Benutzer und autonome Softwaresteuerung ermöglicht, lässt sich mithilfe des Xbox-Receiver-Filters realisieren. Denkbar ist beispielsweise eine manuelle Lenksteuerung kombiniert mit autonomer Gas- und Bremssteuerung.

Kapitel 6

Virtual Test Drive

In diesem Abschnitt wird das Tool VTD der Firma *Vires* [11], sowie das integrierte Tool *ScenarioEditor* vorgestellt. Diesbezüglich werden der Verwendungszweck und die Grundlagen des Programms erläutert.

6.1 Einleitung

Bei VTD handelt es sich um eine Simulationsumgebung der Firma Vires [11]. Sie läuft auf einem x86-basierten Linux Betriebssystem und ist so aufgebaut, dass die einzelnen Komponenten eigenständige Prozesse darstellen, wobei diese in Hintergrund- und Vordergrundprozesse unterteilt sind. Abbildung 6.1 stellt die gesamte Architektur von VTD dar. Das Kernstück bildet dabei die Simulationssteuerung (*simulation control*), welche die verschiedenen Prozesse synchronisiert und die Daten an die entsprechenden Komponenten weiterleitet.

Das *Simulation Control Protocol (SCP)* ist eine Schnittstelle und dafür zuständig, Kommandos zur Steuerung und Konfiguration zu empfangen, die dann über die Simulationssteuerung entsprechend weitergeleitet werden. Durch SCP können beispielsweise Umgebungsparameter, wie das Wetter oder die Uhrzeit geändert werden, Ampelphasen gesteuert, Zustandsdaten aus der laufenden Simulation abgefragt oder Sensormodelle konfiguriert werden, sowie sämtliche Aktionen und Ereignisse, die an die SCP Schnittstelle angebunden sind, übermittelt werden.

Das *Generic Simulation Interface (GSI)* überträgt Datenströme und Bilder aus der Visualisierung und enthält die Zustandsdaten der Simulation. So enthält es beispielsweise Positions- und Zustandsdaten über alle Fahrzeuge, Fußgänger, Fahrbahnen und Lichtquellen/Ampeln. Über diese beiden Schnittstellen werden auch externe Komponenten, wie die Sensormodelle oder die externe Fahrdynamik übertragen. Die dynamischen Aspekte des

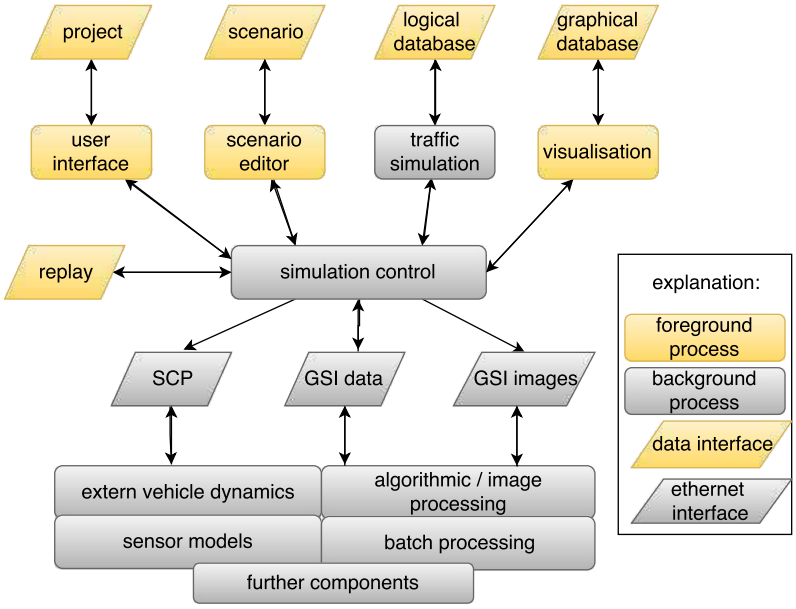


Abbildung 6.1: Virtual Test Drive Architektur nach [16]

Fahrzeugumfelds, also alle Bewegungen im Fahrzeugumfeld, werden durch den Hintergrundprozess Verkehrssimulation (*traffic simulation*) berechnet und dann durch die Visualisierung grafisch dargestellt. Diese sind durch das Szenario gegeben und können mit Hilfe des *Scenario Editors* entwickelt werden.

Die statischen Aspekte des Fahrzeugumfelds sind in den Datenbasen enthalten, wobei in der logischen Datenbasis hauptsächlich die Straßennetze hinterlegt, und in der grafischen Datenbasis die optischen Merkmale des Umfelds enthalten sind.

6.2 Laufzeitphasen

VTD unterscheidet drei verschiedene Laufzeitphasen: Vorbereitungs-, Simulations-, und Wiedergabephase. Während der Vorbereitungsphase wird die Simulationsumgebung eingerichtet und für den zu simulierenden Anwendungsfall konfiguriert. Dabei gibt es zwei Arten von Konfigurationen: Die statische Konfiguration nutzt XML-Dateien für jede Komponente, um die Einstellungen festzulegen und die dynamische Konfiguration erlaubt es bestimmte Einstellungen während der Simulationsphase über das Kommandoprotokoll SCP zu tätigen. Außerdem wird die konkrete Verkehrssituation erstellt, in der simuliert werden soll.

In der Simulationsphase findet die eigentliche Simulation statt. Bei den *Driver-in-the-Loop*- und *Vehicle-in-the-Loop*-Verfahren werden dabei die Fahrereingaben (Gas, Bremse, Lenkung) interaktiv und in Echtzeit über verschiedene Eingabegeräte in die Simulation übertragen. Bei den nicht interaktiven Verfahren *Software-in-the-Loop* und *Hardware-in-*

the-Loop werden die geplanten Fahraktionen des Fahrzeugs durch den Scenario Editor schon vorab konfiguriert.

Die Wiedergabephase ist dafür da, um die Simulation noch einmal abzuspielen und in einzelnen Zeitschritten durchzugehen. Während der Simulationsphase können die Bewegungen und Zustände aller Objekte der Simulation in einer Replaydatei aufgezeichnet werden, die dann für die Wiedergabephase genutzt wird und alles exakt wie in der Simulationsphase simuliert. Allerdings sind in dieser Phase keine Änderungen der Zustände mehr möglich. Die Replaydatei ist entweder eine Datei im Binärformat (*.dat) oder eine CSV-Datei, in der jede Zeile einen Simulationsschritt darstellt.

6.3 Simulationssteuerung

Die Simulationssteuerung ist für die Synchronisation der einzelnen Prozesse zuständig. Dies geschieht, indem sie die Taktung der Simulation bestimmt. Entweder wird dieser Takt durch die Simulationssteuerung selbst vorgegeben oder durch eine externe Komponente empfangen, beispielsweise bei dem *Hardware in the Loop (HIL)*-Verfahren, und dann von der Simulationssteuerung an die anderen Komponenten weitergegeben und kontrolliert. Sie ist mit den einzelnen Komponenten verbunden, sodass sie mit diesen über Protokolle kommunizieren kann. Des Weiteren kontrolliert die Simulationssteuerung die Komponenten bzw. die Simulation, indem sie von allen Komponenten zum Start der Simulation und danach in zyklischen Zeitabständen Statusnachrichten abfragt. Kommt von einer Komponente nach einer bestimmten Zeit keine solche Statusnachricht an, so wird die gesamte Simulation blockiert.

6.4 Das Fahrzeugumfeld

Im Grunde werden alle Elemente der Simulation Objekte genannt, also Fahrzeuge, Fußgänger, Ampeln, Straßenschilder, etc. Das Umfeld des Fahrzeugs sind dann alle Objekte, die sich innerhalb eines vordefinierten Radius um das Fahrzeug befinden.

Die Dauer der Fahrzeit entspricht der Simulationszeit und stellt eine Zeitreihe dar. Das heißt, dass jedem Objekt zu jedem Zeitpunkt der Zeitreihe ein Zustand zugeschrieben werden muss. Die Objekte werden in statische und dynamische Objekte unterteilt. Dabei sind statische Objekte solche, die ihren Zustand über die Simulationszeit hinweg nicht ändern können, während dynamische Objekte dies können, aber nicht zwingend tun müssen. Dadurch können statische Objekte zu einer statischen Datenbasis zusammengefasst werden und müssen nur zu Beginn einmal mit einem Zustand versehen werden. Die dynamischen Objekte werden ebenfalls mit ihren initialen Zuständen in die statische Datenbasis eingefügt, jedoch muss zusätzlich eine Logik definiert werden, die beschreibt, wie sich die

Zustände der Objekte während der Simulation ändern, d.h. wann oder unter welchen Voraussetzungen sich die Objekte wohin bewegen. Diese Logik wird von der Verkehrssimulation berechnet. Die statische Datenbasis teilt sich auf in grafische und logische Datenbasis. In der grafischen Datenbasis sind alle statischen Objekte mit ihren optischen Eigenschaften enthalten, sodass sie ordentlich visualisiert werden können.

VTD verwendet den in der Industrie etablierten Standard *OpenFLIGHT* [17] für die grafische Datenbasis. Dies erlaubt es, dass für die Modellierung neuer statischer Datenbasen auf Bibliotheken von statischen Objekten zurückgegriffen werden kann. Solche Bibliotheken enthalten beispielsweise alle gängigen Straßenschilder der Straßenverkehrsordnung. Die logische Datenbasis ist ebenfalls Teil der statischen Datenbasis und wird im *OpenDRIVE*-Format [14] beschrieben. Sie enthält unter anderem die Beschreibung des Straßennetzes mit den entsprechenden physikalischen Eigenschaften, wie beispielsweise die Fahrbahnbeschaffenheit, gewisse Oberflächeneigenschaften oder ähnliche Aspekte. Außerdem sind Informationen zur Steuerung des Verkehrsflusses wie Ampelprogramme, die Logik der Straßenschilder und Ähnliches enthalten. Um solche Datenbasen zu erstellen, gibt es das Werkzeug *Road Designer* [23]. Dies ist ein interaktiver grafischer Editor, mit dem Straßennetze erzeugt werden können. Dieser stellt Bibliotheken für verschiedene Objekte zur Verfügung und bietet die Möglichkeit, sowohl grafisch wie auch logisch detailliert zu modellieren. Außerdem generiert Road Designer aus dem fertigen Straßennetz eine OpenFLIGHT-Datei, welche direkt die grafische Datenbasis darstellt, und eine OpenDRIVE-Datei, die die logische Datenbasis bildet. Diese können direkt in VTD importiert werden.

6.5 Verkehrssimulation

Die Verkehrssimulation berechnet die Bewegungen aller dynamischen Objekte. Dabei können Fahrermodelle, die individuell auf spezielle Fahrertypen parametrisierbar sind, hinterlegt werden. Im Fahrermodell können Aspekte wie beispielsweise die Höchstgeschwindigkeit, mit der ein Fahrer auf Strecken ohne Geschwindigkeitsbegrenzung fährt, wie schnell ein Fahrer beschleunigt, wie genau sich an Geschwindigkeitsgebote gehalten wird, wie viel Sicherheitsabstand gehalten wird, ab wann ein Fahrer zum Überholen ansetzt und noch einige weitere Aspekte festgelegt werden. Des Weiteren existiert die Option des so genannten Pulk-Verkehrs. Dies ist eine Form von Schwarmverhalten der Fahrzeuge während der Simulation. Beim Pulk-Verkehr existiert ein Radius um ein gewisses Fahrzeug (im Regelfall das Egofahrzeug) und nur innerhalb dieses Radius existieren weitere Fahrzeuge. Verlässt ein Fahrzeug diesen Radius, wird es aus der Simulation entfernt und am Ende des Pulks wird ein neues Fahrzeug generiert.

Aktionen von Fußgängern und Fahrzeugen können auch deterministisch definiert werden. Diese werden dann ausgeführt, sobald gewisse Triggerpunkte aktiviert werden. Trigger-

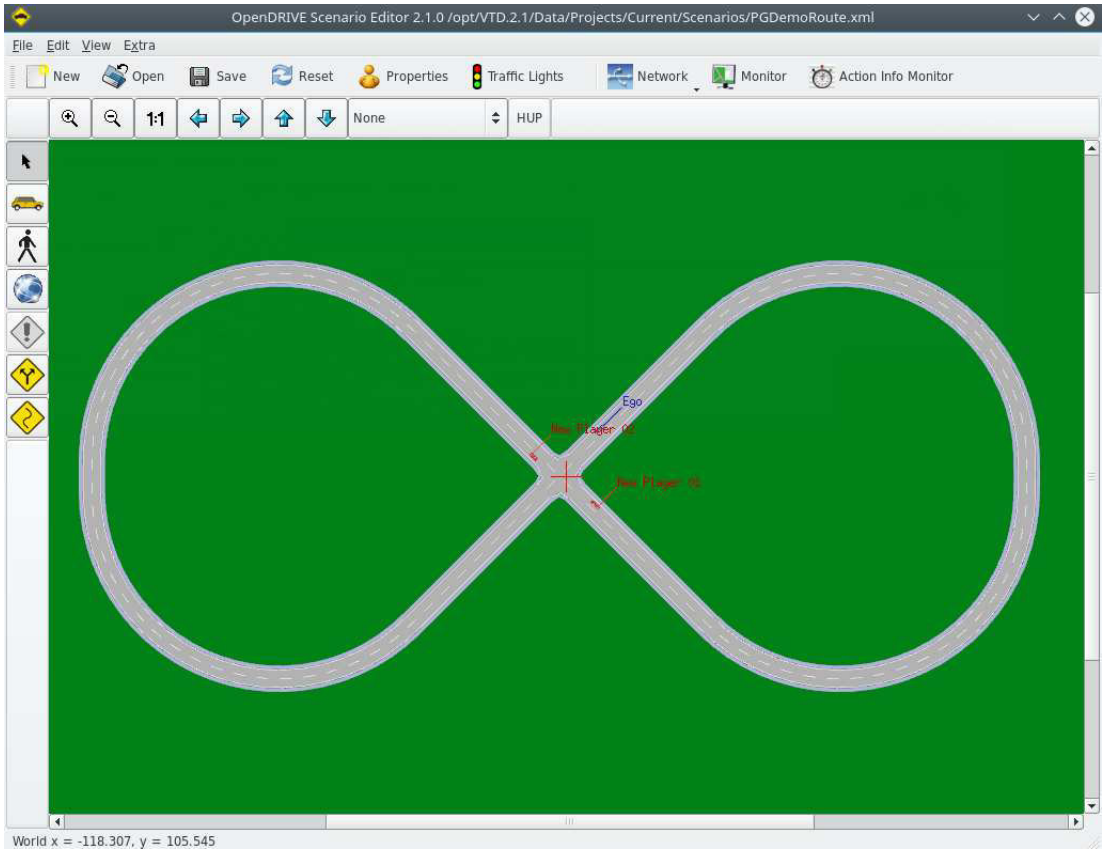


Abbildung 6.2: SzenarioEditor Hauptfenster

punkte können sein, wenn ein Fahrzeug oder ein Fußgänger eine gewisse Koordinate im Szenario erreicht oder wenn ein Fahrzeug oder Fußgänger einen bestimmten Radius um ein anderes Fahrzeug betritt. Es können auch Counter für Triggerpunkte festgelegt werden, sodass eine Aktion nur ausgelöst wird, wenn der Counter größer, kleiner oder gleich einer vordefinierten Zahl ist. Außerdem kann für Fahrzeuge und Fußgänger noch festgelegt werden, ob diese sich anhand bestimmter Pfade oder völlig frei durch das Gelände bewegen.

Diese gesamten Aspekte der Verkehrssimulation können über den Scenario Editor erstellt und bearbeitet werden.

6.6 SzenarioEditor

In Abbildung 6.2 ist das Hauptfenster des SzenarioEditors dargestellt. Der SzenarioEditor lädt das gleiche Szenario über eine XML-Datei, welches in VTD derzeit geladen ist. Dort wird in blau das Fahrzeug angezeigt, welches extern gesteuert werden kann. Hier heißt dieses Ego. Die roten Fahrzeuge werden von VTD simuliert und gesteuert. Das Ego-Fahrzeug ist also dasjenige, auf welchem später das im Rahmen der Projektgruppe entwickelte ADAS

läuft und getestet wird.
Ein Fahrzeug innerhalb des SzenarioEditors kann einer Route folgen oder einem vordefinierten Pfad. Eine Route hat die Eigenschaft, dass diese einen Zyklus darstellt. Diese Route kann beliebig oft befahren werden. Ein Pfad kann vom Nutzer beliebig gesetzt werden. Neben Routen können über den SzenarioEditor ebenfalls Aktionen erstellt werden, indem über das Hauptfenster in den Aktionsmodus gewechselt wird. Diese können dann - wie Routen auch - den Fahrzeugen über deren Eigenschaftenfenster zugewiesen werden.

Teil II

Konzept

Kapitel 7

AutoDSL

In diesem Abschnitt wird die zur Modellierung von Fahrassistenzsystemen entwickelte DSL erläutert. Hierfür wird auf die praktische Umsetzung des Entwurfs mit CINCO eingegangen. Die Umsetzung ist aufgeteilt in die Umsetzung der Modelle und die Umsetzung des Verhaltens- bzw. Datenflussmodells.

7.1 Entwurf

Die DSL soll drei zentrale Funktionen haben. Erstens soll modelliert werden können, welche Zustände ein Fahrassistenzsystem besitzt und unter welchen Bedingungen sich dieser Zustand ändert. Zweitens soll modelliert werden können, welches Verhalten das Fahrassistenzsystem mit welchem Zustand aufweist. Bei diesem Modell muss es somit möglich sein, die Verarbeitung von Sensorwerten zu konkreten Handlungen zu modellieren. Zuletzt soll es möglich sein, das Verhalten des modellierten Fahrassistenzsystems gezielt zu überwachen und zu testen.

7.1.1 Zustandsmodell

Inspiriert von dem Zustandsmodell, welches in der ACC-ISO-Norm [13] verwendet wird, um die verschiedenen Zustände eines ACC zu beschreiben (vgl. Abbildung 2.10), wurde früh die Entscheidung getroffen als Kern der DSL ebenfalls ein Zustandsmodell zu verwenden. Jeder Zustand eines Fahrassistenzsystems entspricht einem Zustand in diesem Modell. Jeder Zustand enthält dann ein entsprechendes Verhaltensmodell, welches das Verhalten in diesem beschreibt. Die Zustände werden mit Kanten verknüpft. Diese Kanten besitzen zusätzlich Bedingungen, oder auch *Guards*, welche steuern, ob eine Übergang durchgeführt wird. Dieses theoretische Konzept wird in Abbildung 7.1 dargestellt.

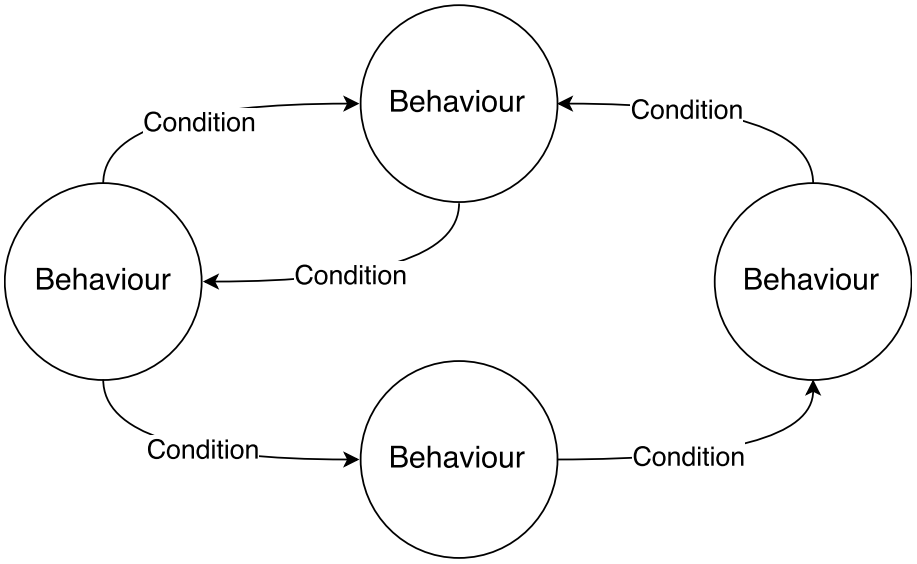


Abbildung 7.1: Das theoretische Zustandsmodell

7.1.2 Verhaltensmodell/Datenflussmodell

Um das Verhalten zu modellieren, müssen Eingabedaten (wie Sensorwerte o.Ä.) verarbeitet werden können und dann die Ergebnisse an die Systeme des Fahrzeugs weitergeleitet werden. Folgende Funktionen sollen dem Nutzer hierbei bereitstehen:

- Arithmetische Operatoren
- Boolesche Operatoren
- Vergleichsoperatoren
- If-Then-Else-Strukturen
- Definition von statischen Werten zur späteren Verwendung
- Ein/Ausgabe von Fahrzeugwerten
- Zwischenspeichern von Werten über mehrere Zyklen
- Verwendung von Submodellen, um sich so eigene Funktionen zu definieren
- Verwendung von C++-Ausdrücken für komplexe Formeln

Eine frühe Idee war es, hier eine textuelle DSL statt einer graphischen DSL zu verwenden. Die Vorteile einer textuellen DSL wären hier eine kompaktere Darstellung und schnellere Entwicklung gewesen. Allerdings ist bei den ersten Entwürfen und dem Versuch beispielhafte Umsetzungen zu implementieren recht schnell klar geworden, dass bei einer graphischen DSL der Überblick über den Ablauf klarer und die Verwendung intuitiver ist.

Deshalb wurde für das finale Modell ein graphischer Ansatz gewählt, welcher sich an einer Kombination aus Kontrollflussdiagrammen und Datenflussdiagrammen orientiert. Die Operationen besitzen Ein- und Ausgabewerte, welche mit Werten der passenden Datentypen verbunden werden können. Außerdem existieren Kanten zwischen den Operationen, welche die Reihenfolge der Operationen angeben. Ein Beispiel ist in Abbildung 7.2 zu sehen. Hier verarbeitet die erste Operation einen Eingabewert aus dem Fahrzeug, wie zum Beispiel einen Sensorwert oder einen Knopfdruck, und gibt die Ausgabe an die nächste Operation weiter. Diese verwendet den Ausgabewert und einen weiteren Fahrzeugwert, um schließlich eine Ausgabe an das Fahrzeug zu machen. Da für die Definition der Zustandsübergangs-

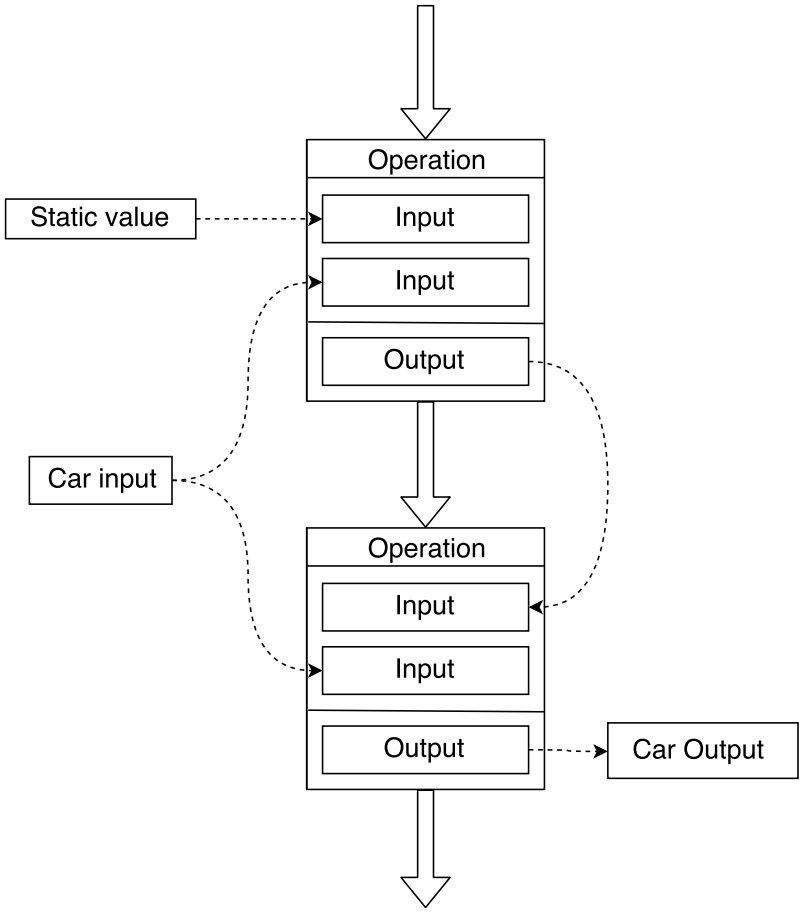


Abbildung 7.2: Beispielhafter Ausschnitt aus dem theoretischen Modell zur Beschreibung des Datenflusses eines Fahrassistenzsystems

bedingungen die gleichen Funktionen und Operatoren benötigt werden (mit Ausnahme der Ausgabe von Werten an das Fahrzeug), kann hierfür ein beinahe identisches Modell verwendet werden. Hier muss dann lediglich anstatt einer Ausgabe von beliebigen Werten an das Fahrzeug eine Ausgabe eines einzelnen Wahrheitswertes, welcher angibt, ob der Übergang stattfindet, ermöglicht werden.

Kapitel 8

Monitoring

Um zu überprüfen, ob sich das entwickelte ACC wie erwartet verhält und der Norm entspricht, muss ein System zur Überwachung des Verhaltens des ACC entwickelt werden. Hierfür wurde die Entscheidung getroffen, eine textuelle DSL zu entwickeln, welche es ermöglichen soll, Regeln in der Form von aussagenlogischen Implikationen zu definieren. Das bedeutet, es werden einzelne Tests definiert. Diese haben drei Elemente:

- Eine Vorbedingung, welche bestimmt wann der Test durchgeführt werden soll,
- eine Definition des Tests,
- einen Optionen-Block, welcher verschiedene Einstellungsmöglichkeiten enthält.

Die Entscheidung, eine textuelle DSL zu verwenden, basiert zum Großteil auf der Tatsache, dass bei der Umsetzung von Vorgaben aus der ACC-ISO-Norm [13] zwangsläufig aussagenlogische Ausdrücke entstehen. Diese könnte man zwar in einer grafischen DSL darstellen, aber gerade bei längeren Ausdrücken mit verschiedenen Operatoren ist eine textuelle Form unserem Ermessen nach deutlich übersichtlicher, intuitiver und mit einer deutlichen Aufwandsreduktion bei der Verwendung dieser verbunden. Nimmt man als Beispiel eine simple Bedingung wie [Beschleunigung] $>$ 50 && [ACC ist nicht im Zustand 'Following'], würde man diese in unserer textuellen DSL wie folgt umsetzen: Throttle $>$ 50 and is not Following. In einer grafischen DSL, die ähnlich aufgebaut ist, wie unsere Rule-DSL, würde man vermutlich mindestens drei Knoten und dazugehörige Kanten erstellen müssen, wodurch für eine relativ einfache Bedingung bereits ein relativ komplexes Teilmodell gebaut werden muss.

8.1 Syntax der Monitoring-Sprache

Die Monitoring-Sprache wurde mit Xtext erstellt. Der Test ist das Kernelement der DSL. Wie bereits in Kapitel 8 beschrieben, besteht ein Test aus einer Vorbedingung, einer Nachbedingung und einem Optionen-Block. Diese Elemente werden folgendermaßen in einem Test beschrieben:

- Vorbedingung: Conditions-Block
- Nachbedingung: Invariants-Block
- Optionen: Options-Block (optional)

```
def Test testA{
  Invariants{
    expr1,
    expr2,
    1 <= 2
  }
  Conditions{
    expr4
  }
}
```

Code-Ausschnitt 8.1: Beispiel eines Tests

Die Conditions- und Invariants-Blöcke enthalten jeweils eine Liste von Expressions. Jede dieser Expressions muss ein Ausdruck sein, der zu einem *Boolean* ausgewertet werden kann. Hierbei werden Expressions verundet, die durch ein Komma getrennt sind. Im Beispiel Code-Ausschnitt 8.1 wird also, sobald der Ausdruck expr4 *True* ergibt, überprüft, ob expr1, expr2 und $1 \leq 2$ *True* sind. Ist dies nicht der Fall, liegt ein Fehler vor.

8.1.1 Monitor und Configuration

Es besteht außerdem die Möglichkeit, mehrere Tests in Monitor-Blöcken zusammenzufassen. Im Code-Ausschnitt 8.2 meldet der Monitor monitorA einen Fehler, sobald testA, testB oder testC einen Fehler melden, also eine Expression in einem der Invariants-Blöcke zu *False* ausgewertet wird.

```
def Monitor monitorA {
  testA, testB, testC
}
```

Code-Ausschnitt 8.2: Beispiel eines Monitors

Analog dazu kann man mehrere Monitore zu einer Configuration zusammenfassen. Diese Monitore werden dann angewendet (siehe Beispiel Code-Ausschnitt 8.3).

```
1 def Configuration config {
2   monitorA, monitorB
3 }
```

Code-Ausschnitt 8.3: Beispiel einer Configuration

8.1.2 Options

Folgende zusätzliche Optionen können im Options-Block gesetzt werden:

- Delay: Die Überprüfung der Expressions im Invariants-Block wird nach dem Eintreten der Expressions im Conditions-Block um Delay Zyklen verzögert.
  Default: 0.
  Beispiel: Delay = 5.
- TimesToRun: Der Test wird nur TimesToRun-mal ausgeführt.
  Default: -1.
  Beispiel: TimesToRun = 2.
- RunFrequency: Der Test wird nach dem Prüfen der Expressions im Conditions-Block für RunFrequency Zyklen pausiert.
  Default: -1.
  Beispiel: RunFrequency = 2.

8.1.3 Expressions

Durch Expressions können boolesche Ausdrücke jeglicher Art beschrieben werden. Dazu stehen folgende Operatoren zur Verfügung: und-Verknüpfung (and), oder-Verknüpfung (or), $<$, $<=$, $==$, $!=$, $>$, $>=$, $+$, $-$, $*$, $/$ und Negation ($!$ und $-$). Außerdem können Expressions geklammert werden. Die Operatoren haben dabei die folgende Rangfolge:

1. and
2. or
3. $<$, $<=$, $==$, $!=$, $>$ und $>=$
4. $+$ und $-$
5. $*$ und $/$
6. Negation ($!$ und $-$)

Diese Operatoren können mit *True*, *False*, einfachen Integerwerten (z.B. 42) und Kommazahlen (z.B. 15.473) verwendet werden. Außerdem stehen diverse Fahrzeugwerte zur Verfügung.

Fahrzeugwerte

Als Fahrzeugwerte können Werte verwendet werden, die direkt vom Fahrzeug ausgegeben werden, und Werte, die direkt an das Fahrzeug ausgegeben werden können. Die Bezeichner der Werte entsprechen den Ein-/Ausgängen des *Cinco-Masterfilters* (siehe Unterabschnitt 12.1.2) und somit den Werten, die in der *AutoDSL* bereit stehen. Eine Liste mit verfügbaren Fahrzeugwerten befindet sich in Anhang A.

Werte aus der *AutoDSL*

Um Werte aus der *AutoDSL* in der Monitoring-Sprache zu verwenden, müssen Modelle mit einem import referenziert werden. Hierfür werden ein Bezeichner und ein Pfad benötigt. In Code-Ausschnitt 8.4 ist dies exemplarisch zu sehen.

```
1 import "myDSL.autodsl" as states
2 import "SharedMemory/acc_memory.sharedMemory" as sharedMem
```

Code-Ausschnitt 8.4: Beispiel für die Referenzierung

Werte aus folgenden Modellen können referenziert werden:

- *SharedMemory*: Innerhalb von Expressions können in *SharedMemory-Modellen* hinterlegte Werte mit folgender Syntax abgerufen werden (für den Wert wert aus dem Modell mod): mod.wert
- *AutoDSL*: Der aktuelle Zustand in einem *AutoDSL-Modell* kann überprüft werden. Hierfür wird folgende Syntax verwendet (für den Zustand state im Modell mod): is mod.state oder is not mod.state.
- Historische Werte: Für importierte Werte und für Fahrzeugwerte ist es möglich, historische Werte abzurufen. Hiermit können zum Beispiel das Steigen eines Wertes oder die Änderung eines Zustandes überprüft werden. Möchte man zum Beispiel den Wert von Throttle vor 14 Zyklen erhalten würde man folgende Syntax verwenden: Throttle[14].

8.2 Validierung

Bei der Erzeugung der DSL aus der Xtext-Definition entsteht zusätzlich zu dem Parser ein Eclipse-Editor für Dateien vom entsprechenden Typ. Dieser Editor unterstützt au-

tomatisch nützliche Funktionen wie eine Autovervollständigungsfunktion und detaillierte Fehlermeldungen bei Syntaxfehlern. Zusätzlich zu diesen werden aber weitere Funktionen zur Validierung der Monitore benötigt. Außerdem muss zum Beispiel überprüft werden, ob referenzierte Dateien existieren und ob alle Elemente für eine vollständige Monitordefinition vorhanden sind. Folgende Aspekte müssen zum Beispiel beachtet werden:

- Korrekte Anzahl an TestFeatures und OptionsFeatures:
  Ein Test besteht aus drei Elementen, welche in beliebiger Reihenfolge definiert werden können. Hier muss sichergestellt werden, dass jedes Element maximal einmal vor kommt und alle Blöcke außer Options mindestens einmal vor kommen. Ähnlich verhalten sich die verschiedenen Options-Parameter, welche in beliebiger Reihenfolge optional vorkommen können, aber maximal einmal pro Options-Block.

- Überprüfung der Typen von Ausdrücken:
  Um sinnlose Ausdrücke wie das oben erwähnte $2 - \text{false} >= 15$ zu verhindern, wird die Tatsache ausgenutzt, dass für alle Daten schon im Voraus die Typen bekannt sind. Das ermöglicht eine relativ einfache statische Typisierung. Hierfür wird eine rekursive Funktion getExpressionType(Expression expr) verwendet, welche den Typ bestimmt. Hierbei werden drei Typen unterschieden: Boolean, Number und der Fehlerfall Unknown, wenn der Typ nicht bestimmt werden kann. So ist zum Beispiel der Typ von expr1 <= expr2 immer dann Boolean, wenn expr1 und expr2 beide den Typ Number haben. Ist dies nicht der Fall, wird der Typ Unknown zurück gegeben. So kann der Fehlertyp Unknown bis zum obersten Expression-Element durchgereicht werden. Um dann sicherzustellen, dass alle selbst definierten Variablen sinnvolle Typen haben, wird folgende Funktion verwendet:

```
1  @Check
2  def checkExpressionType(Variable variable){
3    if(getExpressionType(variable.expr)==ExpressionType.TUnknown){
4      error([...])
5    }
6  }
7
```

Code-Ausschnitt 8.5: Methode zur Überprüfung des Typs einer Variablen

- Eindeutige Bezeichner:
  Es wird standardmäßig nicht verhindert, dass zwei Elemente den gleichen Bezeichner haben. Dies könnte zur Folge haben, dass bei einer Referenz über diesen Bezeichner unklar ist, welches Element gemeint ist. Zur Sicherheit wurde eine Funktion ergänzt, die doppelt verwendete Bezeichner als fehlerhaft markiert.

- Importieren von AutoDSL-Modellen:
  Um ein AutoDSL-Modell zu importieren wird ein Pfad zur Modelldatei angegeben.

Beispiel: import "myDSL.autodsl" as states. Hier müssen mehrere Dinge überprüft werden: Existiert die Datei? Kann sie geöffnet werden? Hat sie den korrekten Typ? (Endet sie auf ".autodsl" oder ".sharedMemory"?)

- Korrekte Delays:
  Verwendet der Test Werte aus vergangenen Zyklen, kann es zu Beginn der Ausführung zu unerwarteten Werten kommen. Beispiel: Bei Throttle[10] wird der Throttle-Wert vor 10 Zyklen verwendet, dieser ist aber für die ersten 10 Zyklen noch nicht definiert und hat einen Defaultwert von 0. Wenn dies nicht beachtet wird, kann es somit zu Beginn einer Fahrt oder einer Simulation zu Problemen kommen. Der Delay-Wert eines Tests verzögert die mögliche erste Ausführung eines Tests um die entsprechende Anzahl Zyklen. Um sicherzustellen, dass keine Fehler durch einen fehlenden oder zu kleinen Delay-Wert entstehen, wird für jeden Test überprüft ob Delay größer ist, als der größte verwendete Zugriff auf Vergangenheitswerte. Ist dies nicht der Fall, erhält der Nutzer eine Warnung, dass möglicherweise zu Beginn unerwartete Werte zustande kommen können.

Kapitel 9

SzenarienDSL

In diesem Abschnitt wird die *SzenarienDSL* vorgestellt, welche verwendet wird, um Szenarien formal zu beschreiben und aus dieser formalen Beschreibung ein Szenario in der *VTD-Simulationsumgebung* zu generieren. Hierfür wird zuallererst im ersten Abschnitt die Idee der *SzenarienDSL* vorgestellt. Anschließend wird im zweiten Abschnitt der Aufbau des *SzenarienDSL-Editors* und dessen Funktionsweise beschrieben. Danach wird im dritten Abschnitt auf die Implementierung des Generators eingegangen. Daraufhin werden zum Schluss im vierten Abschnitt Testszenarien vorgestellt.

9.1 Idee

Nachdem mit der zuvor vorgestellten *AutoDSL* ein Fahrassistenzsystem modelliert und generiert wurde, muss dieses *Generat* auf seine Korrektheit überprüft werden. Hierfür kann die zuvor vorgestellte *TestDSL* verwendet werden. Die daraus generierten Tests prüfen das generierte Fahrassistenzsystem auf korrektes Verhalten. Das Verhalten des Assistenzsystems kann nun in verschiedenen Fahrsituationen überprüft werden.

Weiterhin ist es wünschenswert bestimmte Fahrsituationen bzw. Fahrszenarien in einer Simulationsumgebung abzubilden. Das Szenario beschreibt die Fahrsituation und das Verhalten aller Fahrzeuge. Die Fahrzeuge sollen dieses Szenario in der Simulation automatisch durchführen. Bisher muss hierfür das Szenario aufwendig im *VTD-Szenarieneditor* konfiguriert werden. Eine leichtere grafische Beschreibungsmöglichkeit von Szenarien, aus der die Szenarien in der Simulationsumgebung generiert werden können, ist wünschenswert. Diese *SzenarienDSL* kann ihre Szenarien lediglich für Simulationen generieren, da das Generieren von realen Szenarien viel zu umständlich, zum Teil sehr teuer (Unfallszenarien) und zum Teil schwer umsetzbar (genaue Abstände, Teleportation, ...) wäre.

9.2 Aufbau

In Abbildung 9.1 ist der *SzenarienDSL-Editor* abgebildet. An der rechten Seite des Editors ist die *Elementpalette*, die *Car-*, *Action-*, *Start- /End-* und *Conditionelemente* beinhaltet. Dabei können diese Elemente mittels *Drag&Drop* in den *Szenarienbereich* in der Mitte gezogen werden. Ein Szenario beginnt immer mit einem *Startelement* und endet mit einem *Endelement*. Dazwischen sind *Actionelemente*, mittels Transitionsbedingung (*Conditions*) verbunden. Im unteren Teil des Editors kann die aktuell angeklickte *Action* bzw. *Condition* konfiguriert werden. In der oberen linken Ecke des *Szenarienbereiches* ist der *CarPool*, der alle im Szenario auftretenden Fahrzeuge beinhaltet. Auf diese Fahrzeuge kann innerhalb einer *Action* oder einer *Condition* referenziert werden. Im linken unteren Quadranten des Editors werden die fehlgeschlagenen *Checks* angezeigt, welche die Korrektheit des Szenarios prüfen.

Ein Szenario kann im *Projekt-Explorer* auf der linken Seite erstellt werden. Weiterhin existiert die Möglichkeit, Szenarien ineinander zu schachteln. Ein Szenario kann somit aus mehreren *Unterszenarien* bestehen, die in diesem Szenario wie *Actions* behandelt werden. Auf diese Weise können wiederkehrende *Teilszenarien* ausgelagert werden.

9.2.1 Cars

Es existieren zwei verschiedene Fahrzeugtypen, die in der *Elementpalette* ausgewählt werden können. Zum einen gibt es das *EgoCar*, bei dem es sich um das Fahrzeug handelt, auf welchem die zu testende Software läuft, in unserem Fall die mithilfe der *AutoDSL* entwickelten Assistenzsysteme. Es existiert immer genau ein *EgoCar* je Szenario. Als *Car* werden dagegen die anderen am Szenario beteiligten Fahrzeuge bezeichnet, auf welche das *EgoCar* zu reagieren hat. Von ihnen können beliebig viele existieren. Neben der Möglichkeit mittels Elementpalette neue Fahrzeuge zur Verwendung im Szenario zu erstellen, kann auch durch Nutzen des *CarPools* auf bereits bestehende Fahrzeuge zugegriffen werden. Der *CarPool* repräsentiert zu jeder Zeit den gegenwärtigen Bestand an Fahrzeugen, die im Szenario verwendet werden.

9.2.2 Actions

Actions beschreiben die von Fahrzeugen innerhalb eines Szenarios durchgeführten Handlungen, wie Geschwindigkeitsänderungen, Spurwechsel oder das Interagieren mit einem Assistenzsystem. Zu einem Zeitpunkt können mehrere *Actions* gleichzeitig initiiert werden. Dazu werden sie in einem *ParallelActions-Container* gesammelt aufgeführt. Eine Geschwindigkeitsänderung wird mittels *SpeedChange* eingeleitet. Die *Action* gibt an, zu welcher Zielgeschwindigkeit hin das/die jeweils ausgewählten Fahrzeuge beschleunigen bzw. abbremsen sollen. *LaneChange* ermöglicht das Abbilden von Spurwechseln. Erneut können

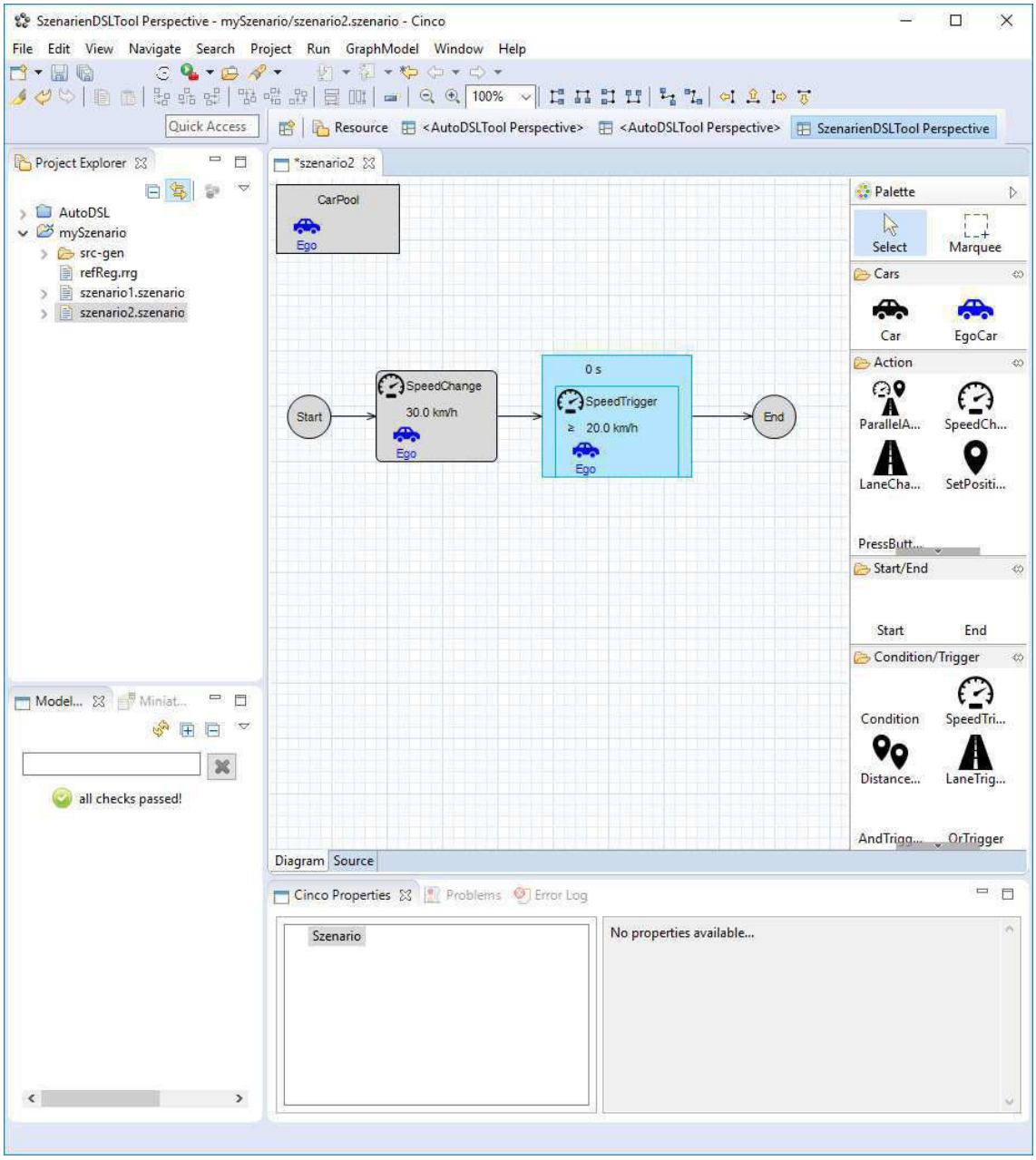


Abbildung 9.1: In dieser Abbildung ist der *SzenarienDSL-Editor* abgebildet. Dabei wurde ein Beispielszenario namens *"szenario2.szernario"* geöffnet, das aus einer *Condition*, einer *Action* und einem *Start-* und *Endelement* besteht.

mehrere Fahrzeuge gleichzeitig betroffen sein. Angegeben wird die Anzahl und Richtung der durchzuführenden Wechsel. *SetPosition* ist die einzige Action, die nicht für das *EgoCar* ausgewählt werden kann. Sie dient dem Platzieren von anderen Verkehrsteilnehmern vor dem *EgoCar* in einem beliebigen Abstand. *PressButton* bildet die Steuerung des nur beim *EgoCar* vorhandenen Assistenzsystems ab und ermöglicht dabei insbesondere das Einschalten sowie das schrittweise In- und Dekrementieren von Sollgeschwindigkeit und eingestelltem Abstand.

9.2.3 Conditions

Innerhalb von *Conditions* werden Bedingungen formuliert, die eintreten sollen, bevor darauffolgende *Actions* ausgeführt werden. Diese Bedingungen (*Trigger*) können hierarchisch angeordnet konjugiert (*AndTrigger*) sowie disjugiert (*OrTrigger*) werden. Ein *SpeedTrigger* überprüft die Über- oder Unterschreitung bzw. das Erreichen einer Geschwindigkeit eines spezifizierten Fahrzeugs. Sowohl *Distance-* als auch *LaneTrigger* überprüfen den Abstand zweier Fahrzeuge. Hier ist ebenfalls beide Male wahlweise eine Prüfung auf Über- oder Unterschreitung bzw. das Erreichen eines ausgewählten Abstandes möglich. Der *DistanceTrigger* prüft den in Metern bemessenen Abstand zweier Fahrzeuge auf derselben Fahrspur. Für die Überprüfung muss spezifiziert werden, von welchem Fahrzeug (*Relative*) erwartet wird, dass es eine Position vor dem anderen Referenzfahrzeug (*Pivot*) einnimmt. Der *LaneTrigger* prüft den Abstand zweier Fahrzeuge auf eventuell unterschiedlichen Fahrspuren. Der Abstand wird hier in der Anzahl der Fahrspuren bemessen, welche sich das zweite Fahrzeug (*Relative*) links vom Referenzfahrzeug (*Pivot*) befindet.

9.2.4 Szenariensammlung

Üblicherweise bietet es sich an, eine entwickelte Software nicht nur in einem einzelnen Szenario zu testen. Um nun nicht jedes geschriebene Szenario einzeln zu generieren, wurde als weiterer Modelltyp die *Sammlung* entwickelt, welche mehrere Szenarien gebündelt an den Generator weitergibt, damit diese danach am Stück ausgeführt werden können. Eine neue *Szenariensammlung* kann auf der linken Seite des *SzenarienDSL-Editors* im *Projekt-Explorer* erzeugt werden. Wird die *Szenariensammlung* geöffnet, können Szenarien mittels *Drag&Drop* aus dem *Projekt-Explorer* in diese gezogen werden.

9.3 Generator

Beim Generieren einer Sammlung werden Szenarien für VTD erzeugt, die nacheinander ausgeführt werden. Dazu müssen einerseits die XML-Dateien, welche die Szenarien-Umgebung in VTD beschreiben, und andererseits ausführbarer C++-Code, welcher in einen ADTF-Filter integriert wird, generiert werden.

Die XML-Dateien bauen auf der in **??** beschriebenen Strecke auf. Das Ego-Fahrzeug wird in jedem Fall generiert und wird am Anfang der Strecke auf der rechten Spur platziert. Weitere erzeugte Fahrzeuge werden auf der Spur links daneben platziert. Die ursprüngliche Idee war, diese auch alle an den Anfang ihrer Strecke zu stellen. Das führte allerdings dazu, dass die Simulation direkt am Anfang einen Unfall erkannte, wodurch die entsprechenden Autos nicht mehr fahren konnten. Deshalb wird nun das erste Fahrzeug links neben das Ego-Fahrzeug platziert und alle Folgenden in ca. 5km Abständen zueinander aufgestellt. Alle generierten Autos haben eine Startgeschwindigkeit von 0km/h. Außerdem ist der Fahrzeugtyp der generierten Autos festgelegt.

Das Interface SCPSender wird dabei von einem ADTF-Filter implementiert, der für die Kommunikation zwischen dem generierten C++-Code und der VTD-Instanz zuständig ist. Zunächst ruft der Filter die Funktion SIMControll::runSzenarios() auf, in welche das erste Szenario in einem neuen Thread startet. Von diesem aus wird das jeweils nächste Szenario über die Methode SIMControll::nextSzenario() gestartet. Dabei wird zwischen den Szenarien die Methode SCPSender::resetADAS() aufgerufen, mit der das ADAS in den Startzustand zurückgebracht wird.

Die über *Realtime Data Bus (RDB)* ausgelesenen Werte für Positionen und Geschwindigkeiten der Fahrzeuge sendet der Filter über die Funktionen SIMControll::sendSpeed(int id, double speed) bzw. SIMControll::sendLaneInfo(int id, int LaneID, double LaneS) an die generierten Klassen zur Ausführung der Szenarien. Diese Werte werden in dem einer Liste von SiMCarData Structs vom SIMController gespeichert.

Befehle für das Ego-Fahrzeug werden über RDB gesendet. Für alle anderen Autos wird ein SCP-Kommando für eine Aktion gesendet. Da die Befehle zum Teil eine kurze Zeit benötigen, um bei der VTD-Simulation anzukommen, wurde eine kurze Verzögerung von 0,5s nach dem Absenden eines Befehls eingebaut.

9.4 Testfälle

Da das in der PG entwickelte ACC für gerade Strecken konzipiert ist, wurde eine simple gerade Strecke ohne Steigungen oder Senkungen mit dem RoadDesigner entworfen, wie sie in Abbildung 9.2 zu sehen ist. Diese Strecke wurde auf eine Länge von etwa 100 km verlängert. Auf Objekte wie bspw. Straßenschilder oder Ampeln und Landschaften jenseits der Straße wurde komplett verzichtet, zum einen der Einfachheit halber und zum anderen, um den Rendering-Aufwand während der Simulation geringer zu halten. Die Strecke wird im Folgenden als Grundgerüst für alle generierten Testfälle bzw. Szenarien verwendet.

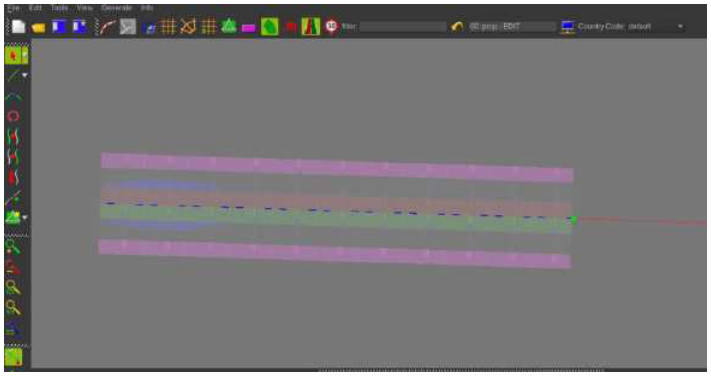


Abbildung 9.2: Layout einer geraden Strecke

Um das in Abschnitt 2.3 entwickelte ACC zu testen bzw. zu verifizieren, wurden mit Hilfe der *SzenarienDSL* für die Simulationsumgebung VTD einige Testfälle in Form von einer Sammlung von Szenarien modelliert und zu jedem Szenario jeweils eine genaue textuelle Beschreibung des Testfalls formuliert. Ein Testfall bzw. seine textuelle Beschreibung besteht jeweils aus der Beschreibung von vier Aspekten:

1. Initiale Situation
2. Vorbedingungen
3. Aktion(en)
4. Nachbedingungen

Zur Überprüfung der Vor- und Nachbedingungen wird die in Kapitel 8 beschriebene *MonitorDSL* verwendet. Zusätzlich überprüfen die Monitore jeweils Invarianten, die aus der *ACC-ISO-Norm* folgen.

Die folgenden Testfälle wurden modelliert und beschrieben.

1. Das Ego-Fahrzeug fährt einem vorausfahrenden Fahrzeug auf, dabei soll das ACC den eingestellten Zeitabstand einhalten und dementsprechend seine Geschwindigkeit dem vorausfahrenden anpassen, ohne die maximale Bremswirkung zu überschreiten.
2. Das Ego-Fahrzeug folgt einem vorausfahrenden Fahrzeug mit einer niedrigeren Geschwindigkeit, als die, die im ACC eingestellt ist. Dann schert das vorausfahrende Fahrzeug aus, sodass das Ego-Fahrzeug zurück auf seine eingestellte Geschwindigkeit beschleunigen kann.
3. Das Ego-Fahrzeug folgt einem vorausfahrenden Fahrzeug mit einer niedrigeren Geschwindigkeit, als die, die im ACC eingestellt ist. Dann beschleunigt das vorausfahrende Fahrzeug, jedoch nur soweit, dass die vom Ego-Fahrzeug eingestellte Geschwindigkeit nicht überschritten wird.

4. Das Ego-Fahrzeug folgt einem vorausfahrenden Fahrzeug mit einer niedrigeren Geschwindigkeit, als die, die im ACC eingestellt ist. Dann beschleunigt das vorausfahrende Fahrzeug über die im ACC eingestellte Geschwindigkeit hinaus.

5. Das Ego-Fahrzeug folgt einem vorausfahrenden Fahrzeug. Dann bremst das vorausfahrende Fahrzeug ab, jedoch nur soweit, dass die minimale Geschwindigkeit des ACCs nicht unterschritten wird.

6. Das Ego-Fahrzeug folgt einem vorausfahrenden Fahrzeug. Dann bremst das vorausfahrende Fahrzeug soweit ab, dass die minimale Geschwindigkeit des ACCs unterschritten wird und sich dieses ausschaltet.

7. Das Ego-Fahrzeug folgt einem vorausfahrenden Fahrzeug. Ein weiteres Fahrzeug schert in die Mitte zwischen Ego- und vorausfahrendem Fahrzeug ein.

8. Das Ego-Fahrzeug fährt eine konstante Geschwindigkeit ohne vorausfahrendes Fahrzeug. Dann schert ein Fahrzeug vor dem Ego-Fahrzeug ein, welches jedoch langsamer fährt als das Ego-Fahrzeug.

9. Das Ego-Fahrzeug fährt eine konstante Geschwindigkeit ohne vorausfahrendes Fahrzeug. Dann schert ein Fahrzeug vor dem Ego-Fahrzeug ein, welches schneller fährt als das Ego-Fahrzeug.

Für die hier beschriebenen Testfälle sind verschiedene Ausprägungen vorhanden. Diese Ausprägungen betreffen die Geschwindigkeit der Fahrzeuge und den beim Ego-Fahrzeug im ACC eingestellten Zeitabstand. So gibt es für jeden Testfall die Ausprägung, dass der Zeitabstand 1,0 s und 1,8 s beträgt. Des Weiteren werden für jeden Zeitabstand die Testfälle in verschiedene Geschwindigkeiten der Fahrzeuge unterteilt. Jeder oben beschriebene Testfall existiert also mit verschiedenen Kombinationen aus Zeitabstand und gefahrener Geschwindigkeit. Insgesamt gibt es dadurch 36 Testfälle.

In Abbildung 9.3 ist Testfall Punkt 4 zu sehen, die Parametrisierung ist dabei wie folgt: Der eingestellte Zeitabstand vom ACC beträgt 1,0 s. Ego- und vorausfahrendes (*NewCar*) Fahrzeug fahren konstant 30 km/h, sodass das Ego-Fahrzeug dem NewCar mit eingestelltem Zeitabstand folgt. Hierbei wären das 8,33 m. Die beim Ego-Fahrzeug im ACC eingestellte Geschwindigkeit beträgt jedoch 50 km/h. Dies bildet die initiale Situation dieses Testfalls und spiegelt ebenfalls die Vorbedingungen wieder, welche von den Monitoren überprüft werden. Zusätzlich dazu gehört noch die Vorbedingung, dass sich das Ego-Fahrzeug im Zustand „Following“ befinden muss.

In den meisten Modellen der Testfälle stellt das Erzeugen der initialen Situation den größten Teil des Modells dar. So auch in diesem, wie in Abbildung 9.3 zu sehen, denn sie geht bis nach der *SetPosition*-Aktion. Zuerst müssen nach dem Startknoten beide Fahrzeuge be-

schleunigt werden, dies geschieht in dem ersten *ParallelActions*-Knoten. Ego beschleunigt zunächst auf 50 km/h und NewCar auf 30 km/h und das ACC von Ego wird eingeschaltet, sodass es sich im Zustand „Idle“ befindet. Die *Condition* danach prüft, ob das Ego-Fahrzeug 50 km/h erreicht hat. Sobald dies geschehen ist, wird der nächste *ParallelActions*-Knoten ausgeführt. In diesem wird das ACC mit der aktuellen Geschwindigkeit, in diesem Fall 50 km/h, aktiviert. Das ACC wechselt somit in den Zustand „SpeedControl“. Dann wird fünf mal die Taste zum Reduzieren des Zeitabstands gedrückt, sodass sich ein Zeitabstand von 1,0 s einstellt, da vom Standardwert 1,5 s fünf mal um 0,1 s dekrementiert werden muss. Die *SzenarioDSL* wurde so konstruiert, dass keine zwei Aktions-Knoten hintereinander folgen dürfen, sondern immer abwechselnd Aktions- und Condition-Knoten vorkommen müssen. Daher kommt nun im Modell ein Condition-Knoten, welcher keinen Trigger enthält und quasi leer ist. Seine einzige Funktion ist einen Delay von 1 s sicherzustellen, damit die nächste Aktion nicht unmittelbar auf die vorherige folgt. Anschließend wird das 30 km/h fahrende NewCar 200 m vor Ego platziert. Da Ego noch seine eingestellten 50 km/h fährt und somit dem NewCar auffährt, löst der nächste Trigger im folgenden Condition-Knoten aus, wenn Ego auch 30 km/h fährt und somit dem NewCar mit eingestelltem Zeitabstand im Zustand „Following“ folgt. Erst danach beginnt der eigentliche Testfall. Das NewCar beschleunigt auf 70 km/h. Danach folgt nur noch eine leere Condition, in welcher 10 s gewartet wird, damit die Monitore die Nachbedingungen prüfen können. Denn sobald der End-Knoten erreicht wird, beendet sich das Szenario und somit der Testfall. Hier bestehen die Nachbedingungen daraus, dass das Ego-Fahrzeug von den 30 km/h bis zu seinen eingestellten 50 km/h mit dem NewCar beschleunigt, währenddessen jedoch nie den Zeitabstand von 1,0 s unterschreitet, die aus der *ACC-ISO-Norm* stammende maximale Beschleunigung von 2 m/s^2 nicht überschreitet und nicht weiter als bis 50 km/h beschleunigt. Außerdem wechselt Ego vom „Following“ Zustand in den „SpeedControl“ Zustand, sobald NewCar aus dem Zeitabstand verschwunden ist bzw. beschleunigt hat, da NewCar bis 70 km/h beschleunigt.

Generell wird bei den Vor- und Nachbedingungen normalerweise getestet, ob Zeitabstände und Geschwindigkeiten eingehalten werden und Zustandswechsel des ACCs stattfinden bzw. nicht stattfinden. Bei den Invarianten wird getestet, ob die maximale Beschleunigung von 2 m/s^2 und die maximale Verzögerung von 3,5 m/s^2 vom ACC nicht über- bzw. unterschritten wird.

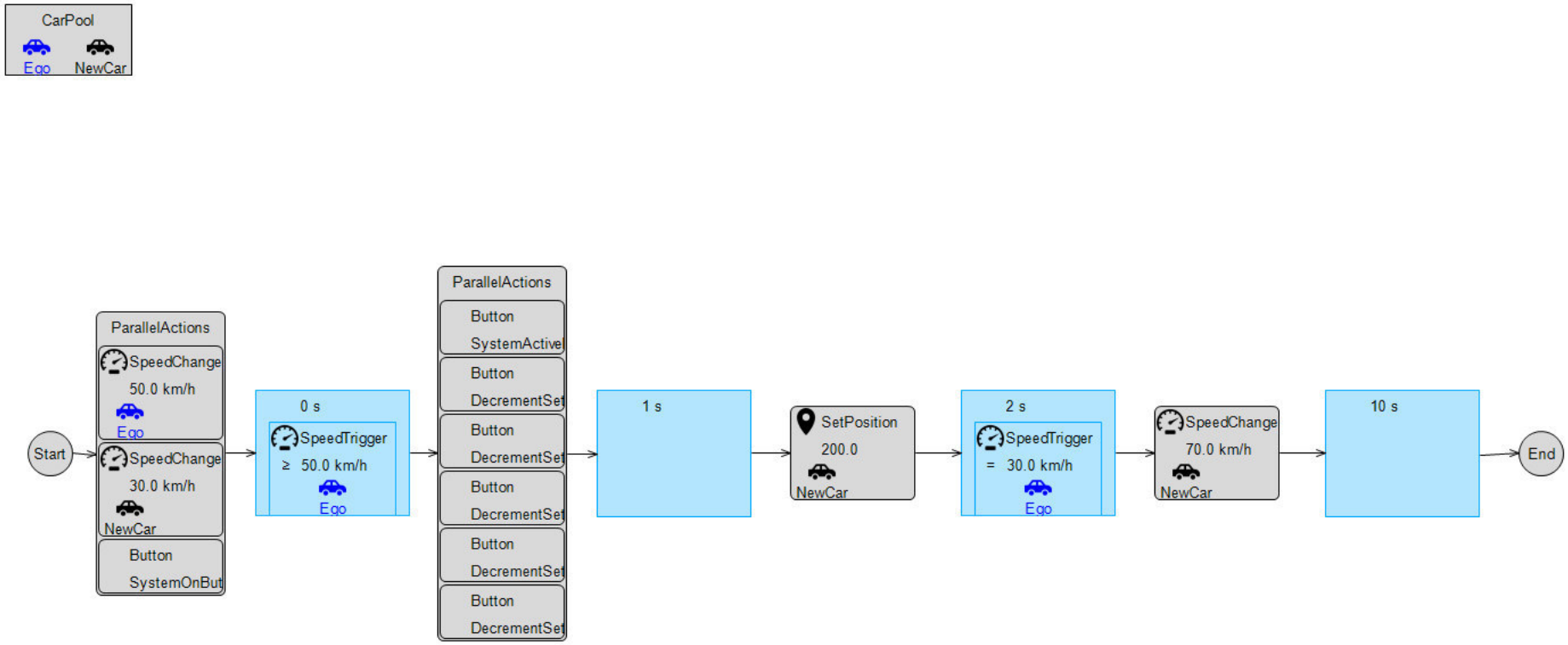


Abbildung 9.3: Beispielhaftes Modell eines Testfalls

Teil III

Technische Umsetzung

Kapitel 10

Umsetzung der AutoDSL

In diesem Abschnitt wird die tatsächliche Umsetzung des in Kapitel 7 Modelles beschrieben. Zu beachten ist hier, dass viele Entscheidungen aufgrund der technischen Limitierungen und dem generellen Aufbau von CINCO getroffen wurden, was rückwirkend natürlich auch einen Effekt auf die endgültige Wahl des theoretischen Modelles hat. So folgen die Wahl für die Bedingungen der Guards und die Entscheidung, für das Verhalten in Zuständen das gleiche Modell zu verwenden, aus der Tatsache, dass Vererbung bei Modellen in Cinco nicht für unsere Zwecke ausreicht. Außerdem hätte die Verwendung eines „Metamodelles“, von welchem Modelle für die beiden Anwendungen erben, zur Folge gehabt, dass viele Funktionen, wie Hooks, dupliziert hätten werden müssen. Da dies einen enormen Aufwand bei jeglichen Änderungen verursacht hätte, wurde die Entscheidung getroffen, für beide Anwendungen das gleiche Modell zu verwenden und über Checks sicherzustellen, dass aktuell keine verbotenen Operationen verwendet werden.

10.1 Zustandsmodell

In dem folgenden Unterkapitel wird erläutert, wie das Zustandsmodell als DSL umgesetzt wurde und wie mit dieser umgegangen werden kann.

10.1.1 Definition der entwickelten domänenspezifischen Sprache

Das Zustandsmodell wird als graphische DSL mit dem Namen „AutoDSL“. umgesetzt. Dazu wurde mit Hilfe von CINCO eine MGL erstellt, die den Entwurf aus Unterabschnitt 7.1.1 umsetzen soll.

```
@style(stateOff)
node OffState {
  incomingEdges (GuardToState[0,*])
  outgoingEdges (StateToGuard[1,*])
}
```

Code-Ausschnitt 10.2: Knotendefinitionen der AutoDSL in MGL

```
graphModel AutoDSL {

  @style(simpleArrow)
  edge SyscompToSyscomp {
  }

  @style(simpleArrow)
  edge StateToGuard {
  }

  @style(simpleArrow)
  edge GuardToState {
  }
}
```

Code-Ausschnitt 10.1: Kantendefinitionen der AutoDSL in MGL

In Code-Ausschnitt 10.1 werden die Kantentypen angegeben, die in der DSL verwendet werden. Insgesamt gibt es drei verschiedene Kantentypen. Alle Typen besitzen den „simpleArrow“ Style, welcher einen schwarzen Pfeil zwischen zwei Knoten darstellt.

Es gibt nun zum einen die Typen, welche zum Navigieren zwischen Zuständen und Guards benutzt werden, namentlich GuardToState und StateToGuard, sowie den SyscompToSyscomp Pfeiltypen, der zwischen Komponenten eines Zustands gezogen werden kann. Im Folgenden werden die zugehörigen Knoten erläutert.

In Code-Ausschnitt 10.2 sind die Definitionen der Knoten zu sehen. Als erstes wurden die erlaubten Kardinalitäten der Zustände angegeben. So befinden sich auf oberster Ebene ein OffState, sowie beliebig viele States und Guards. Der OffState stellt den ausgeschalteten Zustand dar, welche auch im Entwurf wiederzufinden ist. Der „stateOff“-Style gibt an, dass dieser als Kreis mit einem schwarzen Rand und grauem Hintergrund dargestellt wird (vgl. Abbildung 10.1). Da es mindestens einen Zustand geben soll, muss der OffState mindestens eine ausgehende Kante zu einem Guard besitzen. Weiterhin werden beliebig viele eingehende Kanten akzeptiert.

```
@style(guard, "${label}")
container Guard {
  attr EString as label := "Unnamed"
  containableElements(
    ComponentNode[1,1]
  )
  incomingEdges(StateToGuard[1,*])
  outgoingEdges(GuardToState[1,1])
}
```

Code-Ausschnitt 10.4: Knotendefinitionen der AutoDSL in MGL - Guard

```
@style(state, "${label}")
container State {
  attr EString as label := "Unnamed"
  containableElements(
    ComponentNode [0,*]
  )
  incomingEdges (GuardToState[1,*])
  outgoingEdges (StateToGuard[0,*])
}
```

Code-Ausschnitt 10.3: Knotendefinitionen der AutoDSL in MGL - State

Neben dem OffState wurde noch der Zustand, der für das in ihm zu definierende Verhalten als Wrapper dienen soll, als State definiert. Im Prinzip enthält der Zustandsknoten ein oder mehrere Elemente, die in der Rule-MGL spezifiziert worden sind, wie Code-Ausschnitt 10.3 zeigt. Weiterhin besitzt ein Zustand ein Label, welches eine Unterscheidung sowie Identifizierung von Zuständen vereinfachen soll. Dieses wird auch oben an dem Zustand angezeigt. Da ein State nur durch einen Guard erreicht werden kann, muss es mindestens eine eingehende Kante geben, die von einem Guard ausgeht. Generell sind Zustandsübergänge nur über Guards möglich. Daher werden nur StateToGuard und GuardToState als ausgehende bzw. eingehende Kanten akzeptiert. Im Style ist definiert, dass ein State als graues Rechteck dargestellt wird (Vgl. Abbildung 10.1).

Als letzter Knoten, der auf oberster Ebene platziert werden kann, gibt es noch den Guard Knoten. Dieser überprüft, ob ein Übergang in einen anderen Zustand erforderlich ist. Um diese Überprüfung auszuführen, werden wieder Rule-Elemente in dem Guard Knoten benötigt, nun aber mit der festen Kardinalität 1 (siehe Code-Ausschnitt 10.4). Als eingehende Knoten werden beliebig viele Kanten ausgehend eines States akzeptiert. Da es nicht möglich ist, dass mehrere Zustände gleichzeitig aktiv sind, darf es nur eine ausgehende

Kante zu einem State geben. Graphisch wird ein Guard als grünes Rechteck mit dem Text «GUARD» dargestellt. Weiterhin wird auch hier zur Identifizierung ein Label verwendet (vgl. Abbildung 10.1).

```
@postCreate("info.scce.cinco.product.autoDSL.hooks.DropRuleHook")
@doubleClickAction("info.scce.cinco.product.autoDSL.hooks.openRuleModel")
@style(syscomponent, "${label}")
node ComponentNode {
  @pvFileExtension("rule")
  prime rule::Rule as rule
  attr EString as label
  incomingEdges(SyscompToSyscomp[0,1])
  outgoingEdges(SyscompToSyscomp[0,1])
}
```

Code-Ausschnitt 10.5: Knotendefinitionen der AutoDSL in MGL

Zuletzt fehlt nur noch die Definition der Rule-Elemente. In der MGL werden diese als ComponentNode bezeichnet und sind im Code-Ausschnitt 10.5 zu sehen. Dem ComponentNode ist es nur erlaubt, jeweils maximal eine ausgehende und eingehende Kante zu anderen ComponentNodes zu haben. Außerdem besitzt der Knoten, wie auch Guard und State, ein Label, welches zur Identifikation verwendet wird. Da ComponentNodes auf Regeln verweisen, welche in einer eigenen MGL definiert sind, wird ein Verweis auf diese benötigt. Dieser ist als Import in dem Header-Teil der MGL zu finden. Ein solcher Import ist in Code-Ausschnitt 10.6 zu sehen. Weiterhin wird in der ComponentNode Knotendefinition die Rule-MGL referenziert. Damit der Editor weiß, welche Dateien zu der Rule-MGL gehören, wird definiert, dass diese die Dateiendung „rule“ besitzen. Außerdem ist bei dieser Definition zu beachten, dass es weitere Annotationen gibt. So gibt es neben der Style-Annotation noch zwei weitere – `postCreate` und `doubleClickAction`. Erstere setzt das Label der ComponentNode auf den Namen der verwendeten Regel und mit der zweiten Annotation lässt sich die Regel mit einem Doppelklick in einem eigenen Fenster öffnen. Der ComponentNode wurde bereits in Abbildung 10.1 gezeigt.

In dem Header werden eine Reihe von Annotationen angegeben. Die *Primeviewer* Annotation ermöglicht die Referenzierung von anderen DSLs. Hier wird dies für die Rule-DSL verwendet. Mit der Style-Annotation wird auf die zu verwendende Style-Datei verwiesen. Mit den mcam-Annotationen werden sogenannte Checks aktiviert, die zur Verifikation verwendet werden (vgl. dazu Abschnitt 10.3). Zuletzt wird noch der Generator registriert, der aus einem gegebenen Modell Code generiert. Dieser wird in Kapitel 11 genauer erläutert.

```
import "platform:/resource/info.scce.cinco.product.autoDSL/model/Rule.mgl" as rule

@primeviewer
@style("model/AutoDSL.style")
@mcam("check")
@mcam_checkmodule("info.scce.cinco.product.autoDSL.check.autoDSLCheck.
    CheckForInvalidRulesInGuards")
@mcam_checkmodule("info.scce.cinco.product.autoDSL.check.autoDSLCheck.
    CheckForUnreachableStates")
@mcam_checkmodule("info.scce.cinco.product.autoDSL.check.autoDSLCheck.
    CheckForInvalidRulesInStates")
@mcam_checkmodule("info.scce.cinco.product.autoDSL.check.autoDSLCheck.
    CheckForInvalidEdgesBetweenRules")
@generatable("info.scce.cinco.product.autoDSL.generator.DSLGenerator","/src-gen/")
graphModel AutoDSL {
...
...
...
}
```

Code-Ausschnitt 10.6: Header der AutoDSL in MGL

10.1.2 Ausführung

In diesem Teil wird erläutert, wie die Ausführung der AutoDSL zu verstehen ist. Als Beispiel, an dem dies erklärt wird, wird ein Abstandshalter-Modell verwendet, welches nach einmaligem Einschalten nicht mehr ausgeschaltet werden kann. Dieses ist in Abbildung 10.1 dargestellt.

Der Ablauf der Ausführung ist in zwei Schleifen unterteilt, die jeweils hintereinander ausgeführt werden. So gibt es die erste Schleife, die überprüft, ob ein Zustandswechsel nötig ist, also ob ein Guard aktiviert werden muss. Dies wird erreicht, indem in allen Guards, die mit dem aktuellen Zustand über eine eingehende Kante verbunden sind, die Rules evaluiert werden. Falls nun ein Guard aktiviert wird, wird in den Folgezustand gewechselt. Im obigen Beispiel wird also aus dem Offstate in den Abstandshalter-Zustand gewechselt, wenn der „Abstandshalter aktiviert?“-Guard greift.

Die zweite Schleife beschäftigt sich mit dem Verhalten innerhalb eines Zustands. In dieser Schleife werden hintereinander die Rule-Elemente überprüft und gegebenenfalls ausgeführt. Jede Rule hat dafür eine Überprüfung, ob sie im Moment aktiv sein soll. Im Beispiel wird so in jedem Durchlauf erst überprüft ob eine Notbremsung durchgeführt werden muss, falls sich ein Objekt zu schnell dem Auto nähert. Falls dies eintrifft, wird die Notbremsung durchgeführt und die Schleife erneut von Anfang ausgeführt. Lehnt die Notbremsungs-Rule allerdings ab, wird die nächste Regel in der Hierachie, im Beispiel die Abstands-Rule, überprüft und gegebenenfalls ausgeführt.

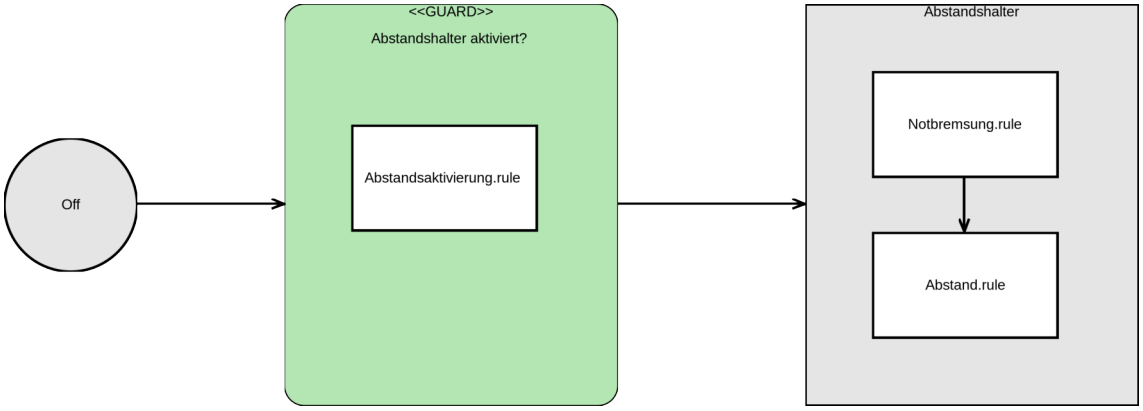


Abbildung 10.1: AutoDSL Beispiel

10.2 Rule-DSL

Dieses Modell entspricht dem in Unterabschnitt 7.1.2 vorgestellten Datenflussmodell. Da diese Modelle abhängig von den Eingabewerten (= Sensorwerte) die Handlungen des Systems bestimmen, werden sie Regeln, oder auch *Rules*, genannt. Da in diesem Modell im Vergleich zum Zustandsmodell eine Vielzahl an verschiedenen Operationen benötigt werden, sollen in diesem Abschnitt hauptsächlich die zu Grunde liegenden Kategorien betrachtet werden. Wie in Unterabschnitt 7.1.2 erwähnt, sollen folgende Funktionen bereitstehen:

- Arithmetische Operatoren
- Boolesche Operatoren
- Vergleichsoperatoren
- If-Then-Else-Strukturen
- Definition von statischen Werten zur späteren Verwendung
- Ein/Ausgabe von Fahrzeugwerten
- Verwendung in Guards
- Zwischenspeichern von Werten über mehrere Zyklen
- Verwendung von Submodellen, um sich so eigene Funktionen zu definieren
- Verwendung von C++-Ausdrücken für komplexe Formeln

Nach einer kurzen Präsentation des Metamodelles, welches die Basis bildet, wird in diesem Abschnitt auf die Umsetzung der geforderten Funktionalitäten eingegangen.

10.2.1 Basis

Das zu Grunde liegende Modell wird in `Rule.mgl` definiert. Ein Rule-Modell hat eine beliebige Anzahl an Operation-Knoten, welche über ControlFlow-Kanten miteinander verbunden werden. Normale Kontrollflusskanten haben den Typ ControlFlowBasic, dadurch können sie später von besonderen Kanten unterschieden werden. Diese Operation-Knoten sind Container und enthalten Input- und Output-Knoten vom abstrakten Typ IO. Sie modellieren die Schnittstellen, über die Operationen Werte mit anderen Operationen teilen. Die beiden verwendbaren Datentypen sind Zahlen (Number) und Wahrheitswerte (Boolean), deshalb werden Input- und Output-Knoten entsprechend der beiden Datentypen unterteilt (vgl. Code-Ausschnitt 10.7). Dies ermöglicht es später, bei der Definition der Operationen festzulegen, welche Arten von Werten wo verwendet werden können. IO-Knoten können entsprechend ihrer Datentypen mit NumberDataFlow und BooleanDataFlow-Kanten verbunden werden. Durch diese Unterscheidung ist es möglich sicher zu stellen, dass Zahlen nicht als Wahrheitswerte verwendet werden können (und umgekehrt). Um mögliche Fehler zu verhindern, wurden außerdem explizite Start- und Endknoten ergänzt. Diese sind einfache Knoten, welche mit Kontrollflusskanten mit den Operationen verbunden werden können. Ein Modell hat genau einen Startknoten und möglicherweise mehrere Endknoten (zum Beispiel durch Abzweigungen bei If-Then-Else-Operationen). In einer vorherigen Version wurden Start und Ende implizit durch das Fehlen von ein- und ausgehenden Kontrollflusskanten in den Operationen definiert, dies hatte allerdings zur Folge, dass nicht immer auf einen Blick klar war, wo der Kontrollfluss beginnt. Die neuen Knoten bilden einen deutlichen optischen Kontrast und können auch in einem sehr komplexen Modell schnell erkannt werden.

```
1 abstract node IO {}
2 abstract node Input extends IO {}
3 abstract node BooleanInput extends Input {}
4 abstract node NumberInput extends Input {}
5 abstract node Output extends IO {}
6 abstract node BooleanOutput extends Output {}
7 abstract node NumberOutput extends Output {}
```

Code-Ausschnitt 10.7: Input- und Output-Typen in Rule.mgl

10.2.2 Arithmetische, Boolesche und Vergleichsoperationen

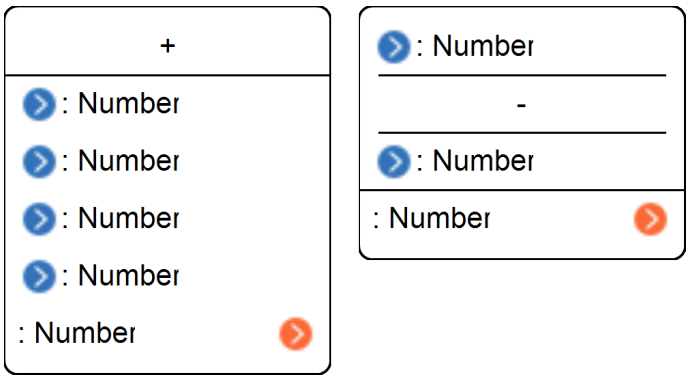


Abbildung 10.2: Eine kommutative und eine nicht-kommutative Operation

Folgende Operationen werden aktuell unterstützt: $+$, $-$, $\times$, $/$, x^y, $<$, $>$, $\leq$, $\geq$, max, min, *AND*, *OR* und *NOT*. In Code-Ausschnitt 10.8 ist das Beispiel der Addition zu sehen. Zu sehen ist hier zunächst die style-Zuweisung, danach werden der postCreate-Hook und postResize-Hook gesetzt. Diese sorgen dafür, dass Operationen mit der korrekten Anzahl an Inputs erzeugt werden und dass die Inputs, sollte die Größe der Operation verändert werden, mit vergrößert/verkleinert werden. Außerdem wird dort sichergestellt, dass sie an der korrekten Position bleiben. In der eigentlichen Definition werden dann die enthaltbaren Knotentypen festgelegt: Kein BooleanInput, ein oder mehrere NumberInputs, kein BooleanOutput und exakt ein NumberOutput. Die Additionsoperation hat also beliebig viele Zahlen als Eingabe und gibt genau eine Zahl aus. Die anderen Operationen sind analog definiert. In Abbildung 10.2 sind zwei Operationen zu sehen, hierbei handelt es sich um die Addition und die Subtraktion. Da die Addition kommutativ ist, also $a+b+c = c+a+b = \ldots$, kann sie beliebig viele Inputs besitzen. Im Gegenteil dazu steht die Subtraktion, welche nur eine feste Anzahl an Inputs besitzen kann. Hiernach bietet es sich also an, Operationen in nicht-kommutative und kommutative Operationen aufzuteilen, welche sich somit auch im style unterscheiden.

```
1 @style(commutableOperation, "+")
2 @postCreate("info.scce.cinco.product.autoDSL.hooks.CreateOperation")
3 @postResize("info.scce.cinco.product.autoDSL.hooks.ResizeOperation")
4 container Addition extends CommutableOperation {
5   containableElements(BooleanInput[0,0], NumberInput[1,*],
6                       BooleanOutput[0,0], NumberOutput[1,1])
7 }
```

Code-Ausschnitt 10.8: Beispiel einer Operation (Addition) aus Rule.mgl

10.2.3 IF-THEN-ELSE

Um IF-THEN-ELSE-Strukturen zu modellieren, wurde der Knoten Decision ergänzt. Bei diesem handelt es sich um eine Operation, welche lediglich einen Boolean-Wert als Input erhält und keine Daten ausgibt, sondern zwei ausgehende ControlFlow-Kanten besitzt. Um die beiden ausgehenden Kanten später zu unterscheiden, wurden die beiden Kantentypen ControlFlowDecisionTrue und ControlFlowDecisionFalse erstellt (vgl. Code-Ausschnitt 10.9). Diese verhalten sich wie normale ControlFlow-Kanten, können aber unterschieden werden. Dadurch ist es möglich den Decision-Knoten so zu definieren, dass er genau eine Kontrollflusskante für jedes mögliche Ergebnis besitzt. Zusätzlich haben die beiden Kanten über entsprechende styles passende Beschriftungen erhalten (siehe Abbildung 10.3).

```
1 @style(commutableOperation, "?")
2 @postCreate("info.scce.cinco.product.autoDSL.hooks.CreateOperation")
3 @postResize("info.scce.cinco.product.autoDSL.hooks.ResizeOperation")
4 container Decision extends Operation{
5   outgoingEdges(ControlFlowDecisionFalse[1,1], ControlFlowDecisionTrue[1,1],
      ControlFlowBasic[0,0])
6   containableElements(BooleanInput[1,1], NumberInput[0,0], BooleanOutput[0,0
      ], NumberOutput[0,0])
7 }
```

Code-Ausschnitt 10.9: `Decision`-Knoten aus `Rule.mgl`

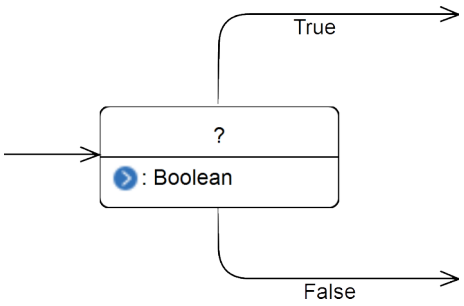


Abbildung 10.3: Decision-Knoten aus dem Rule-Modell

10.2.4 Statische- und Fahrzeugwerte

Um zwischen statischen Werten, Fahrzeugwerten, und normalen offenen Input/Output-Schnittstellen zu unterscheiden, wurden entsprechende Knotentypen erstellt. Stellt man die Vererbungsstruktur in einem Baum dar, so sieht sie wie folgt aus:

IO (`abstract`)
    Input (`abstract`)
        NumberInput (`abstract`)

```
@style(inputPort, "Number")
@disable(move,resize)
@palette("Ports")
@icon("icons/inputPort.png")
@contextMenuAction("info.scce.cinco.product.autoDSL.hooks.ToStatic")
@contextMenuAction("info.scce.cinco.product.autoDSL.hooks.ToCar")
@doubleClickAction("info.scce.cinco.product.autoDSL.hooks.ToCar")
@postCreate("info.scce.cinco.product.autoDSL.hooks.CreateIO")
@preDelete("info.scce.cinco.product.autoDSL.hooks.DeleteIO")
node NumberInputPort extends NumberInput {
  incomingEdges(NumberDataFlow[1,1])
}
```

Code-Ausschnitt 10.10: Ein beispielhafter `IO`-Typ aus `Rule.mgl`

- NumberCarInput
- NumberInputPort
- NumberStaticInput
- BooleanInput (`abstract`)
  - BooleanCarInput
  - BooleanInputPort
  - BooleanStaticInput
- Output (`abstract`)
  - NumberOutput (`abstract`)
    - NumberCarOutput
    - NumberOutputPort
  - BooleanOutput (`abstract`)
    - BooleanCarOutput
    - BooleanOutputPort

Standardmäßig werden alle Operationen mit InputPort und OutputPort-Knoten erstellt. In Code-Ausschnitt 10.10 ist ein Beispiel eines solchen spezialisierten Porttypen zu sehen. Zu beachten sind hier die Hooks zur Umwandlung in andere Porttypen, welche es ermöglichen, Ports durch Doppelklicks oder über Rechtsklick → `Convert to ...` in andere Porttypen umzuwandeln. Statische Inputs besitzen zusätzlich noch ein Attribut, über welches der Nutzer den Wert setzen kann. Fahrzeugwerte besitzen einen Enum-Wert, über den der Nutzer über ein Dropdown-Menü den gewünschten Wert auswählen kann. In der Abbildung 10.4 ist eine Operation mit den drei verschiedenen Input-Typen zu sehen.

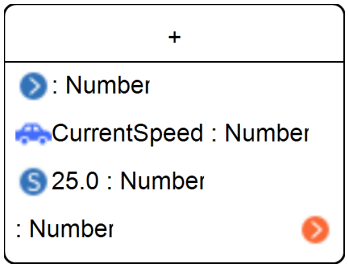


Abbildung 10.4: Eine Addition mit den drei Input-Typen

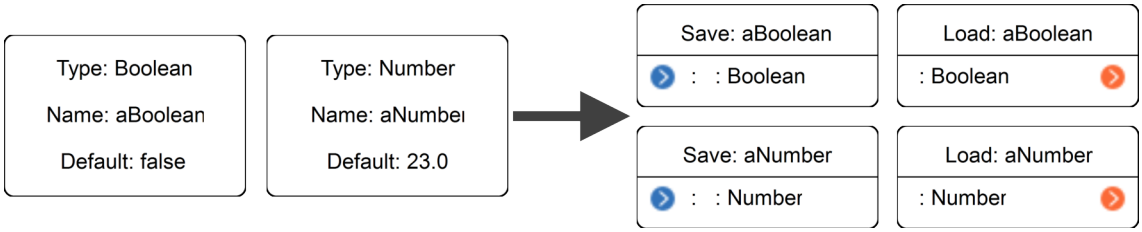


Abbildung 10.5: Ein Beispiel für die Verwendung von `SharedMemory`-Elementen

10.2.5 Verwendung in Guards

Um eine `Rule` als Bedingung in einem Guard zu verwenden, fehlt aktuell die Möglichkeit, einen Boolean an ihn auszugeben. Dazu wurde ein Knoten BooleanGuardOutput vom Typ BooleanOutput ergänzt, welcher als Eingabe einen Boolean-Wert erhält und welcher keine ausgehende Kontrollflusskante hat. Nun kann dieser wie ein entsprechender BooleanCarOutput verwendet werden, nur dass hier der Wert nicht an das Fahrzeug übergeben wird, sondern an den entsprechenden Guard, welcher die `Rule` enthält.

10.2.6 Zwischenspeichern von Werten

Um dem Nutzer zu ermöglichen, beliebige Werte über mehrere Ausführungszyklen und verschiedene `Rule`-Modelle zu speichern und abzurufen, wurde ein einfacher Modelltyp ergänzt. Dieser Modelltyp, genannt `SharedMemory`, enthält einfache Knoten vom Typ `StoredBoolean` und `StoredNumber`, die nur einen Bezeichner und einen `defaultValue` besitzen. Diese Knoten können dann per *Drag & Drop* aus dem *Project Explorer* einfach vom Nutzer in beliebige `Rule`-Modelle gezogen werden. Der Nutzer kann dann auswählen, ob der Wert geladen oder gespeichert werden soll. Nach der Auswahl wird eine simple Operation mit nur einem `Input` oder einem `Output`, je nach Auswahl, erzeugt, die dann wie andere Operationen verwendet werden kann. in Abbildung 10.5 ist zu sehen, wie die entstehenden Operationen (rechts) aussehen, und wie die Knoten im `sharedMemory`-Modell (links) aussehen.

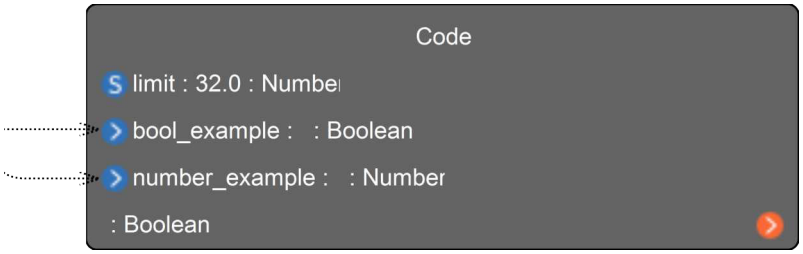


Abbildung 10.6: Eine frei programmierbare Code-Operation

10.2.7 Submodelle

Es kann vorkommen, dass innerhalb verschiedener `Rule`-Modelle ähnliche oder gar identische Strukturen auftreten. Um die Verwendung zu vereinfachen, indem redundante Umsetzungen erspart bleiben, wurde die hierarchische Nutzung von Submodellen ermöglicht. Eine solche `SubRule` kann dabei verschiedene Gestalten annehmen. Im einfachsten Fall entspricht sie einem herkömmlichen `Rule`-Modell und bildet einen in einem oder mehreren Modellen wiederkehrenden Ablauf ab. Es gibt jedoch Situationen, in denen Parametrisierung von Interesse ist oder die Möglichkeit, innerhalb der `SubRule` angestellte Berechnungen später zu nutzen. Für die Umsetzung entsprechender Variabilität wurden neue Knoten für jeweils SubRuleInputs und SubRuleOutputs ergänzt. Eine solche `SubRule`, die auf diese Modellelemente zurückgreift, kann in Folge allerdings nicht mehr sinnvoll eigenständig verwendet werden, da sie nun von den Parameterwerten abhängig ist, die so nur in einer Defaultbelegung vorliegen. Damit ein Nutzer `Rule`-Modelle als `SubRule` einsetzen kann, muss er diese lediglich mittels *Drag&Drop* aus dem *Project Explorer* ziehen. Wurden im Untermodell Ein- und Ausgabewerte festgelegt, werden diese in Folge als `In-` bzw. `Outputs` angezeigt.

10.2.8 Verwendung von beliebigen C++-Ausdrücken

Ein weiterer Wunsch war es, beliebigen `C++`-Code in Operationen zu verwenden, um zum Beispiel komplexere Gleichungen kompakt zu verwenden. Hierfür wurde die Operation `ProgrammableNode` ergänzt. Sie hat ein Attribut `code`, in welchem der Nutzer beliebigen Code eintragen kann. Als Eingabeports für diese Operation stehen besondere Ports bereit, welche ein `identifier`-Attribut besitzen, mit welchem der Name der Ports gesetzt werden kann. Diese Bezeichner können dann im `code`-Attribut als Variablen verwendet werden (vgl. Abbildung 10.6). Eine `ProgrammableNode`-Operation kann dann entweder einen `BooleanOutput` oder einen `NumberOutput` besitzen. In dem Fall muss im `C++`-Code ein entsprechender `return`-Befehl verwendet werden.

10.3 Modellverifikation

Zusätzlich zu den durch die MGL erzeugten automatischen Überprüfungen der Modelle auf die korrekte Anzahl an Kanten oder enthaltenen Knoten, wurden weitere Überprüfungen (*Checks*) ergänzt. In diesem Abschnitt werden die durch Checks feststellbaren möglichen Fehler in den Modellen präsentiert.

10.3.1 AutoDSL

Zur Erinnerung: das AutoDSL-Modell ist das zentrale Zustandsmodell, welches steuert, in welchem Zustand sich das Assistenzsystem aktuell befindet. Folgende Problemursachen wurden bisher identifiziert:

- Unerreichbare Zustände:
  Es ist theoretisch möglich ein Modell zu erstellen, welches alle Anforderungen an notwendige Kanten und Knotenmengen erfüllt, aber dennoch unerreichbare Zustände enthält. Ein gutes Beispiel ist exemplarisch in Abbildung 10.7 zu sehen. Zu beachten ist hier, dass die beiden rechten Zustände Unreachable A und Unreachable B nicht vom Startzustand Off erreicht werden können.

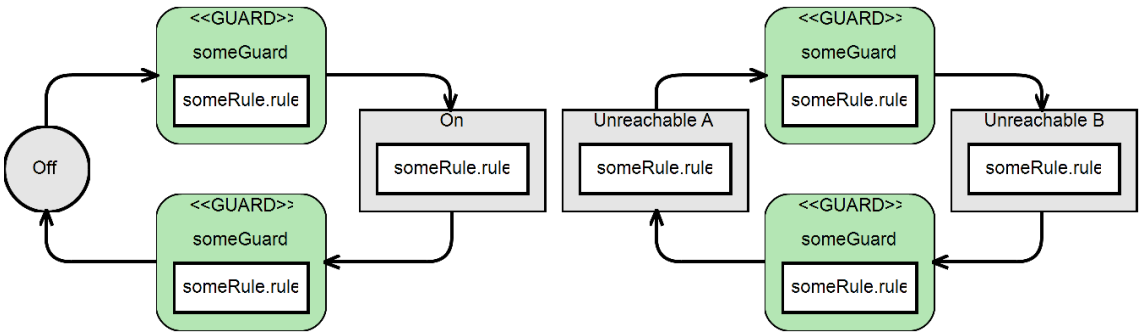


Abbildung 10.7: Ein AutoDSL-Modell mit unerreichbaren States

- Korrektheit der Rules in Guards:
  Es muss sichergestellt werden, dass in den Rules, welche als Bedingung für einen Guard verwendet werden, keine Werte an das Auto gesendet werden, da sonst bei jeder Überprüfung eines Guards diese Werte versendet werden würden. Außerdem muss mindestens ein BooleanGuardOutput vorhanden sein, da diese verwendet werden, um zu bestimmen, ob ein Zustandswechsel durchgeführt wird.

- Korrektheit der Rules in States:
  Es muss sichergestellt werden, dass Rules in einem State keine BooleanGuardOutputs enthalten, diese sind ausschließlich für die Verwendung in Guards vorgesehen. Sollte ein BooleanGuardOutput vorhanden sein, würde dies zu einem return im generierten Code führen, was ein unerwartetes Verhalten zur Folge hätte.

- Ungültige Kanten zwischen Rules:
  Rules in States können mit Kanten verbunden werden, um so mehrere Rules hintereinander durchzuführen und zu ordnen. Hier muss sichergestellt werden, dass nur Rules innerhalb des selben States miteinander verbunden werden und dass keine Zyklen entstehen.

- Ungültige Bezeichner für States:
  Um beim Monitoring später auf die verschiedenen Zustände zu referenzieren, werden die Bezeichner, also das label, verwendet. Um sicherzustellen, dass keine Fehler durch in Xtext ungültige Bezeichner entstehen, wird überprüft ob sie:

  – Nicht leer sind.

  – Nicht ihren Bezeichner mit einem anderen State teilen.

  – Nicht mit Zahlen beginnen

  – Keine Leerzeichen enthalten

  – Nur alphanumerische Zeichen oder _ enthalten

  Die gleiche Überprüfung findet analog bei SharedMemory-Modellen für die gespeicherten Werte und ihre Bezeichner statt.

10.3.2 Rule

Rule-Modelle werden verwendet, um den Datenfluss zu modellieren. Sie werden einerseits verwendet, um die Bedingungen für Zustandsübergänge zu modellieren und andererseits, um die Handlungen in einem Zustand zu modellieren. Dadurch entstehen einige Probleme, welche beachtet werden müssen:

- Existenz eines klaren Startknotens:
  In dem Modell ist vorgesehen, dass exakt eine Operation keinen Vorgänger besitzt. Diese Operation ist dann der Startknoten. Es muss also überprüft werden ob dies zutrifft. Sollte dies nicht der Fall sein, ist nicht klar, wo das generierte Programm beginnen soll.

- Unerreichbare Operationen:
  Ähnlich wie in den AutoDSL-Modellen sollte überprüft werden, ob es Operationen gibt, welche vom Startknoten nicht erreichbar sind. Ein Modell mit unerreichbaren Operationen kann zwar theoretisch korrekt sein, aber die Vermutung liegt nahe, dass unerreichbare Knoten keine beabsichtigten Entwurfsentscheidungen sind.

- Zyklenfreiheit:
  Da vorgesehen ist, dass jede Rule genau einem Schritt der Steuerung des Fahrassis-

tenzsystems entspricht und somit die Steuerung auf die Ergebnisse der Rule wartet, wären Endlosschleifen fatal für das System. Deshalb muss sichergestellt werden, dass Rule-Modelle keine Zyklen enthalten. Da es nicht immer möglich ist zu bestimmen, ob ein Zyklus terminiert, sind Zyklen und Rückwärtskanten jeglicher Art verboten.

- Korrektheit der Reihenfolge:
  Es muss sichergestellt werden, dass keine Werte als Eingaben verwendet werden, welche zu dem Zeitpunkt noch nicht berechnet wurden. Also dürfen nur Werte von Operation-Knoten verwendet werden, welche im Kontrollfluss vor dem aktuellen Knoten liegen.

- Korrektheit der hierarchischen Gliederung:
  Es ist notwendig bei der Verwendung von Rule-Modellen als SubRule darauf zu achten, dass die gewählte Platzierung in der Hierarchie regelkonform ist. Ein Modell mit BooleanGuardOutput-Knoten kann beispielsweise nur in dem Fall als Submodell genutzt werden, wenn es das letzte Element im Modell darstellt, da nach dem BooleanGuardOutput selbst keine weiteren Elemente zulässig sind. Diese Eigenschaft wird durch die gesamte Hierarchie propagiert. Außerdem darf für die Verwendung als Guard auf keiner noch so tiefen Subebene eine Manipulation des Fahrzeugs initiiert werden. CarOutputs jeglicher Art sind in diesem Kontext daher nicht zulässig.

- Keine Ambiguität der Rückgabe:
  Sowohl im Falle des Einsatzes als Guard mit BooleanGuardOutput, als auch als SubRule mit SubRuleOutputs muss gewährleistet werden, dass nach einer Verzweigung im Rahmen einer Entscheidung jeder einzelne Pfad auf dem entsprechenden Rückgabeknoten endet.

Kapitel 11

Generator

In diesem Abschnitt wird der entwickelte Generator vorgestellt, der aus den Modellen der AutoDSL ausführbaren C++-Code generiert. Der Generator wurde in XTEND (s. Kapitel 4) implementiert und lässt sich aus einem Zustandsmodell der AutoDSL heraus anstoßen. Insgesamt behandelt der Generator drei Arten von Klassen:

- die Strukturen, die unabhängig von den Modellen sind,
- die Klasse für das Zustandsdiagramm und
- die Klassen für die einzelnen Datenflussmodelle.

Wie diese generiert werden, ist in den folgenden Abschnitten erläutert.

11.1 Modellunabhängige Strukturen

Die modellunabhängigen Strukturen sind hauptsächlich Interfaces und Klassen, die von den generierten Klassen implementiert werden bzw. von denen sie erben. Das Interface StateRule soll von allen Klassen implementiert werden, die aus einem Datenflussmodell generiert werden und mindestens einen CarOutput besitzen. Dabei zählt es auch, wenn der CarOutput sich in einer SubRule befindet. Das Interface schreibt die folgenden Funktionen vor:

void onEntry — Eine Methode, die während einer Zustandstransition auf dem Folgezustand aufgerufen wird.

void Execute(const CarInputs&, CarOutputs&) — Eine Methode, die auf dem aktuellen Zustand in einer Endlosschleife wiederholt aufgerufen wird.

void onExit — Eine Methode, die während einer Zustandstransition auf dem Ausgangszustand aufgerufen wird.

Die Methoden onEntry und onExit können dazu verwendet werden, Daten zu sichern, die bei einem Zustandsübergang verloren gehen würden. Ähnliche Interfaces stehen mit GuardRule und NeutralRule auch für die Datenflussmodelle bereit, in denen sich keine CarOutputs sondern ein GuardOutput bzw. keiner der beiden Outputtypen befindet. Dabei ändert sich jeweils die Signatur der Execute-Funktion. Bei einer GuardRule sind die einzigen Parameter die CarInputs, dafür wird ein Bool als Rückgabetyp erwartet. Bei der NeutralRule gibt es keinen Rückgabetyp und es werden nur die Inputs als Parameter erwartet.

Die Klasse State ist eine Implementierung des Interfaces StateRule, welche einen Vektor von StateRule-Instanzen verwaltet. Die Methoden des Interfaces sind dabei so umgesetzt, dass sie nacheinander auf den Elementen der Liste ausgeführt werden. Analog dazu gibt es die Klasse Guard, welche das Interface GuardRule implementiert und einen Vektor von diesen verwaltet.

Außerdem gibt es noch die Klasse StateMachine, in welcher die generierten Zustände und Transitionen verwaltet werden sollen. Die wichtigste Methode hierbei ist Run. Wird diese aufgerufen, so wird erst die Execute-Methode des aktuellen Zustands ausgeführt, anschließend werden die Bedingungen der anliegenden Transitionen geprüft und gegebenenfalls ein Zustandswechsel durchgeführt.

Zusätzlich dazu befinden sich in der Datei IO.h die beiden Structs CarInputs und CarOutputs. In diesen gibt es für jeden CarInput bzw. -Output eine Variable, damit alle Zustände leicht auf alle benötigten Daten zugreifen können.

Weiter gibt es auch Klassen, die verschiedene Funktionen bereitstellen. Diese Funktionen sind einerseits die Berechnung eines Minimums bzw. Maximums eines Arrays, andererseits Funktionen zum Loggen verschiedener Werte in einen Ringpuffer oder in log-Dateien.

11.2 Zustandmodellbezogene Klasse

Für den Zustandsautomaten wird eine Klasse generiert, die von StateMachine erbt. Der Name dieser Klasse ergibt sich aus dem Dateinamen des Modells. Diese legt für jeden State eine Klassenvariable vom Typ State und für jeden Guard eine vom Typ Guard an. Im Konstruktor wird der Vektor jeder Variablen mit den aus den Rules generierten Klassen gefüllt.

Die mindestens einmal behandelten Datenflussmodelle werden vom Generator in einer Liste verwaltet, damit es nicht zu Komplikationen oder Verzögerungen durch das mehrfache Generieren einer Klasse kommt, falls die zugehörige Rule mehrfach verwendet wird. Außerdem wird für jeden Zustandsübergang der Ausgangszustand, der zu erfüllende Guard und der

Zielzustand gespeichert. Am Ende des Konstruktors wird der Off-Zustand als Startzustand verwendet.

11.3 Datenflussmodellbezogene Klassen

Für jedes generierte Datenflussmodell wird eine neue Klasse mit einem Namen entsprechend des Dateinamens der „.rule“-Datei angelegt. Diese Klassen implementieren eines der in Abschnitt 11.1 genannten Interfaces StateRule, GuardRule oder NeutralRule.

In diesen Klassen gibt es für jede im Modell referenzierte SubRule eine Instanz der dafür generierten Klasse. Außerdem wird beim Generieren für jeweils die erste Referenz auf ein SharedMemory Modell ein Struct angelegt, dessen Name sich aus dem dazugehörigen Modell ergibt.

Momentan ist in den generierten Klassen Execute die einzige Funktion aus dem implementierten Interface, die mit Inhalt gefüllt ist. Beim Generieren der Execute-Methode wird zunächst nach einem Startknoten gesucht, welcher den Einstiegspunkt des Generators markiert. Beginnend mit dem Startknoten wird immer je nach Typ des Knotens entsprechend Quellcode hinzugefügt:

Rechenoperationen	Die Inputs werden aufgelöst und mit dem entsprechenden Rechenoperanden verknüpft.
Vergleiche	Die beiden Inputs werden aufgelöst und mit dem entsprechenden Vergleichsoperanden verknüpft.
max/min	Die Inputs werden in einem Array zwischengespeichert und mittels einer Sonderfunktion wird das Maximum/Minimum bestimmt.
logische Operationen	Die Inputs werden aufgelöst und mit dem entsprechenden logischen Operanden verknüpft.
DirectOutput	Der Input wird aufgelöst und sein Wert in der IO-Klasse in der dem Output entsprechenden Variablen abgespeichert.
GuardOutput	Der Input wird aufgelöst und sein Wert in einer vorab angelegten globalen Variable guard gespeichert.
Entscheidungen	Der Input wird aufgelöst und in der Bedingung einer **if**-Anweisung verwendet. Im **then**-Block wird die Generierung des Codes fortgeführt startend mit dem Knoten, der nach der **true**-Kante folgt. Anschließend wird im **else**-Block die Generierung des Codes fortgeführt startend mit dem Knoten, der nach der **false**-Kante folgt.

SubRuleOutput Die Inputs werden in Klassenvariablen der entsprechenden StateRule, GuardRule oder NeutralRule Instanz gespeichert und können beim Referenzieren von ihnen ausgelesen werden.

SubRule Die Inputs werden in Klassenvariablen der SubRule-Instanz geschrieben. Anschließend wird die Execute-Funktion dieser Instanz ausgeführt.

SharedMemorySave Der Input wird in die entsprechende Variable in einem SharedMemory-Struct geschrieben.

Programmierbare Knoten Es wird eine Funktion erstellt, dessen Signatur den Inputs und dem Output des Knotens entspricht. Dabei werden die angegebenen Identifier für die Parameter entsprechenden der Funktion verwendet. Der geschriebene Code wird als Implementierung der Funktion einfach übernommen. Die Methode wird aufgerufen, wenn mit dem Kontrollfluss der entsprechende Knoten erreicht wird.

Alle Knoten, mit Ausnahme des Entscheidungsknoten, haben nur eine ausgehende Kontrollflusskante. Falls diese vorhanden ist, wird die Codegenerierung mit dem Folgeknoten fortgesetzt.

Auch haben die meisten Knoten genau einen Output, der später weiterverwendet werden kann. Für diesen wird eine neue Variable des entsprechenden Typs angelegt, deren Name immer ein Unterstrich gefolgt von einem Hashwert der ID des Outputs aus dem Modell ist. Es wird ein Hashwert verwendet, weil die ID selbst Zeichen beinhalten kann, die für einen Variablennamen unzulässig sind, wie zum Beispiel „-“.

Das Auflösen der Inputs funktioniert anhand einer Fallunterscheidung:

- bei einem statischen Input wird einfach der entsprechende Wert verwendet,
- bei einem CarInput wird die zugehörige Variable aus der IO-Klasse verwendet und
- bei anderen Inputs wird zurückverfolgt, was der Input referenziert, und die entsprechende Output-Variable wird verwendet. Dabei wird bei Referenzen aus dem SharedMemory auf Variablen des entsprechenden Structs und bei SubRules auf die entsprechende Klassenvariable zugegriffen. Ansonsten wird die referenzierte Variable über den Hashwert der ID des über eine Datenflusskante verbundenen Outputports bestimmt.

Dabei ist besonders darauf zu achten, dass die referenzierten Variablen genau zu den Kanten im Datenflussmodell passen.

Kapitel 12

Umsetzung in ADTF

In diesem Kapitel werden selbstentwickelte Filter, wie beispielsweise der Lidar-Distance-Filter (Unterabschnitt 12.1.3) erläutert, die zur Umsetzung der zu modellierenden Logik von Nöten sind. Darauf folgend wird das Konzept der abstrakten Schicht zur Hardwareplattform in Form der *Generated-ADAS-API* vorgestellt, die als Grundlage für generierte Assistenzsysteme dient. Die zusammenwirkenden Filter in Form von verschiedenen ADTF-Konfigurationen werden im Anschluss daran beschrieben.

12.1 ADTF-Filter

ADTF-Filter werden durch Plugins bereitgestellt. Filter können über Input-Ports Eingabedaten erhalten und über Output-Ports Daten bereitstellen. Stellt ein Filter gleichzeitig Input- und Output-Ports zur Verfügung, erfolgt die Datenverarbeitung nach dem EVA-Prinzip (Eingabe, Verarbeitung und Ausgabe). Die Input-Ports eines Filters befinden sich links im visualisierten Filterknoten, die Output-Ports befinden sich rechts. Die Datenausgabe kann direkt nach erfolgter Berechnung nach Dateneingabe erfolgen. Dies lässt sich durch bereitgestellte Methoden hin zu einer getakteten Weiterleitung der Daten erweitern. Die Entwicklung eigener ADTF-Plugins erfolgt mithilfe des ADTF-SDKs. Die Struktur der Filter wird über Vererbung von Filter-Prototypen vorgegeben. Als Programmiersprache sieht das SDK C++ vor. Die verwendeten und bei einer Basisinstallation mitgelieferten ADTF-Plugins werden in Anhang B erläutert. Einige dieser Filter wurden im Rahmen des *Audi Autonomous Driving Cup (AADC)*[1] entwickelt.

Die im Rahmen der Projektgruppe entwickelten Filter lassen sich mit den ADTF-Filtern zusammen verwenden und implementieren jeweils spezielle Funktionalitäten.

[1] https://www.audi-autonomous-driving-cup.com/

12.1.1 Rplidar-Filter

Abbildung 12.1: Lidar-Filter

Der von BFFT bereitgestellte Lidar-Filter steuert das am Fahrzeug verbaute Lidar an und liefert der ADTF-Umgebung eine Liste von Punkten, die erkannte Hindernisse repräsentieren. Dabei ist der Nullpunkt des kartesischen Koordinatensystems der Punkte an der Fahrzeugfront angesetzt (siehe auch Abbildung 12.9). Mithilfe der Punktkoordinaten lässt sich so der Abstand des Punktes zur Fahrzeugfront in mm ermitteln. Die Punkte lassen sich wie in Abbildung 12.2 als Punktwolke um das Fahrzeug herum visualisieren. Dazu kann das in ADTF integrierte 2D-Display-Plugin genutzt werden. Der Filter liefert nicht einen einzigen Abstandswert, sondern je nach Baudrate etwa 300 - 400.

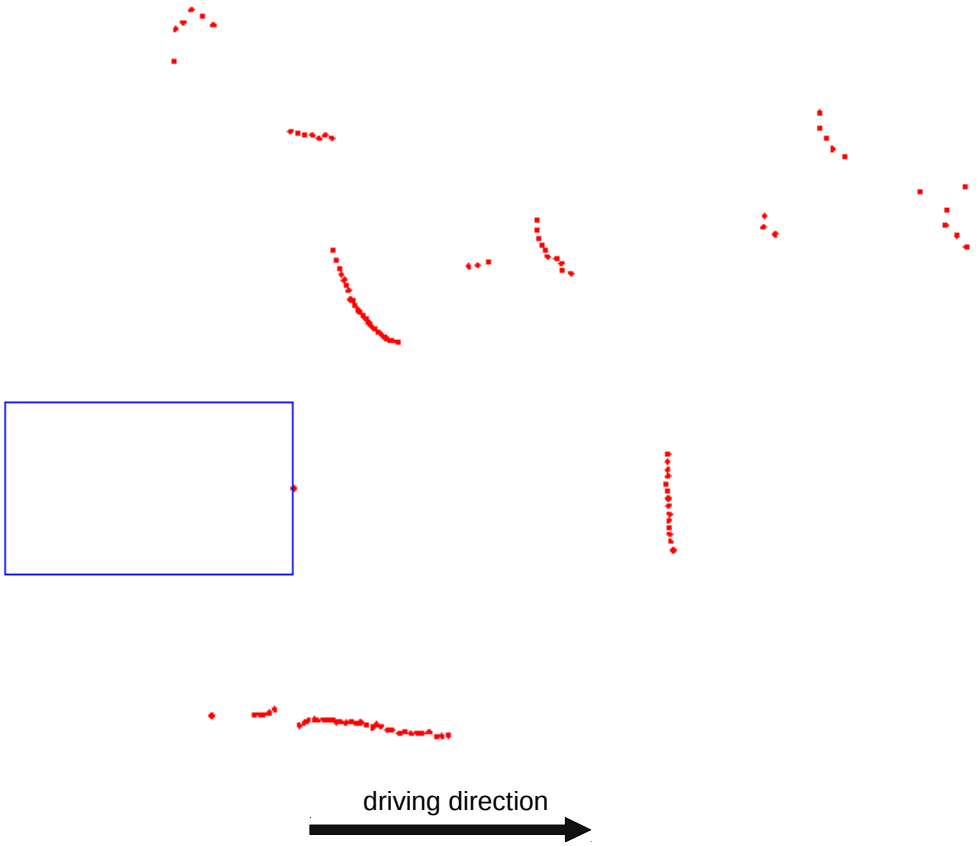


Abbildung 12.2: Visualisierung der Punktwolke. Der blaue Kasten stellt das Fahrzeug in Fahrrichtung nach Osten dar. Die roten Punkte sind die vom Lidar erkannten Hindernisse.

Die erkannte Punktwolke ist zweidimensional, da das Lidar nur Hindernisse auf Ebene des Lasers detektiert. Nicht auf dieser Ebene befindlichen Hindernisse können deshalb bauartbedingt nicht erkannt werden. Da das Lidar an der Fahrzeugfront auf Achsebene verbaut ist, sind von dem 360°-Erkennungsumfeld nur etwa 200° sinnvoll auswertbar. Die übrigen nicht verwendbaren 160° sind dem Umstand geschuldet, dass Fahrzeugaufbauten den Laserstrahl blockieren. Diese Hindernisse sind in der Grafik sichtbar als Nullpunkt direkt an der Fahrzeugfront. Dies erfordert zusätzliches Filtern der durch den rplidarfilter erzeugten Punktwolke. Die Weiterverarbeitung der Punktwolke zum Zwecke der Abstandsermittlung zum vorausfahrenden Fahrzeug erfolgt im Lidar-Distance-Filter.

Eine Möglichkeit zur Verbesserung der Hinderniserkennung ist das Verwenden eines Ringbuffers für Punktwolken. Innerhalb der Projektgruppe wurde der bereitgestellte Filter derart abgeändert, dass jeweils eine konstante Anzahl der aktuellsten vom Lidar gelieferten Punktwolken logisch mithilfe eines Ringbuffers übereinandergelegt werden, um die Anzahl erkannter Hindernispunkte potentiell zu vervielfachen. Der Ringbuffer enthält somit die Punktwolken mehrerer Zeitpunkte. Zum einen erzeugen dadurch Objekte, die relativ zum Lidar eine konstante Geschwindigkeit haben, deutlich mehr Hindernispunkte und sind deutlicher von falsch-positiven zu unterscheiden. Zum anderen erzeugen andere sich bewegende Objekte sog. Schatten, siehe Abbildung 12.5, da sie zu unterschiedlichen Zeitpunkten unterschiedlich weit entfernt vom Lidar sind. Da jedoch die aktuellste Punktwolke auch immer Teil des Ringbuffers ist, werden auch Hindernispunkte mit minimalem Abstand zum Lidar eines sich potentiell nähernden Objektes berücksichtigt. Die weiter entfernten Hindernispunkte früherer Zeitpunkte dieses Objekts sind als wenig sicherheitskritisch einzustufen. Für die meisten Anwendungen dürfte die Verwendung eines Ringbuffers daher eine Verbesserung darstellen.

Ein reales Szenario zeigen Abbildung 12.3 sowie Abbildung 12.4. Zum einen ist die Visualisierung der Punktwolke sichtbar, zum anderen der reale Versuchsaufbau.

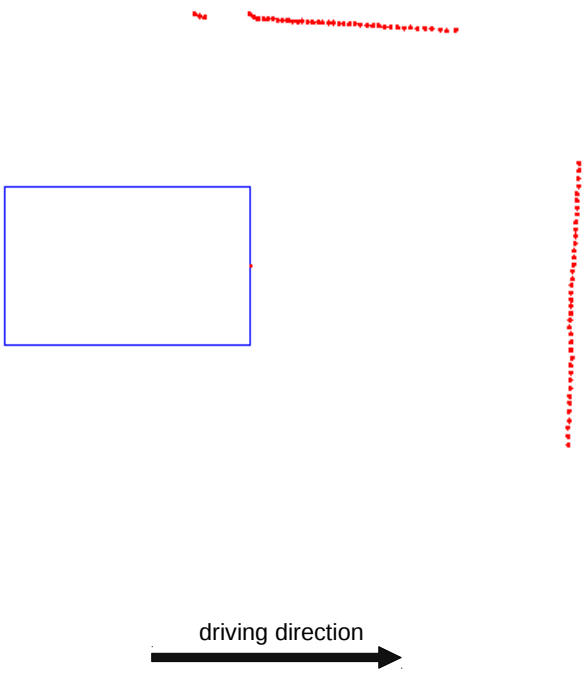


Abbildung 12.3: Visualisierung einer Punktwolke. Das blau dargestellte Fahrzeug steht vor zwei aufgebauten Kartons, welche klar durch die erzeugten Hindernispunkte sichtbar sind. Abbildung 12.4 zeigt eine Fotografie des Szenarios.

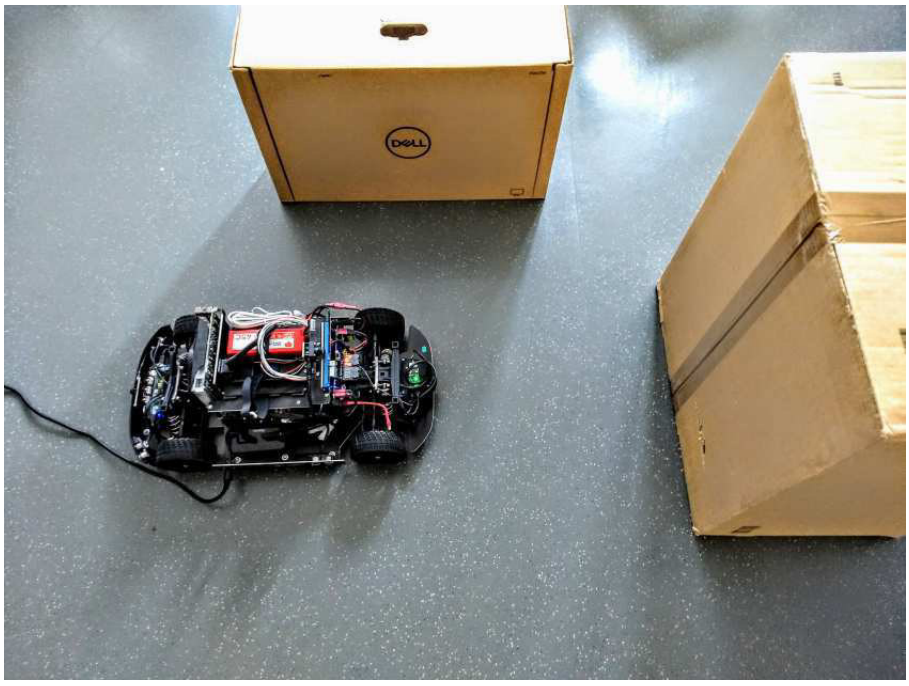


Abbildung 12.4: Fotografie vom Versuchsaufbau des in Abbildung 12.3 dargestellten Szenarios.

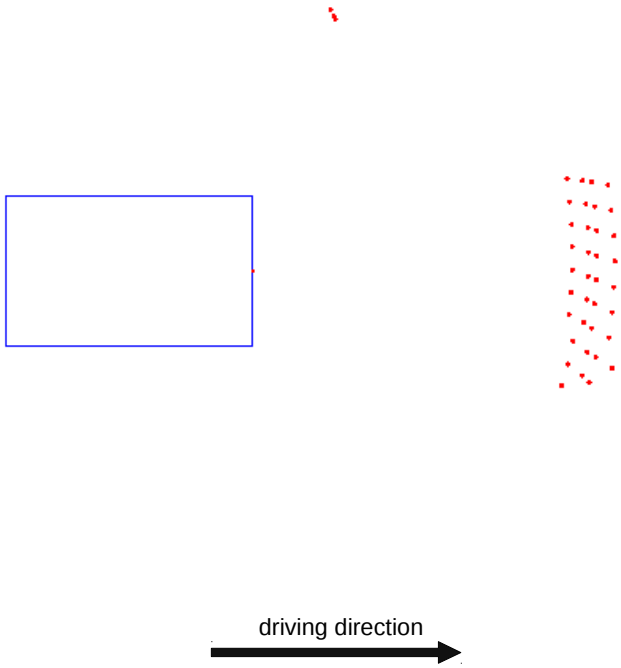


Abbildung 12.5: Visualisierung der gepufferten Punktwolke mit einer Ringbuffergröße von 4. Sichtbar ist der Schatten eines sich bewegenden Objekts.

12.1.2 Cinco-Master-Filter

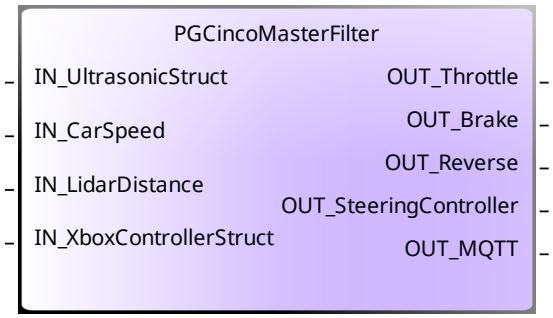


Abbildung 12.6: Cinco-Master-Filter

Der Cinco-Master-Filter trennt die Logik nach In- und Output-Signalen und der eigentlichen Logik des generierten ADAS. Der Cinco-Master-Filter bildet die Schnittstelle zwischen Fahrzeug und generiertem ADAS. Dabei wird die Steuerungslogik des Fahrzeugs über eine bereitgestellte API abstrahiert, sodass im generierten ADAS nicht auf ADTF- bzw. fahrzeugspezifische Ansteuerung der Sensorik und Aktuatorik Rücksicht genommen werden muss. Das ADAS kommuniziert somit ausschließlich über die API mit dem Fahrzeug. Zu beachten ist, dass durch diese Abstraktion das Fahrzeug real aber auch simuliert sein kann.

Wie in Abbildung 12.10 dargestellt, bietet die *CarAPI* über Getter-Methoden die Sensordaten an und steuert über Setter-Methoden die Aktuatorik. Die meisten Getter-Methoden greifen intern auf einen Ringbuffer zu, der zur Mittelwerts- oder Medianbildung über Sensorwerte benutzt wird. Dadurch übernimmt die *CarAPI* einfaches Filtern bestimmter Werte, was sich insbesondere bei realen Sensoren positiv widergespiegelt hat.

Die im Cinco-Produkt modellierte Logik wird in die `run`-Methode des *GeneratedADAS* generiert, welche in getakteten Abständen im Cinco-Master-Filter aufgerufen wird. Vor dem Durchlaufen der `run`-Methode sorgt der Cinco-Master-Filter dafür, dass aktuelle Sensor- und Eingabedaten bereitgestellt sind. Dieser Vorgang entspricht der Phase des *Input processing* in Abbildung 12.7. Nach dem Durchlaufen der `run`-Methode werden die über die Setter-Methoden gesetzten Werte an das Fahrzeug gesendet. Außerdem werden Eingabe- und Ausgabedaten sowie aktueller Zustand des ADAS über den Output-Pin *OUT_MQTT* versendet, um diese Informationen externen Systemen zugänglich zu machen. Diese Phasen werden in Abbildung 12.7 als *Execution* und *Output processing* bezeichnet. Nach dem Durchlaufen der Aufrufkette beginnt sie getaktet von vorn.

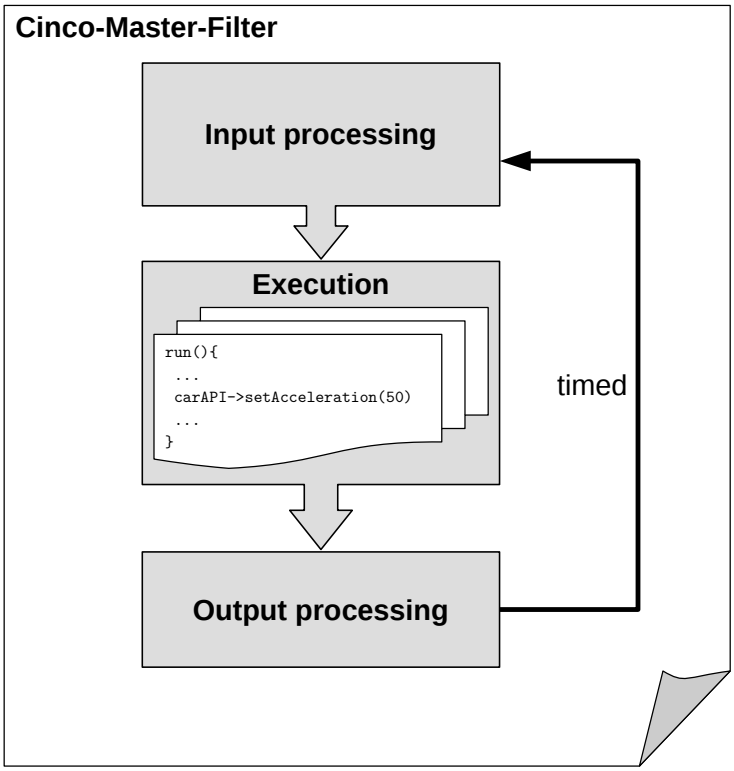


Abbildung 12.7: Aufrufkette des Cinco-Master-Filters

Generated ADAS

Das Generated ADAS gibt die Struktur des generierten Assistenzsystems vor. Die *CarAPI* verwendet keine ADTF-spezifischen Datentypen. Dadurch wird im Cinco-Produkt

von ADTF entkoppeltes Generieren ermöglicht und eine universelle sowie plattformunabhängige Schnittstelle definiert. Die Zusammenführung und Verwaltung der Schichten erfolgt im Cinco-Master-Filter (Unterabschnitt 12.1.2). Für den Generator sind somit ADTF-spezifische Elemente und Strukturen unsichtbar. Dabei stellt die *CarAPI* über ein *Gamepad*-Objekt, über as die Interaktion des Nutzers abgebildet werden kann. Über ein *CarData*-Objekt werden die aktuellen Sensorwerte verfügbar gemacht, sowie Steuerungsbefehle an das Modellfahrzeug weitergeleitet. Die Ansteuerung der einzelnen LEDs des Xbox-Controllers werden in *LEDConfiguration* enumeriert. Die Xbox-Controllersignale werden über ein *GamepadOutputs*-Struct verarbeitet. Das dazugehörige Klassendiagramm ist in Abbildung 12.10 dargestellt.

12.1.3 Lidar-Distance-Filter

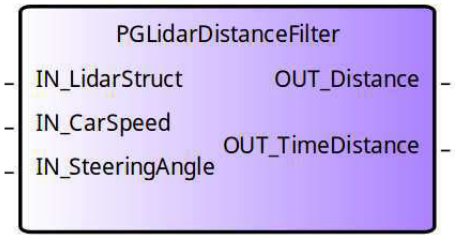


Abbildung 12.8: Lidar-Distance-Filter

Der Lidar-Distance-Filter ermittelt die räumliche und zeitliche Distanz zum vorausfahrenden Fahrzeug bzw. Hindernis. Als Eingaben zur Berechnung dienen dazu die aktuelle Geschwindigkeit des Fahrzeugs in m/s sowie eine vom Lidar erzeugte Punktwolke der erkannten Hindernisse, visualisiert in der Abbildung 12.2. Der Lenkwinkel des Fahrzeugs wird in der aktuellen Entwicklungsiteration nicht für die Distanz-Berechnung verwendet, allerdings wird diese Information für Kurvenfahrten zur Berechnung benötigt. Innerhalb der Projektlaufzeit wurden Kurvenfahrten allerdings nicht weiter verfolgt. Um eine effektive Erkennung von Hindernissen unter Ausschluss von Fehlerkennungen zu ermöglichen, wird der Sensorbereich longitudinal in unterschiedlich breite und tiefe *Distanz-Buckets* genannte Bereiche partitioniert. Im Nahfeld des Sensors wird ein kleiner Bereich verwendet, in höherer Entfernung wächst der Bereich, in dem Hindernissen erkannt werden. Um die Rate der falsch-positiv erkannten Hindernisse zu minimieren und so einen fehlerärmeren Distanzwert zu berechnen, wird eine Mindestanzahl von Hindernispunkten pro Sensorbereich festgelegt. Eine Mindestanzahl wird festgelegt, da die Sensordaten z.B. durch Reflexionen verunreinigt werden. Reale Hindernisse erzeugen in aller Regel mehr Hindernispunkte als die festgelegte Mindestanzahl. Die Anzahl der Sensorbereiche sowie deren Abmessungen und Mindestanzahl von Punkten lassen sich parametrisieren. Die genaue Parametrisierung des Filters muss noch in einem Feldtest explorativ bestimmt werden, die Abmessungen sind allerdings

zu einem gewissen Grad durch die Fahrzeugabmessungen vorgegeben. Dabei müssen für Sensordaten verschiedener reale Hindernisse unterschiedliche Parameter getestet werden. Von besonderer Bedeutung für ACC-Funktionalität ist die Beschaffenheit der Sensordaten im Falle eines vorausfahrenden Fahrzeugs. Abbildung 12.9 stellt das angewendete Verfahren dar.

Die Punkte A, B, C, C', Z sowie Z' stellen zur Veranschaulichung vereinfachte Hindernisse dar. Die mit I., II. und III. beschrifteten Bereiche stellen die Distanz-Buckets dar. Dabei wird ein Bucket durch einen *Bucket-Basispunkt* $(x_i, 0)$ auf der X-Achse sowie einer Ausdehnung in X- und Y-Richtung beschrieben.

Der rot-schraffierte Halbkreis an der Fahrzeugfront markiert den vom Sensor nicht detektierbaren Bereich. Hindernisse in diesem Bereich sind zu nah am Fahrzeug. Daraus resultiert, dass Punkt A nicht zur Distanzmessung beiträgt und damit nicht erkannt wird. Im Bucket I. befindet sich der Punkt B, im Bucket II. befindet sich kein Punkt und im Bucket III. befinden sich die Punkte C und C'.

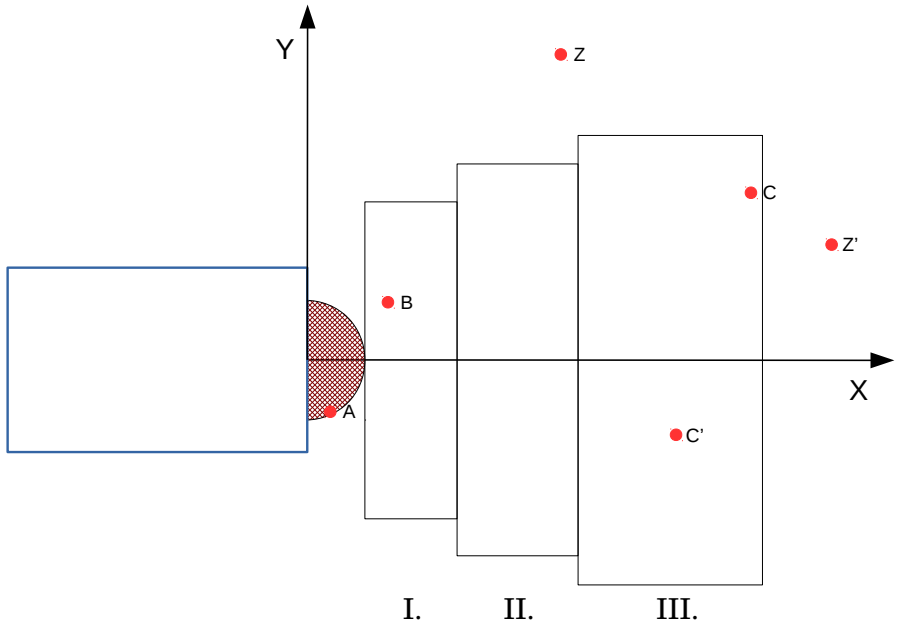


Abbildung 12.9: Schematische Darstellung der verwendeten Distanz-Buckets

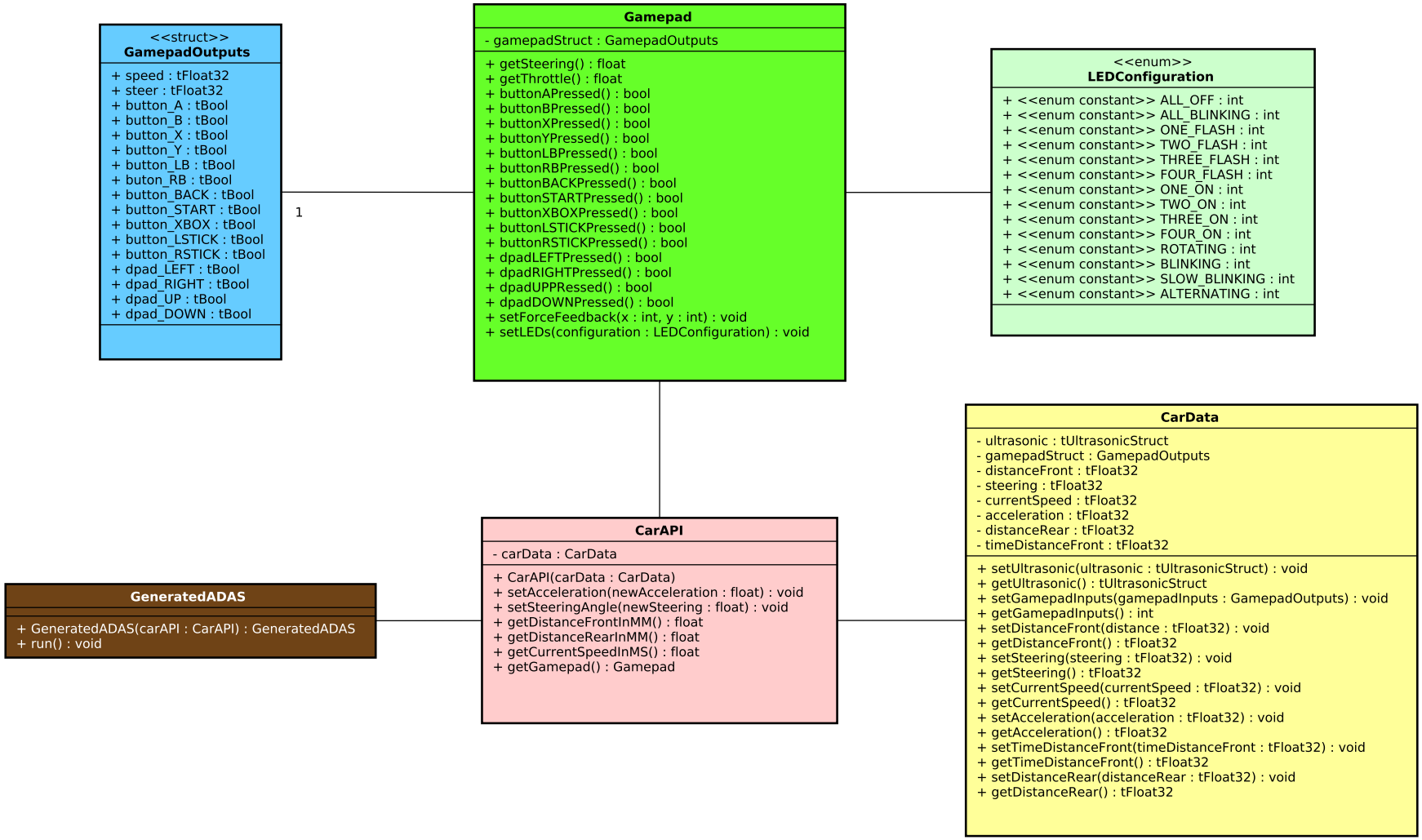


Abbildung 12.10: UML-Diagramm

Da sich in Bucket I. nur ein Punkt befindet und damit nicht die Mindestanzahl pro Bucket in diesem Beispiel erreicht wird, sieht das Verfahren den Punkt B als falsch-positiv erkannt an.

In Bucket III. werden hinreichend viele Punkte erkannt. Als Distanzwert wird allerdings nicht der direkte Abstand zu den Punkten C und C' ermittelt, sondern die dem Fahrzeug zugewandte Grenze von Bucket III. zurückgegeben. Dies entspricht der X-Koordinate x_3 des Bucket-Basispunktes.

Die Punkte Z sowie Z' werden nicht für die Distanzbestimmung berücksichtigt, da sie außerhalb jedweder Bucket-Grenzen liegen.

Der Algorithmus iteriert vom nähesten zum entferntesten Bucket und prüft für jeden, ob die Mindestanzahl an Hindernisspunkten für den Bucket erreicht wird. Ist dies nicht der Fall, wird der nächste Bucket überprüft. Diese Vorgehensweise sorgt dafür, dass immer der geringste Abstand zum Hindernis geliefert wird.

12.1.4 Xbox-Receiver-Filter

Abbildung 12.11: Xbox-Receiver-Filter

Der Xbox-Receiver-Filter leitet die über den kabellos verbundenen Xbox-Controller (Abbildung 12.12) eingegebenen Steuerungssignale an den Cinco-Master-Filter weiter. Die Ausgaben sind vom Typ *GamepadOutputs* (siehe Abbildung 12.10). Es sind alle in Abbildung 12.12 sichtbaren Controller-Tasten auslesbar. Der Filter wurde entwickelt, da eine manuelle Steuerung im autonomen Fahrmodus des Fahrzeugs nicht möglich ist, für die Entwicklung eines ACCs allerdings Benutzereingaben wie die Gaspedalstellung erforderlich sind. Im RC-Modus lässt sich das Fahrzeug ausschließlich mit der RC-Fernbedienung steuern. Der Filter ermöglicht die Änderung des Steuersignals zum einen mit dem linken und zum anderen mit dem rechten Analogstick des Controllers. Das durch den rechten Analogstick erzeugte Steuersignal wird dabei mit nur 30% Intensität weitergeleitet. Diese Möglichkeit wurde geschaffen, um (reale sowie simulierte) Fahrzeuge bei höheren Geschwindigkeiten feiner steuern zu können.

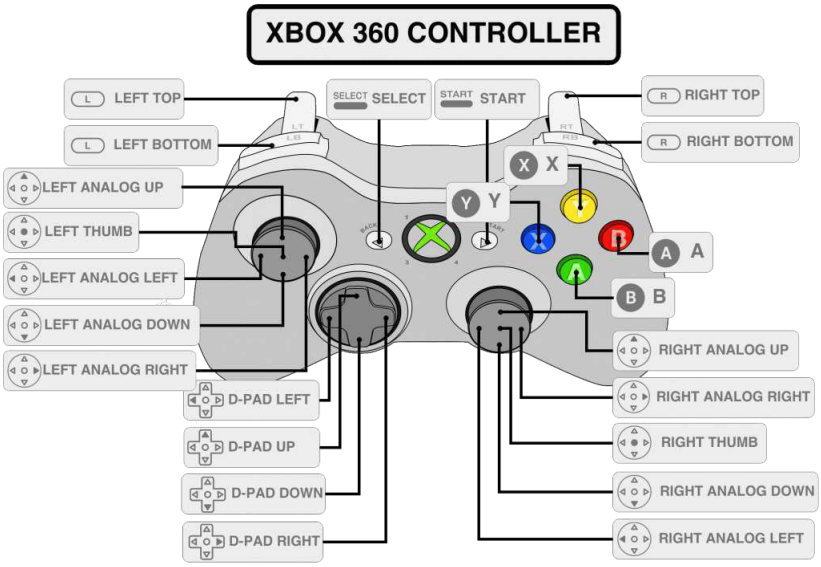


Abbildung 12.12: Xbox-Controller Layout [20]

Intern arbeitet der Filter mithilfe eines Systemdienstes, der die Anbindung des Xbox-Controllers steuert. In einem eigenen Thread werden die Eingabebefehle verarbeitet und über die Interprozess-Kommunikations-Schnittstelle *Desktop-Bus (D-BUS)* Funktionen wie die Gamepad-LEDs und das Force-Feedback angesteuert. Diese Funktionen sollen als Rückmeldungs-Schnittstelle zum Fahrer verwendet werden. So könnte beispielsweise bei drohendem Auffahren das Force-Feedback des Controllers ein haptisches Feedback zur Distanz vermitteln. LEDs könnten benutzt werden, um die einzelnen ACC-Zustände zu visualisieren.

12.1.5 Emergency-Brake-Filter

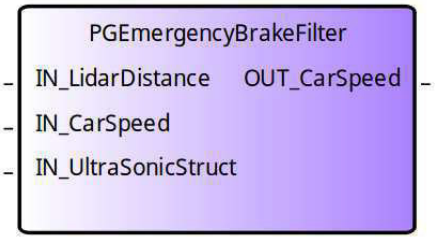


Abbildung 12.13: Emergency-Brake-Filter

Die Funktionalität des Filters könnte später in generierte Assistenzsysteme integriert werden und somit Teil des generierten ADAS sein. Um jedoch eine sichere Testumgebung herzustellen, in der eine fehlerhafte Implementierung des Generators nicht zu Schäden am Modellauto oder Umgebung führen, wird ein separater Filter bereitgestellt. Die in Abschnitt 12.2 vorgestellten Konfigurationen enthalten den vorgestellten Filter nicht mehr, da die Funktionalität bereits im ACC modelliert ist.

Der Emergency-Brake-Filter schleift die vom Fahrzeug zu fahrende Geschwindigkeit durch, wenn Abstände zu Hindernissen über einem gewissen Schwellwert liegen. Falls die Abstände zu gering sind, wird das Fahrzeug kontrolliert abgebremst. Der Schwellwert ist abhängig von der aktuell gefahrenen Geschwindigkeit. Für Rückwärtsfahrten wird auf Sensordaten der hinteren Ultraschallsensoren zurückgegriffen.

12.1.6 MQTT-Receiver-Filter

Der MQTT-Receiver-Filter ermöglicht das generische Empfangen und Senden von Nachrichten über einen sog. MQTT-Broker. Dieser kümmert sich um die Verbindung zwischen Clients und dem eigentlichen Nachrichtenaustausch.

Zunächst wird kurz grundlegend auf das zugrundeliegende *Message Queue Telemetry Transport (MQTT)*-Protokoll eingegangen. Anschließend werden Implementierungsdetails und die Verwendung des entwickelten Filters erläutert.

MQTT-Protokoll

Das MQTT-Protokoll ist ein offenes Client-Server-Nachrichtenprotokoll für Machine-to-Machine-Kommunikation, das insbesondere auch trotz hoher Netzwerkverzögerungen Übertragung von Nachrichten zwischen Geräten ermöglicht [12].

Clients verbinden sich mit einem Server („Broker“) und können Nachrichten an ihn senden („publishen“). Nachrichten setzen sich grundlegend aus einem Topic und dem Nachrichteninhalt zusammen. Das Topic wird als String angegeben und kann aus mehreren Ebenen bestehen, welche durch / getrennt werden:

- pgacc/car1
- pgacc/car1/currentState
- pgacc/car2/currentState

Clients können Nachrichten-Topics beim Broker abonnieren und empfangen daraufhin Nachrichten vom Broker mit entsprechendem Topic. Um alle Topics ab einer bestimmten Ebene zu abonnieren, lässt sich das Zeichen # wie folgt setzen [12]:

- #
- pgacc/#

So bezeichnet # alle Topics, pgacc/# bezeichnet alle Topics, die mit pgacc/ beginnen.

Nachrichten können des Weiteren mit drei verschiedenen Dienstgüten (*Quality of Service*) versendet werden [12]:

QoS 0: Nachricht höchstens einmal senden. Bei Verbindungsabbrüchen kommt die Nachricht möglicherweise nicht an.

QoS 1: Nachricht mindestens einmal bestätigt senden. Die Nachricht wird solange gesendet, bis sie von Empfänger bestätigt wird, kann dabei aber auch mehrfach ankommen.

QoS 2: Nachricht so senden, dass sie beim Empfänger exakt einmal ankommt.

Filter-Implementierung

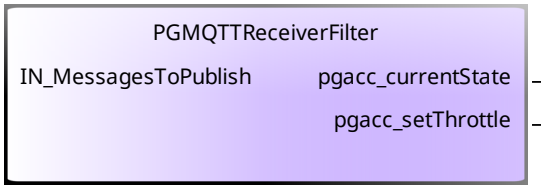


Abbildung 12.14: MQTT-Receiver-Filter

Mit dem MQTT-Receiver-Filter lassen sich über einen MQTT-Broker Nachrichten senden sowie empfangen. Der Filter lässt sich generisch über seine Filter-Properties konfigurieren. MQTT-Server-Adresse sowie automatisch generierte Client-ID lassen sich über den ADTF-Property-Browser konfigurieren, siehe auch Abbildung 12.15. Zudem lassen sich die zu abonnierenden Topics als kommaseparierte Liste konfigurieren. Besonderheit ist dabei, dass für jedes abonnierte Topic ein dynamischer ADTF-Output-Pin initialisiert wird. Die Output-Pin-Namen werden dynamisch aus dem Topic-Namen generiert, wobei ausschließlich ADTF-kompatible Zeichen verwendet werden. Insbesondere das Zeichen / ist für ADTF-Outpin-Namen nicht erlaubt, weshalb es durch _ ersetzt wird. Ankommende Nachrichten werden ihrem Topic entsprechend an die jeweiligen Output-Pins weitergeleitet. Falls Nachrichten-Topics auf mehrere Pins gematched werden, werden sie auch entsprechend auf mehrere Pins weitergeleitet. Für an den Broker zu versendende Nachrichten steht ein allgemeiner Input-Pin zur Verfügung. Als Binärtyp für über Pins empfangene und zu sendende Nachrichten wird der C++-Standardtyp std :: pair<string, string> verwendet, wobei der erste Tupel-Eintrag für das Topic verwendet wird und der zweite Tupel-Eintrag für den Nachrichteninhalt.

Abbildung 12.14 zeigt eine Beispielkonfiguration des Filters. Als dynamische Output-Pins bestehen *pgacc_ currentState* sowie *pgacc_ setThrottle*.

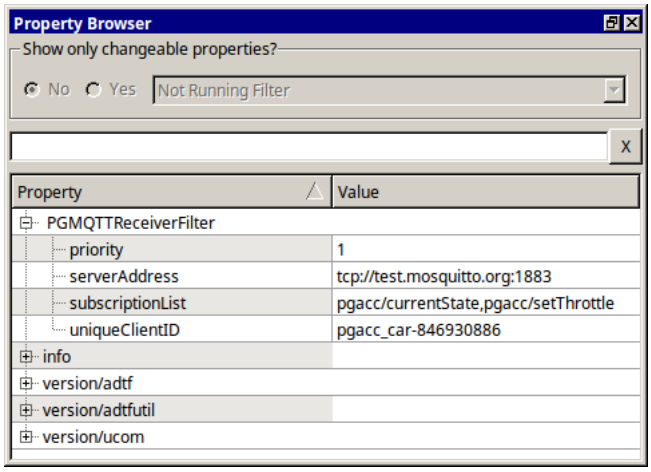


Abbildung 12.15: MQTT-Receiver-Filter

Als Grundlage für die MQTT-Implementierung wird der MQTT-C++-Client des Eclipse Paho-Projekts [18] verwendet, welcher das Protokoll implementiert und über eine C++-API verfügt, die eingekapselt im MQTT-Receiver-Filter verwendet wird.

12.1.7 MQTT-LidarPoint-Visualisation-Filter

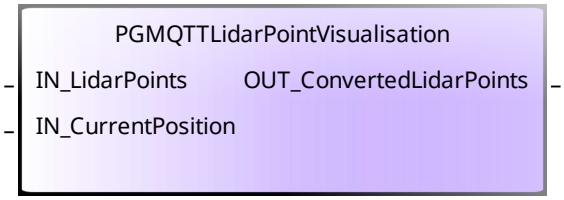


Abbildung 12.16: Konfigurierbare Eigenschaften des MQTT-LidarPoint-Visualisation-Filters

Ziel des MQTT-LidarPoint-Visualisation-Filters ist die Darstellung erkannter Hindernisse durch das Lasersystem der Halle des Innovationslabors (siehe Abschnitt 14.2). Da während das Fahrzeug fährt, Sensorwerte nur abstrakt dargestellt werden können (siehe Abbildung 12.2), ist wegen der hohen Dynamik die Interpretierbarkeit erschwert. Das Lasersystem kann durch den Lidar erkannte Hindernispunkte dort auf dem Boden darstellen, wo sie sich in der Realität befinden. Auf dem in Abbildung 12.17 dargestellten Foto ist das Ergebnis des Filters dargestellt.

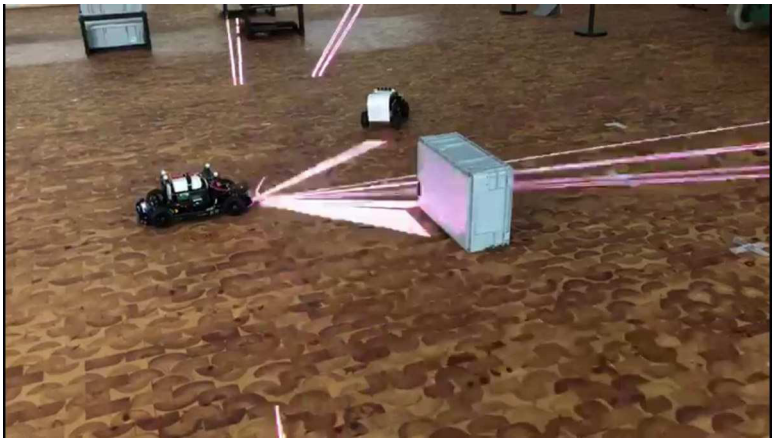


Abbildung 12.17: Laservisualisierung vom Lidar erkannter Hindernisse

Der Filter erzeugt eine Liste von durch den Lidar erkannten Hindernispunkten im *JavaScript Object Notation (JSON)*-Format. Die vom in Unterabschnitt 12.1.1 vorgestellten Filter erzeugten Rohpunkte werden anhand der Fahrzeugposition ins Koordinatensystem der Halle des Innovationslabors übertragen.

Der Eingabeparameter *CurrentPosition* besteht dabei aus drei Komponenten:

- Absolute Fahrzeugposition in x- und y-Koordinate der Halle des Innovationslabors (siehe Abschnitt 14.2)
- Winkel der Ausrichtung des Fahrzeugs

Die Ausgabe des Filters erfolgt als MQTT-Nachricht. Dabei kann das Topic über Filterparameter individuell gesetzt werden. Die Payload der Nachricht ist im JSON-Format kodiert. Das JSON-Objekt enthält dazu ein Attribut namens points. Das points-Objekt besteht aus einem mehrdimensionalen Array von Zahlen. Ein einzelner Hindernispunkt wird als zweidimensionales Array [x,y] repräsentiert. Diese MQTT-Nachricht wird über den MQTT-Broker der Halle bereitgestellt und von der Laseransteurung interpretiert.

12.1.8 Throttle-Brake-Converter-Filter

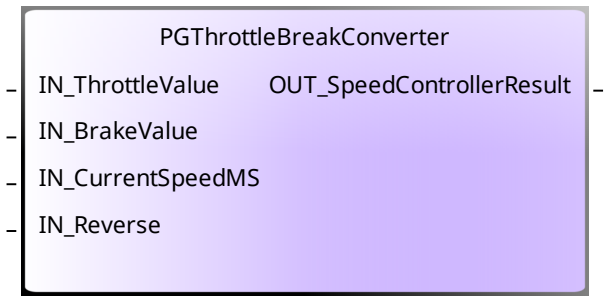


Abbildung 12.18: Throttle-Brake-Converter-Filter

Der Filter wurde entwickelt, da allgemeine Simulationslogik Gas- und Bremspedal-Steuerung annimmt, das Modellfahrzeug aber mit einem einzelnen Leistungswert (SpeedController) arbeitet. Der Filter ist daher nur bei Realfahrten des Modellfahrzeugs von Nöten. Da es lediglich einen einzelnen Leistungswert gibt, ist beispielsweise kein explizites Bremsen möglich, sondern lediglich eine steuerbare Fahrleistung in Rückwärtsrichtung. Der Leistungswert des Modellautomotors liegt dabei zwischen 100 in Vorwärtsfahrrichtung und -100 in Rückwärtsrichtung. Der Filter übernimmt deshalb eine Konvertierung von Gas- und Bremspedal-Stellungswerten in einen Leistungswert. Der Wertebereich des Ausgabewertes lässt sich zudem über die Filtereigenschaften konfigurieren (*speedControllerNegativeResultCap*, *speedControllerPositiveResultCap*).

Der Throttle-Brake-Converter-Filter berechnet gedächtnisbehaftet und parametrisierbar aus eingehenden Gas- und Bremspedal-Stellungswerten sowie der aktuellen Fahrzeuggeschwindigkeit einen von der Motorsteuerung interpretierbaren Leistungswert. Außerdem wird die aktuell zu fahrende Fahrtrichtung als Eingabewert berücksichtigt.

Sind Gas- und Bremspedal beide in Nullstellung, so muss realistischerweise der Ausgabewert mit der Zeit gegen Null gehen, da das Fahrzeug ansonsten nicht ausrollt. Dazu wird der aktuelle Ausgabewert mit dem sog. *dragFactor* multipliziert, welcher ein konfigurierbarer Zahlenwert im Intervall von $(0, 1)$ ist. Je näher der *dragFactor* bei 1 liegt, desto langsamer erfolgt das Ausrollen:

$$\text{Ausgabewert}_{neu} = \text{Ausgabewert}_{alt} \cdot dragFactor \tag{12.1}$$

Weiter wird über den Konverter sichergestellt, dass zwei aufeinanderfolgende Leistungswerte keine zu große Differenz haben. Grundsätzlich wäre es möglich, über den in Abschnitt B des Anhangs vorgestellten Arduino-Communication-Filter aufeinanderfolgende Werte mit großer Differenz, beispielsweise zunächst einen Wert von 100 direkt gefolgt von einem Wert von -100 zu senden. Das würde aufgrund der durch Massenträgheit wirkenden Kräfte dazu führen, dass die Stromaufnahme des Elektromotors beim Wechsel von voller Leistung nach vorne zu voller Leistung Rückwärts zu groß wird und die verbaute Schmelzsicherung zerstört wird. Dieses Problem tritt insbesondere bei Handsteuerung über den RC-Controller auf.

Der konfigurierbare *gainFactor* bestimmt, wie stark auf eine Gaspedalanforderung reagiert wird. Das Vorgehen bei einer Bremspedalanforderung ist abhängig von ihrer Intensität. Bei einem Bremspedaldruck von weniger als 45% wird der aktuelle Ausgabewert sukzessive in Abhängigkeit der Bremsintensität verringert, ist demnach gegebenenfalls positiv und treibt das Fahrzeug weiter an, jedoch langsamer werdend. Bei einem Bremspedaldruck von 45% und mehr wird das Bremsen unter Berücksichtigung der Geschwindigkeit vorgenommen. Der Wert von 45% hat sich sowohl in experimentellen Realfahrten als auch während Simulationsfahrten als sinnvoll erwiesen. Der Grund dafür ist, dass für eine starke Bremswirkung

ein negativer Ausgabewert benötigt wird, dieser jedoch andauernd zu einer Änderung der Fahrtrichtung des Fahrzeugs führen würde. Bei hohen gefahrenen Geschwindigkeiten wird ein entsprechend hoher negativer Ausgabewert erzeugt, der zu einer starken Verzögerung des Fahrzeugs führt. Bei geringeren Geschwindigkeiten ist der negative Ausgabewert geringer, da das Fahrzeug sonst direkt die Fahrtrichtung ändern würde. Steht das Fahrzeug fast still und die Bremsanforderung ist hoch, so geht der Ausgabewert damit gegen Null.

Der separate Inputpin *Reverse* emuliert das Einlegen des Rückwärtsgangs. Das Ergebnis der Berechnung des Motorsteuerungswerts wird dabei invertiert.

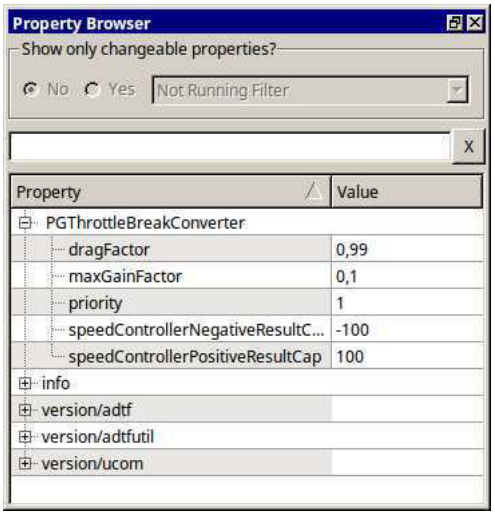


Abbildung 12.19: Throttle-Brake-Converter-Filter

Die in Abbildung 12.19 eingestellten Parameter wurden während Realfahrten explorativ ermittelt.

12.2 ADTF-Konfigurationen

Wie in Abschnitt 5.1 angerissen, bestehen Konfigurationen aus miteinander verbundenen Filtern. Innerhalb der Projektgruppe sind grundlegend zwei verschiedene Konfigurationen entwickelt worden, welche zum einen auf Realfahrten und zum anderen auf Simulationsfahrten abzielen. Bei Simulationsfahrten wird unterschieden zwischen einer interaktiven Konfiguration und einer Konfiguration zum automatischen Abfahren von Testszenarien (siehe Kapitel 9). In der interaktiven Simulationskonfiguration ist vorgesehen, dass mittels Benutzereingaben das Fahrzeug frei gesteuert und zudem nach Bedarf auf Funktionen des ADAS zurückgegriffen werden kann. So können schwerwiegende Fehler in der Implementierung des ADAS bereits vor einer ersten Realfahrt gefunden und behoben werden. Gleichzeitig wird bei Simulationsfahrten auf perfekte Sensorik zurückgegriffen, was eine zusätzliche Fehlerquelle ausschließt.

Grundsätzliche Gemeinsamkeit aller Konfigurationen ist der eingebundene Cinco-Master-Filter (siehe Unterabschnitt 12.1.2), da dieser die grundlegende Architektur für generierte ADAS bereitstellt.

12.2.1 Realfahrt-Konfiguration

Abbildung 12.20 zeigt die Realfahrt-Konfiguration. Hauptsächlich dient diese Konfiguration zur Durchführung von Realfahrten in der Halle des Innovationslabors (siehe Abschnitt 14.2).

Augenscheinlich ist die Verwendung von mit Fahrzeug-Sensorik und -Aktorik interagierenden Filtern. So ist links oben zunächst der rplidarfilter (Unterabschnitt 12.1.1) eingebunden. Dieser ist zum einen zu Debuggingzwecken mit einem 2D-Display verbunden, welches die Punktwolke wie in Abbildung 12.2 visualisiert. Zum anderen wird die ausgegebene Rohpunktwolke weitergeleitet an den Lidar-Distance-Filter Unterabschnitt 12.1.3 sowie den MQTT-LidarPoint-Visualisation-Filter (Unterabschnitt 12.1.7). Der Lidar-Distance-Filter extrahiert aus der Rohpunktwolke einen Distanzwert und stellt dem Cinco-Master-Filter diesen zur Verfügung.

Zusätzlich zu der Punktwolke erhält der MQTT-LidarPoint-Visualisation-Filter über den MQTT-Receiver-Filter (Unterabschnitt 12.1.6) die aktuelle Fahrzeugposition in Hallenkoordinaten des Innovationslabors. Dazu ist der Receiver-Filter so konfiguriert, dass er das MQTT-Topic pgacc/pgacc_sam/currentPosition abonniert. Dieses wird über das Kamerasystem der Halle mit Positionsdaten beliefert. Die vom MQTT-LidarPoint-Visualisation-Filter in Hallenkoordinaten transformierte Punktwolke wird wiederum über den MQTT-Receiver-Filter anderen MQTT-Clients bereitgestellt.

Der Cinco-Master-Filter benötigt zusätzlich ein Steuerungssignal, zur Verfügung gestellt durch den Xbox-Receiver-Filter (siehe Unterabschnitt 12.1.4), sowie ein über den AADC-Arduino-Communication-Filter (Abschnitt B) bezogenes Ultraschallsensordatum. Über den Inputpin *CarSpeed* erhält der Filter die aktuelle Fahrzeuggeschwindigkeit. Die Outputpins *Throttle*, *Brake* und *Reverse* werden an den Throttle-Brake-Converter-Filter (Unterabschnitt 12.1.8) weitergeleitet. Der Outputpin *SteeringController* wird direkt an den AADC-Arduino-Communication-Filter gesendet. Über den MQTT-Outputpin können weitere Daten per MQTT über den MQTT-Receiver-Filter verfügbar gemacht werden.

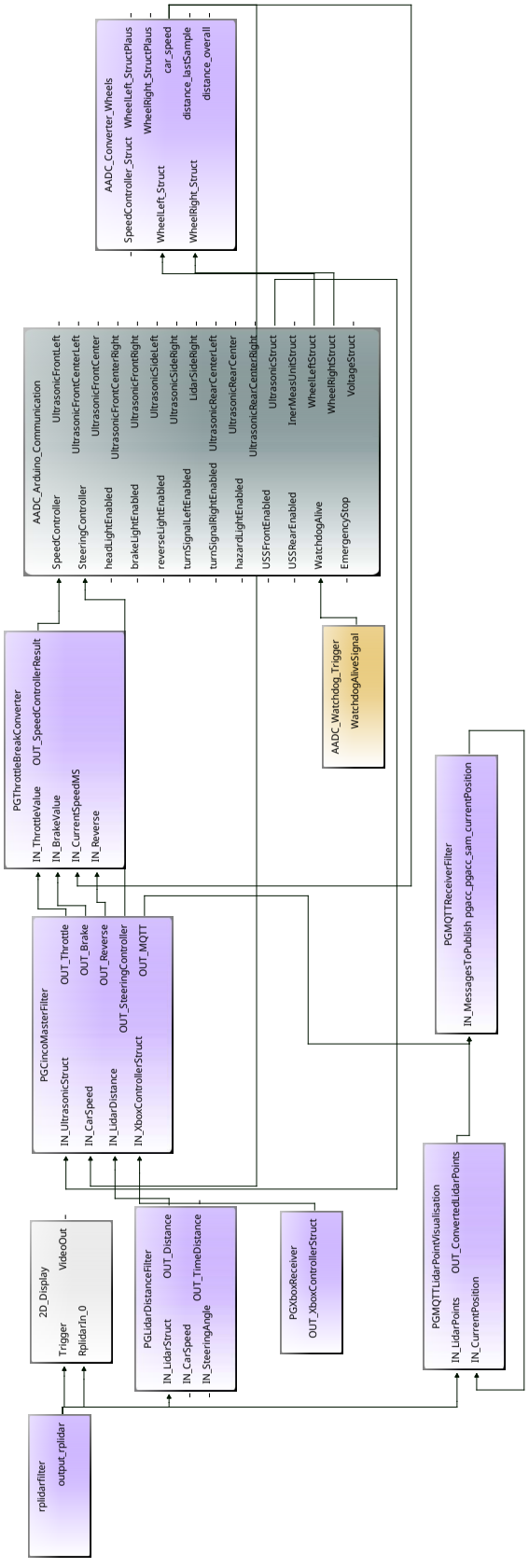


Abbildung 12.20: Visualisierung der ADTF-Realfahrt-Konfiguration

Der Throttle-Brake-Converter-Filter wandelt die Eingabedaten in ein vom AADC-Arduino-Communication-Filter interpretierbaren SpeedControllerStruct um und leitet dieses an diesen weiter. Dieser wickelt die Motorsteuerung und Lenkung über *SpeedController* und *SteeringController* ab. Weiter benötigt er ein Watchdogsignal, welches über den AADC-Watchdog-Trigger bereitgestellt wird. Über die verbundenen Outputpins *UltrasonicStruct* sowie *WheelLeftStruct* und *WheelRightStruct* werden Ultraschall und Radsensorwerte bereitgestellt. Diese werden über den AADC-Converter -Wheels-Filter (Abschnitt B) unter anderem zur Bereitstellung der aktuellen Fahrzeuggeschwindigkeit genutzt.

Alle unverbundene Input- und Outputpins werden in dieser Konfiguration nicht benötigt.

12.2.2 VTD-Simulations-Konfiguration

Diese Konfiguration wird verwendet, um in VTD ablaufende Simulationen mit ADTF zu steuern. Neben dem Cinco-Master-Filter (Unterabschnitt 12.1.2) spielt in dieser Konfiguration der VTD2ADTF-Filter Abbildung 12.24 eine zentrale Rolle.

Bei den in Abbildung 12.21 dargestellten Filtern RDB_EGO, SCP_RAW_ALL und RDB_RAW_ALL handelt es sich um die in Abschnitt B vorgestellten Ethernet-Device-TCP-Filter. Diese Netzwerkfilter sind so konfiguriert, dass sie mit dem VTD-Simulationsrechner als Host kommunizieren und die in VTD freigegebenen Ports für das Versenden und Empfangen von RCP- und SCP-Nachrichten verwenden.

Die Netzwerkfilter sind mit den entsprechenden Pins des VTD2ADTF-Filters verbunden. Der Filter RDB_EGO ist derart konfiguriert, dass Daten des vom ADAS gesteuerten Ego-Fahrzeugs empfangen werden. Die Filter RAW_ALL empfangen jeweils alle RDB/SCP-Daten.

Die mit Table_Display gekennzeichneten Komponenten werden genutzt, um wichtige Informationen als Hilfe beim Debugging von ADASs darzustellen. In dieser Konfiguration wird beispielsweise der *Throttle*-Ausgabewert des Cinco-Master-Filters abgegriffen.

Die Pins des Cinco-Master-Filters sind wie in der Realfahrt-Konfiguration entsprechend mit den jeweils zugehörigen Pins der anderen Filter verbunden.

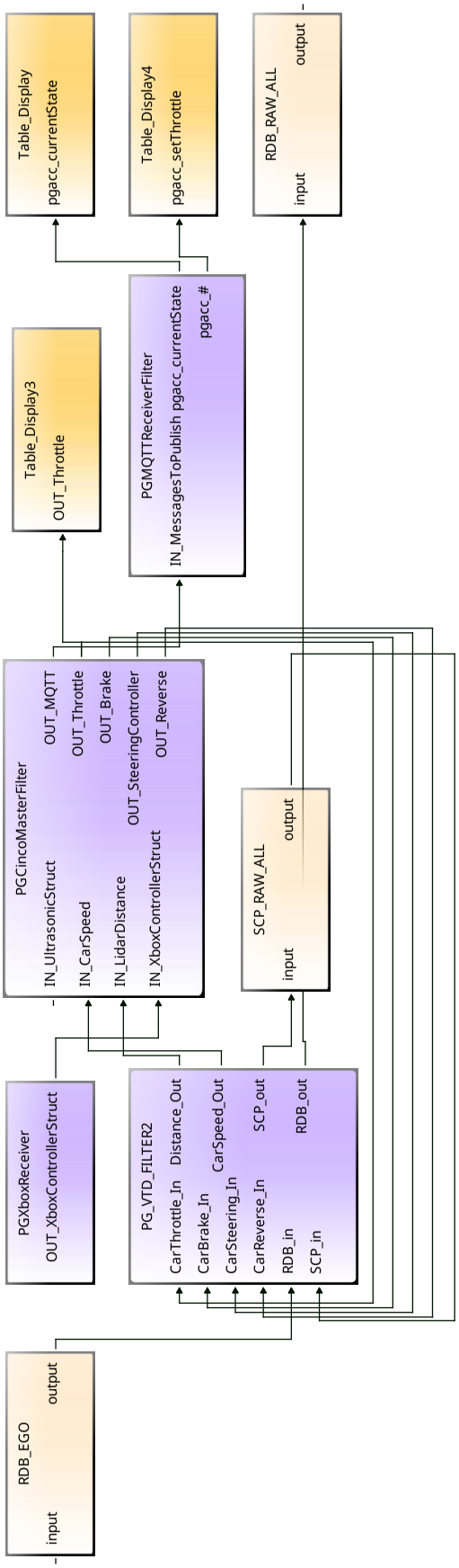


Abbildung 12.21: Visualisierung der ADTF-VTD-Simulations-Konfiguration

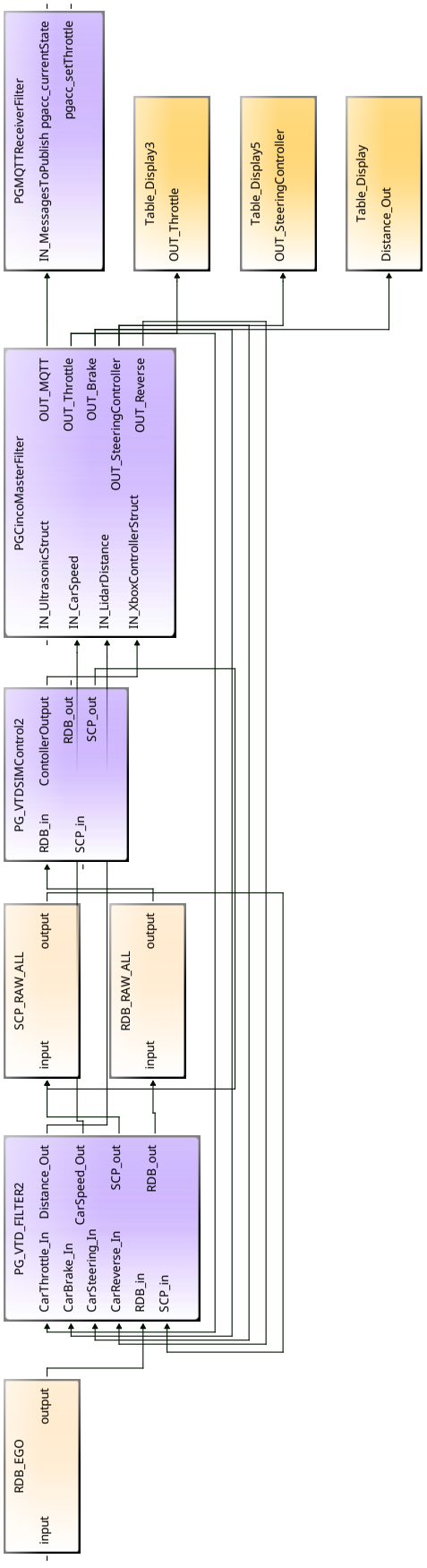


Abbildung 12.22: Visualisierung der ADTF-Testszenarien-Konfiguration

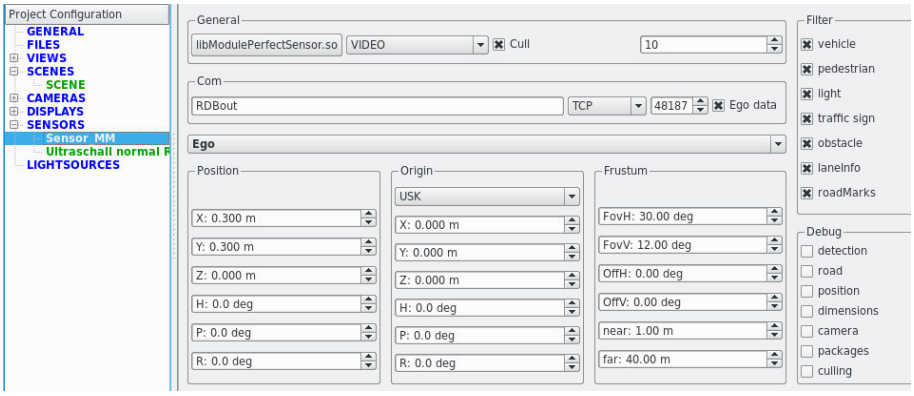


Abbildung 12.23: Sensorkonfigurationsmenü von VTD

12.2.3 Testszenarien-Konfiguration

Die in Abbildung 12.22 abgebildete Testszenarien-Konfiguration gleicht im Wesentlichen der VTD-Simulations-Konfiguration. Allerdings ist in dieser kein direktes Steuern des Ego-Fahrzeugs über den Xbox-Controller möglich. Stattdessen wird über den VTDSimControl-Filter, vorgestellt in Abbildung 12.21, die Steuerung des Fahrzeugs simuliert. Der VTD-SimControlfilter steuert über die SCP- und RDB-Outputpins die in Kapitel 9 modellierten Szenarien.

VTD-Anbindung an ADTF

Da das generierte ACC in einer ADTF-Instanz läuft und nicht nur bei Realfahrten, sondern auch innerhalb der Simulation getestet werden soll, muss eine Verbindung zwischen VTD und ADTF hergestellt werden. VTD ist in der Lage, Daten per Ethernet auszutauschen. Es kann dazu sowohl eine *User Datagram Protocol (UDP)*, als auch eine *Transmission Control Protocol (TCP)*-basierte Verbindung verwendet werden. Im Hauptmenü von VTD kann spezifiziert werden, über welche Ports welche Daten geschickt werden (siehe Abbildung 12.23). In diesem Menü können auch die zu sendenden Daten (bspw. Daten bzgl. Verkehrszeichen, Hindernisse etc.) und die Sensoren konfiguriert werden. Es können Daten in Form von RDB- und SCP-Daten gesendet werden.

Wie bereits in Abschnitt 12.2 kurz angerissen, wird dafür ein ADTF-Filter benötigt, der an verschiedene Ethernet-Device-TCP-Filter angeschlossen wird, wie es in Abbildung 12.21 zu sehen ist. Abbildung 12.24 zeigt den VTD2ADTF Filter. Dieser bekommt die Sensor- und Fahrzeugdaten des Ego-Fahrzeugs in Form von RDB-Daten über einen Ethernet-Device-TCP-Filter. Das Ego-Fahrzeug wird dabei derart gesteuert, dass normierte Werte zwischen 0 und 1, jeweils die Stärke des Durchdrückens des Gas- bzw. des Bremspedals darstellen. Der Wert 1 bedeutet dabei, dass das Pedal vollkommen durchgedrückt ist. Zudem besitzt das Ego-Fahrzeug ein Automatikgetriebe, das bedeutet, dass von dem Filter beim Be-

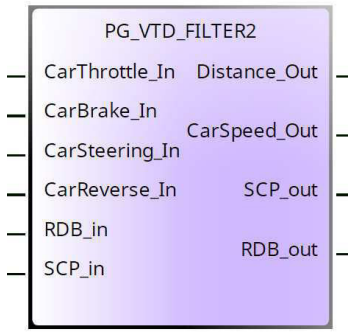


Abbildung 12.24: VTD2ADTF Filter

schleunigen und Bremsen keine direkten Gangwechsel vorgenommen werden müssen, sondern lediglich zwischen fahren, rollen, stehen sowie rückwärts fahren unterschieden werden muss. Die aktuelle Geschwindigkeit wird an den CincoMasterFilter gereicht.

12.3 Umsetzung der Evaluation mittels visueller Liveauswertung

Zur Visualisierung relevanter Sensorwerte, Monitore sowie des aktuellen ADAS-Zustands wurde innerhalb der Projektgruppe eine auf HTML- und Javascript-basierte Webanwendung entwickelt. Bestimmte Sensorwerte werden in Diagrammform als Graph dargestellt. Neben Live-Darstellung der Daten werden für jedes in der Simulation absolvierte Szenario aus der Sammlung diese Diagramme gespeichert. Somit lässt sich im Nachhinein nachvollziehen, welche Sensorwerte zu gewissen Zeitpunkten gemessen wurden.

Die Daten werden über MQTT empfangen. Dazu werden diejenigen MQTT-Topics abonniert, die von den ADTF-Filtern (siehe Abschnitt 12.1) zum Datenaustausch verwendet werden.

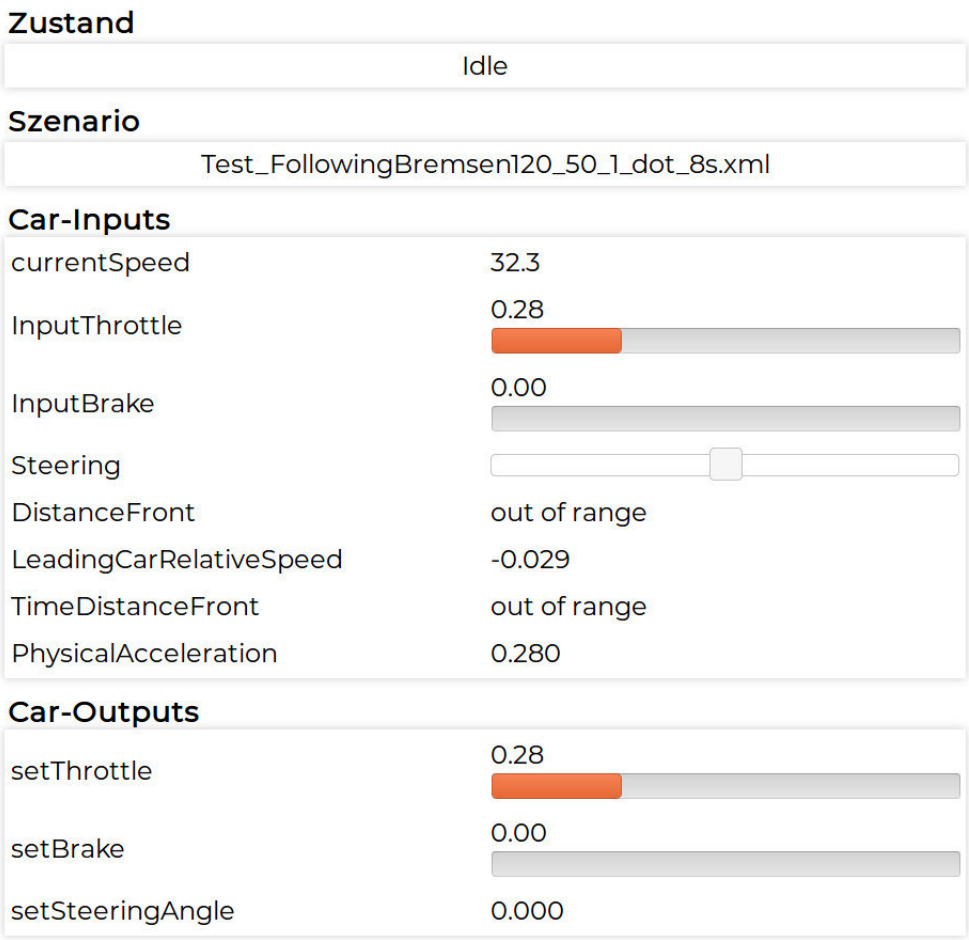


Abbildung 12.25: Hauptinfobereich der ADAS-Live-Visualisierung

Abbildung 12.25 zeigt den Hauptinfobereich der ADAS-Live-Visualisierung. Im Feld „Zustand“ wird der aktuell über MQTT übertragene Zustand des ADAS ausgegeben. Wird ein spezifisches Szenario innerhalb der Simulationsumgebung (siehe Abschnitt 6.1) absolviert, so wird der Dateiname des Szenarios im entsprechenden Feld angezeigt. Weiterhin werden im Bereich „Car-Inputs“ wesentliche Sensordaten sowie Fahrzeugdymanikwerte, also Gas- und Bremspedalstellung (*setThrottle*, *setBrake*) sowie Lenkausrichtung (*setSteeringAngle*), dargestellt. Der Bereich „Car-Outputs“ stellt die vom ADAS zu setzenden Fahrzeugdynamikwerte dar.

Unterhalb des Hauptinfobereichs befindet sich eine tabellarische Auflistung (siehe Abbildung 12.26) der zur Laufzeit ausgewerteten Monitore (siehe Kapitel 8). Anhand eines Farbindikators lässt sich der aktuelle Monitorzustand ablesen. Ähnlich einer Ampel bedeutet eine grüne Anzeige, dass die Invariante des Monitors zum aktuellen Zeitpunkt erfüllt ist, während eine rote Anzeige die Nichterfüllung der Invariante signalisiert. Eine graue

Anzeige bedeutet, dass die Vorbedingung des Monitors nicht gültig ist und somit keine Aussage über die Invariante getroffen werden kann.

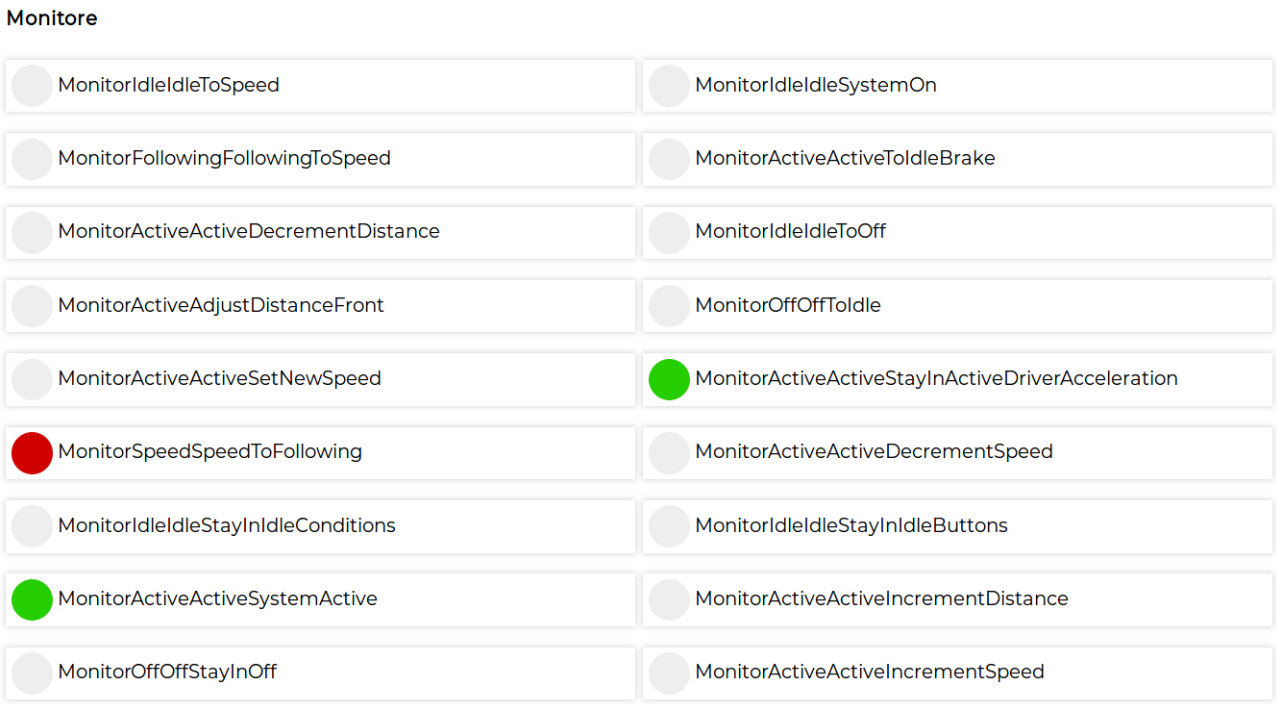
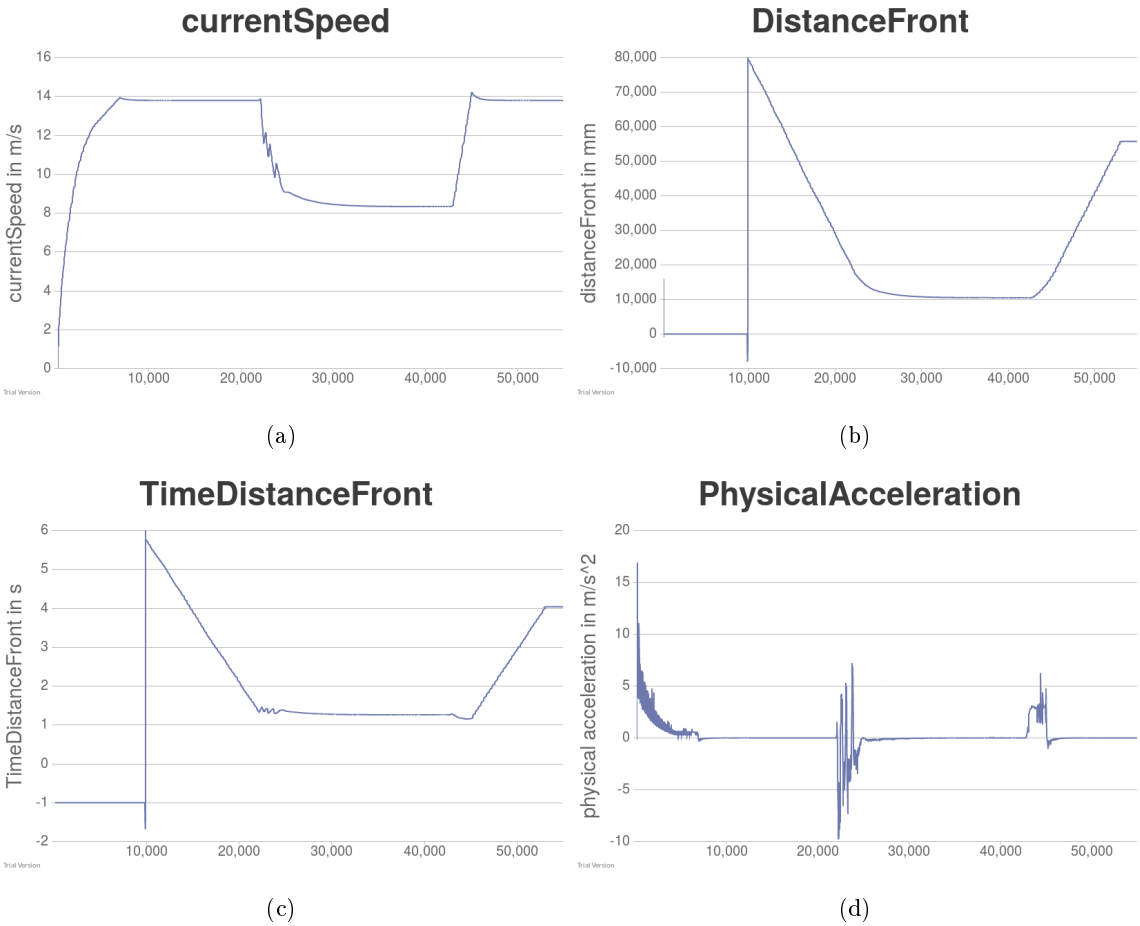


Abbildung 12.26: Auflistung der Monitorzustände innerhalb der ADAS-Live-Visualisierung

Die in Abbildung 12.27 dargestellten Diagramme zeigen graphisch die Auswertung des Testfalls bzw. Szenarios, welches bereits in Abschnitt 9.4 beschrieben wurde. Dabei ist in den Diagrammen auf der X-Achse jeweils die Zeit in ms angegeben und auf den Y-Achsen die entsprechenden Messgrößen in ihrer Einheit. Aus den Diagrammen lässt sich ablesen, dass das Szenario wie in der Beschreibung logisch vorgesehen abgelaufen ist. Es ist im Graphen der Geschwindigkeit (*currentSpeed*) zu sehen, dass das Ego-Fahrzeug zunächst auf 50 km/h (bzw. $13,89$ m/s) beschleunigt. Zu dieser Zeit befindet sich im Sensorfeld des Ego-Fahrzeugs kein Hindernis, weshalb in den Diagrammen *DistanceFront* und *TimeDistanceFront* ein Platzhalterwert von -1 verwendet wird. Bei ca. 10 s ist dann in eben diesen Diagrammen sichtbar, dass das vorausfahrende Fahrzeug in die Reichweite des Ego-Fahrzeugs kommt und der Abstand der beiden Fahrzeuge zueinander sich kontinuierlich verringert. Dies passiert bis zu dem Zeitpunkt, wo der Ausschlag bei dem Graphen der Beschleunigung (*PhysicalAcceleration*) stattfindet. Allerdings sieht man in diesem auch, dass die aus der Norm vorgegebene maximale Beschleunigungsrate von 2 m/s^2 deutlich überschritten wird. Selbes gilt ebenso für die maximale Bremswirkung (negative Beschleunigung) von $3,5$ m/s^2. Im *TimeDistanceFront*-Diagramm ist ersichtlich, dass nach diesem Zeitpunkt der eingestellte Zeitabstand von 1 s leicht überschritten wird, aber konstant bleibt. Der letzte Aspekt des Szenarios, das Beschleunigen des vorausfahrenden Fahrzeugs über die im

ACC eingestellte Geschwindigkeit, hat ordnungsgemäß funktioniert. In den Diagrammen sieht man dazu, dass das Ego-Fahrzeug bis zu seinen eingestellten 50 km/h mitbeschleunigt und dort konstant diese Geschwindigkeit beibehält, während sich der Abstand zum vorausfahrenden Fahrzeug weiter vergrößert, da dieses auf 70 km/h beschleunigt.



(a) (b)

(c) (d)

Abbildung 12.27: Von der Live-Visualisierung erzeugten Diagramme nach Absolvieren des in Abbildung 9.3 dargestellten Szenarios

Teil IV

Fallstudie: ACC

Kapitel 13

Beispiel: ACC-Modell

Um das Zusammenspiel zwischen dem AutoDSL Modell und dem Generator zu testen, wurde ein einfaches und möglichst der *ACC-ISO-Norm* entsprechendes ACC-Modell entworfen. Weiterhin sollten durch ein erstes Modellieren die Schwächen im Entwurf gefunden werden, die für die Nutzung der Software unentbehrlich sind oder dessen Behebung die Verwendung des CINCO Produkts komfortabler und einfacher macht.

13.1 ACC-Zustandsmodell

Das Modell sollte ein ACC für gerade Strecken umsetzen. Außerdem musste beachtet werden, dass die Testfahrzeuge nicht alle Features eines normalen Autos besitzen und bereits existierende Features gegebenenfalls anders als in einem normalen Fahrzeug (z.B. Beschleunigung durch den elektrischen Motor) funktionieren.

Zunächst wurde die *ACC-ISO-Norm* [13] hinzugezogen. Das ISO-Zustandsmodell (vgl. Abbildung 2.10 wurde so abgewandelt, dass die Gegebenheiten und Voraussetzungen berücksichtigt werden (vgl. Abbildung 13.1). Die drei Hauptzustände der Norm *Off*, *Standby* und *Active* konnten ohne Probleme übernommen werden. Auch das Trennen der Subzustände *FollowingControl* und *SpeedControl* in Active kann modelliert werden.

Alle Zustände der Norm sind damit enthalten. Unterschiede zeigen sich an den Guards der Kantenübergänge. Die Definitionen für *ACC On* und *ACC Off* sind in dem modellierten ACC stark vereinfacht und auf das Minimum reduziert. Das ACC wird angeschaltet, wenn der Fahrer diese Aktion durchführt.

Andere Einflüsse sind nicht modelliert, z.B. entfallen Tests über die Bereitschaft von Motor und Getriebe. Der Zustand *Selbsttest* ist ein Platzhalter, der für spätere Erweiterungen angelegt wurde. Das Ausführen eines Selbsttests soll in diesem Modell immer erfolgreich sein und wird daher in der Umsetzung später weggelassen.

Im Standby Zustand wird also sofort in den *Idle* Zustand gewechselt, der auf die Aktivierung des ACC durch den Fahrer wartet, solange dieser nicht bremst und das Fahrzeug eine gewisse Geschwindigkeit überschreitet. In der *ACC-ISO-Norm* liegt diese Geschwindigkeit bei $5m/s^2$, da hier aber mit Modellfahrzeugen gearbeitet wurde, wurde der Wert als Variable definiert, damit er später angepasst werden kann.

Sind diese Bedingungen erfüllt, kann in den Active Zustand gewechselt werden. Es wird zunächst immer in den Zustand SpeedControl gewechselt. Von SpeedControl wird in den Zustand FollowingControl gewechselt, wenn ein anderes Fahrzeug erkannt wird. Ein Fahrzeug zu erkennen bedeutet die Registrierung eines bewegenden Objektes in einem gewissen Bereich vor dem Fahrzeug. Wird dieses Fahrzeug nicht mehr erkannt, wird in den vorangegangenen Zustand zurück gewechselt.

Das ACC geht nur aus, wenn das Fahrzeug abgeschaltet wird oder der Fahrer selbst entscheidet, den ACC Modus auszuschalten.

Um das ACC in den Standby Zustand zu versetzen, kann der Fahrer bremsen oder aktiv durch Tasteneingabe wechseln. Wenn das ACC nicht bremst und das Fahrzeug langsamer als $7m/s^2$ ist, wechselt das ACC programmgesteuert in den Standby Zustand. Hier wird dann der Selbsttest Zustand (wie zuvor) betreten.

Aus den Zuständen SpeedControl und FollowingControl kann beim Auftreten von Fehlern in den Zustand Fehler gewechselt werden, der dann durch Bremsen des Fahrers das ACC in den Zustand Off versetzt.

Der Standby Zustand kann, wie der Active Zustand, durch Ausschalten vom Fahrer oder des Fahrzeugs in den Zustand Off wechseln. Theoretisch ist es auch möglich, beim Scheitern des Selbsttests in den Off Zustand zu gelangen. Da der Selbsttest in der Umsetzung des zugehörigen Zustands weggelassen wird, wird diese Kante dort auch weggelassen.

13.2 Umsetzung des Zustandmodells

Da keine Unterzustände definiert werden können, mussten die Effekte der drei großen Zustände mit ihren Subzuständen kombiniert werden. Dies bedeutet, dass alle Kanten, die aus dem Zustand herausführen, selbst an jeden Subzustand hinzugefügt werden müssen. Führen Kanten zu einem übergeordneten Zustand muss entschieden werden, an welchen Subzustand die Kante geknüpft wird. Der Zustand Off ist am einfachsten, da er keine Unterzustände hat. Er wird beim Modellieren so übernommen. Der Zustand Standby muss aufgelöst werden. Alle eingehenden Kanten werden an den Zustand Selbsttest geknüpft. Im Zustand Active werden alle eingehenden Kanten an den Zustand SpeedControl gehängt. Der Zustand Fehler wird für das erste Modell weggelassen.

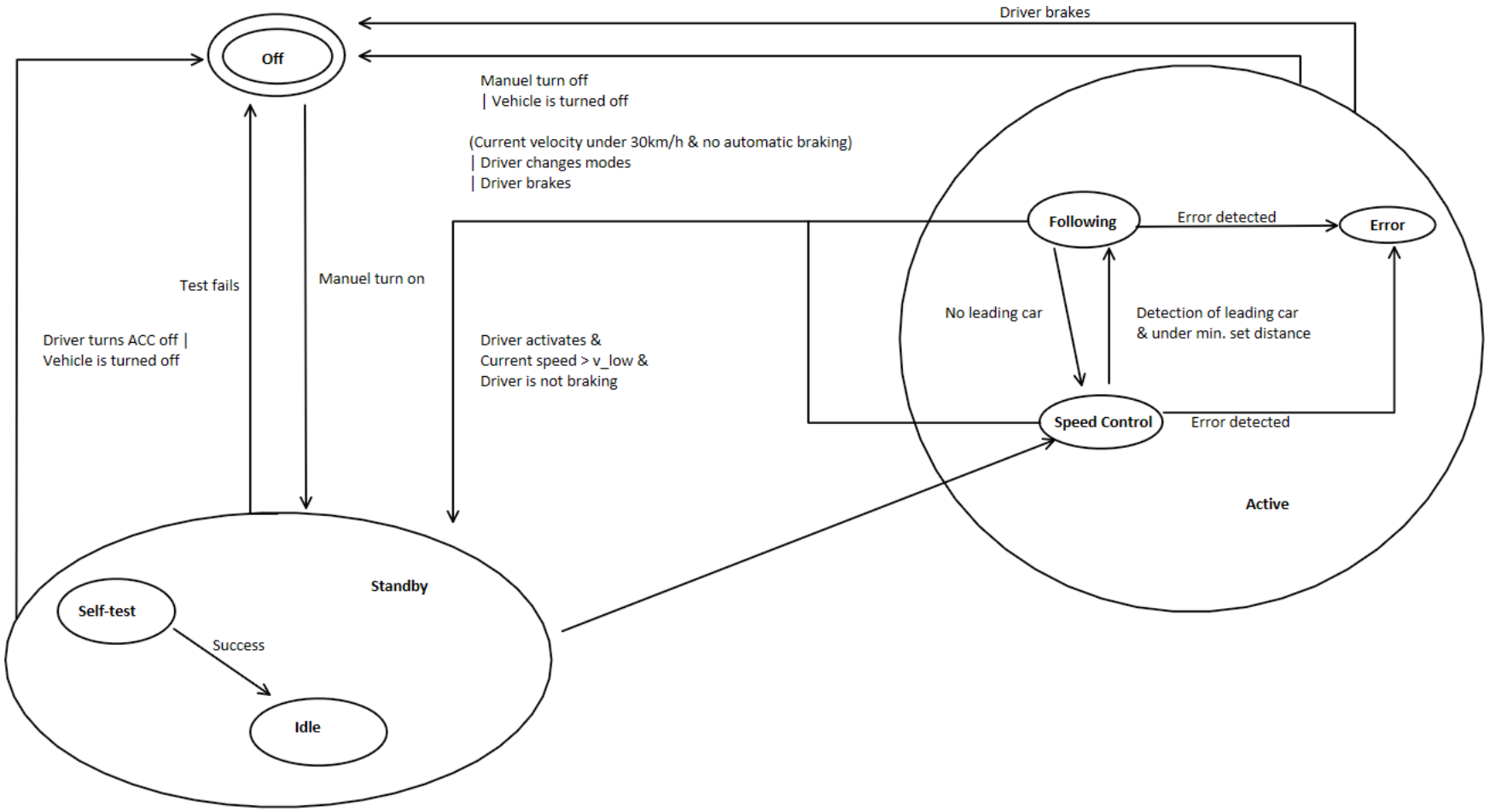


Abbildung 13.1: Modellierung eines eigenen Zustandsdiagramm für ein ACC, welches später in der Software modelliert werden soll.

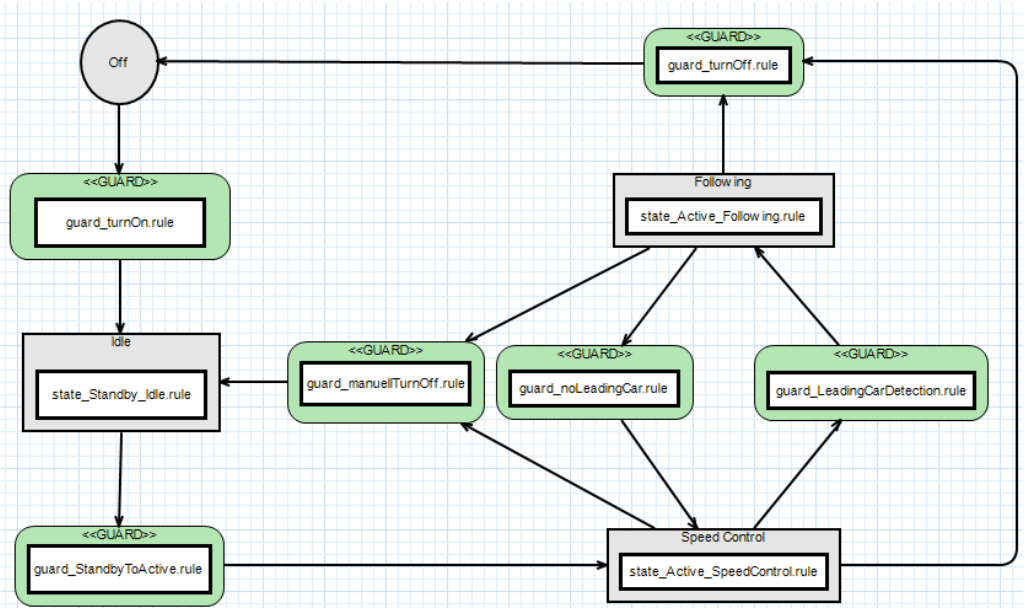


Abbildung 13.2: Umsetzung des Zustandsdiagramm aus Abbildung 13.1 in der Software. Man beachte, wie ähnlich die Diagramme sind.

Zunächst werden die Zustände modelliert (vgl. Abbildung 13.2). Der Zustand Off ist in der Software ein spezieller Zustand und existiert bereits. Die Zustände Idle und Selbsttest sind leere Platzhalter. Der Zustand Fehler wird vorerst weggelassen, da keine Fehler existieren, die in den ersten Modellierungen überprüft werden sollen.

Von dem Zustand Off geht eine Kante in den Zustand Idle. Der Guard prüft, ob der Fahrer das ACC eingeschaltet hat. In diesem Fall bedeutet das, es wird geprüft, ob das Eingangssignal für das Einschalten auf „true“ gesetzt ist.

Der Zustand Idle führt keine Funktionen aus. Hier hat der Fahrer die Möglichkeit Einstellungen vorzunehmen, die später im Zustand Active genutzt werden. Ist die Konfiguration abgeschlossen, kann in den aktiven Zustand gewechselt werden. Dabei wird der Zustand Idle mit dem Zustand SpeedControl verbunden. In den Zustand Active kann gewechselt werden, wenn die Mindestgeschwindigkeit von $7m/s$ überschritten ist, der Fahrer nicht bremst (Bremspedal-Eingangssignal ist neutral) und erneut das Eingangssignal für das Einschalten auf „true“ gesetzt wird. Der Guard für den Wechsel ist einfach zu realisieren. In diesem Guard wird mit einem einfachen IF geprüft, ob die aktuelle Geschwindigkeit des Fahrzeugs kleiner ist als die Mindestgeschwindigkeit. Mit einem AND kann dann geprüft werden, ob das Bremspedal neutral ist, ein Ergebnis NOT kehrt den booleschen Wert um. Ein erneutes AND mit den zwei bestimmten booleschen Werten und der Abfrage, ob das Einschaltsignal aktiv ist, ergeben das Ergebnis des Guards. Es werden also zwei ANDs, ein IF und ein NOT aus dem Werkzeugkasten genutzt. Die Umsetzung ist sehr einfach

gehalten und kann von jeder Person mit Kenntnissen der booleschen Logik gelöst werden. Die ursprüngliche Guardbeschriftung kann beinahe eins zu eins umgesetzt werden.

Im aktiven Zustand müssen die zwei Zustände SpeedControl und FollowingControl modelliert werden. Die beiden Zustände können durch jeweils eine Kante erreicht werden. Der Guard von SpeedControl nach FollowingControl wird aktiv, wenn ein Fahrzeug erkannt wird. Damit der Guard eine Entscheidung treffen kann, werden auch hier zunächst zwei IFs genutzt, um boolesche Werte zu bestimmen, gefolgt von einem AND zur Verknüpfung der Werte.

Um dann von FollowingControl nach SpeedControl zu gelangen, wird der gleiche Guard mit negiertem Ergebnis verwendet.

Um das ACC auszuschalten, werden die Zustände mit zwei verschiedenen Guards an den Zustand Off und den Zustand Idle verbunden. Um von den aktiven Zuständen in den Zustand Idle zu gelangen, reicht es, wenn der Fahrer stark bremst. Dieser Guard lässt sich mit einem einfachen IF modellieren. Außerdem wird mit einem IF geprüft, ob das Fahrzeug langsamer als $30km/h$ fährt und mit einem AND kann dann geprüft werden, ob kein automatischer Bremsvorgang im Gange ist. Die beiden Ergebnisse (langsamer als $30km/h$ und kein Bremsen oder Fahrer bremst) werden mit einem OR verknüpft, die Auswertung des ORs liefert das Ergebnis des Guards. Soll das ACC vollständig ausgeschaltet werden, muss geprüft werden, ob der Fahrer das Einschaltsignal auf „1“ gesetzt hat (z. B. durch das Betätigen eines Knopfes am Lenkrad).

Nun kann der Datenfluss für die aktiven Zustände SpeedControl und FollowingControl modelliert werden. Der Datenfluss im Zustand SpeedControl (vgl. Abbildung 13.3) zu modellieren, wird eine neue Rule angelegt. Diese bestimmen unter anderem den Datenfluss, werden aber auch bei den Guards verwendet. Um nun die Geschwindigkeit zu regulieren, wird ein einfacher PID-Regler, der als Unterregel modelliert wurde, genutzt. Das CINCO Produkt stellt bereits einen PID-Regler zur Verfügung. Dieser muss nur noch konfiguriert werden. Die Werte p, i und d können später durch Versuche festgestellt werden. Der Regel werden dann die aktuelle Geschwindigkeit des Fahrzeugs durch CurrentSpeed und die Zielgeschwindigkeit durch SetSpeed übergeben. Das Ergebnis des PID-Reglers wird dann verglichen mit dem Wert für das Gaspedal. Wenn der Fahrer stärker beschleunigt als der PID-Regler, dann wird dieser übernommen und als Ausgabe an das Fahrzeug weitergereicht. Dies geschieht in dem unteren Output Knoten (vgl. Abbildung 13.3). Wenn die Beschleunigung durch den Fahrer in dem LessOrEqual Knoten als kleiner als die PID Ausgabe ausgewertet wird, wird über die folgende Decision umgeleitet und eine andere Ausgabe als vorher gewählt. Diesmal wird als neue Beschleunigung die PID Ausgabe gewählt. Mit fünf Knoten kann also mit der Software ein nicht triviales Problem modelliert und gelöst werden.

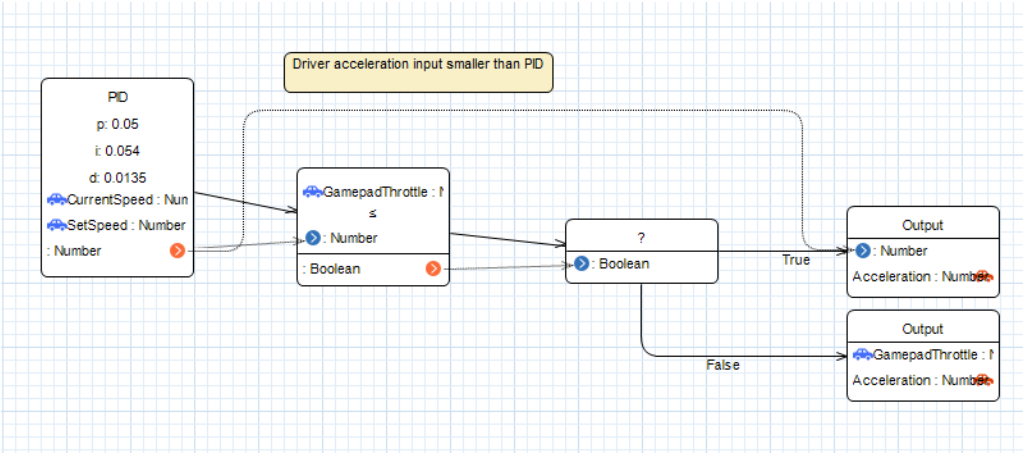


Abbildung 13.3: Die Darstellung zeigt das Datenflussmodell für den Zustand SpeedControl.

Um den Datenfluss im Zustand FollowingControl (vgl. Abbildung 13.4) zu modellieren muss zunächst der Abstand zum vorausfahrenden Fahrzeug berechnet werden. Mit dem Abstand zum vorausfahrenden Fahrzeug (DistanceFront), der aktuellen Geschwindigkeit des Fahrzeuges (CurrentSpeed) und dem gewünschten Abstand zum Vorausfahrenden wird der Abstand zwischen beiden Fahrzeugen berechnet. Um die Zielgeschwindigkeit zu berechnen, muss auf gleiche Weise noch die Geschwindigkeit des vorausfahrenden Fahrzeugs aus der relativen Geschwindigkeit und der Geschwindigkeit des Fahrzeugs unter Nutzung einer Division berechnet werden. Die Multiplikation des berechneten Abstands und die Geschwindigkeit des vorausfahrenden Fahrzeugs liefern die neue Geschwindigkeit. Die kleinere Geschwindigkeit aus der eingestellten und der berechneten Geschwindigkeit ist die Zielgeschwindigkeit des Fahrzeugs und kann mit der Nutzung des Minimum Knotens bestimmt werden. Ein PID-Regler berechnet dann die Beschleunigung, die benötigt wird, um von der aktuellen Geschwindigkeit auf die Zielgeschwindigkeit zu gelangen. Die Beschleunigung ist dann die Ausgabe der Rule. Für Bremsen ist der Wert negativ, für Beschleunigung ist der Wert positiv. Dieser Ansatz ist noch aufwendiger und mathematisch komplexer als der SpeedControl Datenfluss. Diesmal wurden acht Knoten benötigt. Die Knoten selbst bieten grundlegende mathematische Funktionen und Operationen an und dank der grafischen Repräsentation wird die verwendete Mathematik einfach und gut lesbar umgesetzt. Damit ist dann ein ganzes ACC modelliert. Mit ein wenig Übung im Umgang mit der Software ist ein ACC schnell zusammengestellt und formale Modelle wie das zuvor definierte Zustandsdiagramm können nahezu nahtlos umgesetzt werden.

13.3 Umsetzung Monitoring

Obwohl bei dem eigentlichen ACC-Zustandsmodell bereits bei der Modellierung auf die Einhaltung der *ACC-ISO-Norm* [13] geachtet wurde, gehören zu dem Modell außerdem

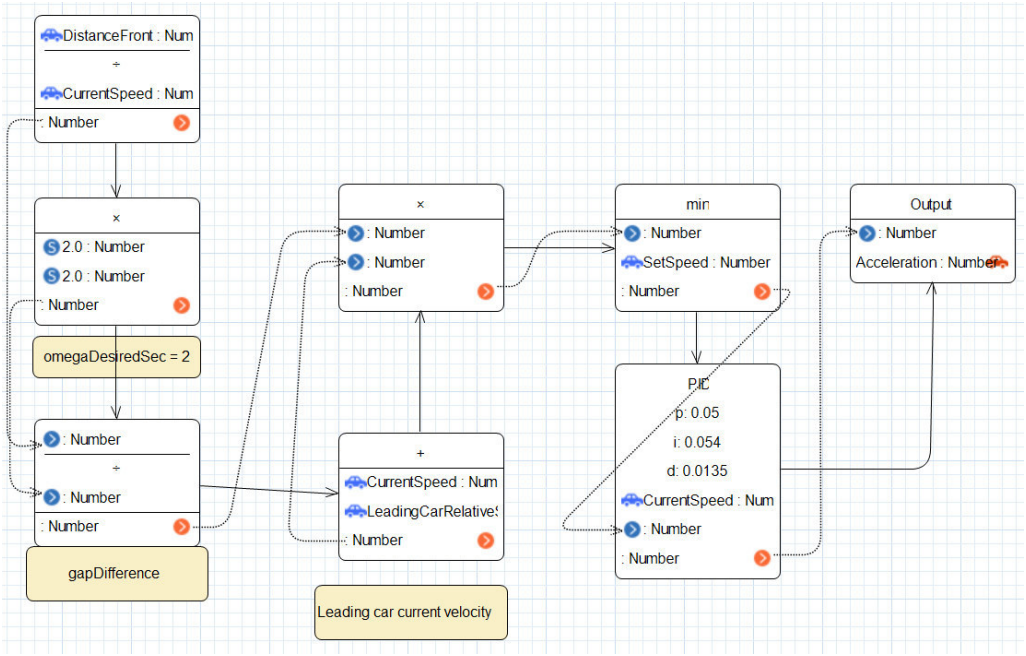


Abbildung 13.4: Die Darstellung zeigt das Datenflussmodell für den Zustand FollowingControl.

noch einige Monitore, die während der Laufzeit das Verhalten des ACCs überwachen. Die Umsetzung der *ACC-ISO-Norm* in diesen Monitoren wird im Folgenden erklärt.

Die Monitore wurden in der eigens dafür entwickelten und bereits erläuterten textuellen Monitor-Sprache entwickelt. Obwohl mehrere verschiedene Monitore mit jeweils mehreren Tests zu einer Configuration zusammengefasst werden können, wurden die Monitore, aufgeteilt nach Zuständen des ACC-Zustandsmodells, der Übersichtlichkeit wegen in unterschiedliche Dateien aufgeteilt.

13.3.1 Zustand Off

Im Off-Zustand müssen eigentlich nur zwei Fälle überprüft werden: Wird der On-Button gedrückt, dann muss das ACC in den idle-Zustand wechseln. Bei allen anderen Aktionen (gedrückten Buttons o.Ä.) verbleibt das ACC im Off-Zustand (siehe Code-Ausschnitt 13.1).

13.3.2 Zustand Idle

Für den Idle-Zustand gibt es 5 Tests. In diesem Zustand muss die Variable SystemOn immer auf True gesetzt sein, da das ACC zwar noch nicht aktiv ist, aber eingeschaltet. Außerdem muss das ACC in den Off-Zustand wechseln, sobald der SystemOn-Button gedrückt wird, und in Idle bleiben, wenn der „Distanz erhöhen“- oder „Geschwindigkeit/-Distanz verringern“-Button gedrückt wird. In Code-Ausschnitt 13.2 werden diese Fälle nicht gezeigt, da sie recht trivial sind und denen im Off-Zustand stark ähneln. Das Code-Ausschnitt 13.2 zeigt jedoch die Umsetzung der beiden Fälle, bei denen das ACC aktiviert

```
def off_1 = is not states.Idle[1] and
      is not states.Following[1] and
      is not states.SpeedControl[1]

def Test off_to_idle{
  Conditions{
    off_1, SystemOnButton[1]
  }
  Invariants{
    idle
  }
}

def Test off_stay_in_off{
  Conditions{
    off_1,
    SystemActiveButton[1] or
    DecrementSetDistanceButton[1] or
    IncrementSetDistanceButton[1] or
    DecrementSetSpeedButton[1] or
    IncrementSetSpeedButton[1]
  }
  Invariants{
    off
  }
}
```

Code-Ausschnitt 13.1: Tests des Off-Zustands

werden soll. In beiden Tests wird überprüft, ob sich das System im letzten Zyklus im Zustand Idle befand, einer der beiden „Aktivieren“-Buttons gedrückt wurde, die aktuelle Geschwindigkeit hoch genug war und ob ein Bremsvorgang stattfand. Dabei ist es nicht von Bedeutung, welcher der beiden „Aktivieren“-Buttons gedrückt wurde, also ob die aktuelle Fahrzeuggeschwindigkeit eingestellt oder die zuletzt gespeicherte wieder aufgenommen werden soll. Interessant sind dabei hauptsächlich die letzten beiden Punkte. Wenn einer dieser Punkte nicht erfüllt war, muss sich das System im aktuellen Zyklus weiter im Idle-Zustand befinden (unterer Test), ansonsten muss ein Wechsel zu SpeedControl erfolgt sein (oberer Test).

```
def idle_1 = is states.Idle[1]

def Test idle_to_speed{
  Conditions{
    idle_1,
    IncrementSetDistanceButton[1] or SystemActiveButton[1],
    CurrentSpeed[1] > sharedMemory.ACC_DefaultMinACCSpeed_m_s,
    Brake[1] == 0
  }
  Invariants{
    speedControl
  }
}

def Test idle_stay_in_idle_conditions{
  Conditions{
    idle_1,
    SystemActiveButton[1] or IncrementSetDistanceButton[1],
    CurrentSpeed[1] <= sharedMemory.ACC_DefaultMinACCSpeed_m_s or Brake[1] > 0
  }
  Invariants{
    idle
  }
}
```

Code-Ausschnitt 13.2: Tests des Idle-Zustands

13.3.3 Zustände SpeedControl und FollowingControl

Die Tests für die Zustände SpeedControl und FollowingControl lassen sich größtenteils zu einem Aktiv-Zustand zusammenfassen. Dieser wird im nächsten Unterabschnitt erklärt. Die für diese beiden Zustände spezifischen Tests beziehen sich ausschließlich auf den Wechsel zwischen den Zuständen, wie Code-Ausschnitt 13.3 zeigt. Der Wechsel von FollowingControl zu SpeedControl erfolgt immer dann, wenn kein vorausfahrendes Fahrzeug mehr erkannt wird oder dieses weiter als der eingestellte Mindestabstand entfernt ist (oberer Test). Umgekehrt wird von SpeedControl nach FollowingControl gewechselt, wenn es ein vorausfahrendes Fahrzeug gibt und dieses dem eigenen Fahrzeug näher ist, als der eingestellte Mindestabstand es erlaubt (unterer Test).

13.3.4 Zustand Aktiv

Code-Ausschnitt 13.4 zeigt eine Auswahl der Tests, die sowohl auf den SpeedControl- als auch auf den FollowingControl-Zustand anwendbar sind. In den ersten beiden Tests wird das Verhalten des ACCs überprüft, wenn der Fahrer selbst bremst oder Gas gibt. Wenn der Fahrer bremst und die von ihm aufgebrachte Bremskraft die des ACCs übersteigt, dann wird das ACC deaktiviert und wechselt in den Idle-Zustand. Wenn jedoch der Fahrer Gas gibt und die Beschleunigung des Fahrers die des ACCs übersteigt, dann darf das ACC nicht in

```
def Test following_to_speed{
  Conditions{
    following_1,
    (HasLeadingCar[1] and TimeDistanceFront > SetDistance) or !HasLeadingCar[1]
  }
  Invariants{
    speedControl
  }
}

def Test speed_to_following{
  Conditions{
    speedControl_1,
    HasLeadingCar[1],
    TimeDistanceFront < SetDistance
  }
  Invariants{
    following,
    ObstacleDetectedWarning
  }
}
```

Code-Ausschnitt 13.3: Tests der SpeedControl- und FollowingControl-Zustände

den Idle-Zustand wechseln, sondern muss weiterhin in einem der beiden aktiven Zustände sein.

Der dritte Test, der in Code-Ausschnitt 13.4 gezeigt wird, überprüft, ob das ACC selbstständig die Geschwindigkeit reduziert, wenn der Abstand zum vorausfahrenden Fahrzeug zu gering wird. Ist dies der Fall, müssen sowohl die aktuelle Geschwindigkeit, als auch die aktuelle ACC-Beschleunigung geringer sein, als in dem Zyklus davor.

Weiterhin wird in dem Monitor überprüft, dass die 4 Buttons zum Reduzieren bzw. Erhöhen der eingestellten Geschwindigkeit und des eingestellten Abstands sowie der Button zum Setzen der aktuellen Geschwindigkeit als Zielgeschwindigkeit die gewünschte Wirkung erzielen und die neuen Werte tatsächlich in den Speicher des ACCs übernommen werden.

13.3.5 Grenzen der Monitore

Die gezeigten bzw. erklärten Monitore decken nicht sämtliche Vorgaben innerhalb der *ACC-ISO-Norm* ab. Wie das ACC-Zustandsmodell selbst, bewegen sich auch die Monitore in den Grenzen des in diesem Rahmen Umsetzbaren. Für einige Anforderungen der *ACC-ISO-Norm* sind die Werte, die das generierte ACC liefert, zu ungenau, sodass es sich nicht lohnen würde, dafür Monitore zu schreiben. Zusätzlich dazu sind einige Anfor-

```
def active_1 = is states.Following[1] or is states.SpeedControl[1]

def Test active_to_idle_brake{
Conditions{
active_1,
Brake[1] > sharedMemory.ACC_Break[1] or
CurrentSpeed[1] < sharedMemory.ACC_DefaultMinACCSpeed_m_s
}
Invariants{
idle
}
}

def Test active_stay_in_active_driver_acceleration{
Conditions{
active_1,
Throttle[1] >= sharedMemory.ACC_Throttle[1]
}
Invariants{
active
}
}

def Test adjust_distance_front{
Conditions{
DistanceFront[1] <= SetDistance[1] * CurrentSpeed[1]
}
Invariants{
CurrentSpeed < CurrentSpeed[1],
sharedMemory.ACC_Throttle < sharedMemory.ACC_Throttle[1]
}
}
```

Code-Ausschnitt 13.4: Tests des aktiven Zustands

derungen zur Demonstration der Funktionsfähigkeit des entwickelten ACCs relevanter als andere. Beispielsweise sieht die *ACC-ISO-Norm* vor, dass die Änderungsrate der negativen Beschleunigung einen gewissen Wert nicht übersteigen darf. Diese Anforderung hat zwar durchaus einen praktischen Nutzen auf echten Fahrzeugen, für unsere Zwecke ist es aber zum Beispiel wichtiger, dass die notwendigen Zustandswechsel korrekt vonstattengehen. Außerdem gibt es keine Monitore, die z.B. das Verhalten in Kurvenfahrten überprüfen, da das entwickelte ACC nicht für Kurvenfahrten ausgelegt ist. Wenn das ACC entsprechend erweitert würde, könnten diese aber ergänzt werden.

Kapitel 14

Verifikation

Dieses Kapitel behandelt zwei Aspekte der Verifikation: zum einen die Auswertung der Testfälle in der Simulationsumgebung VTD mit der von uns dazu entwickelten DSL SzenarienDSL und zum anderen die Verifikation über reale Fahrten mit den Modellautos in einer speziellen Laborumgebung. Abschnitt 14.2 gibt einen Überblick über die dazu notwendige Umgebung im Innovationslabor und über die Erkenntnisse, die daraus gewonnen werden konnten.

14.1 Evaluation der Testfälle

Nun lassen sich mit Hilfe der Monitore und der *ADAS-Live-Visualisierung* die modellierten Testfälle prüfen. Die Simulation bzw. das Ausführen der gesamten Testsuite in Echtzeit dauert etwa eine Stunde. Die genaue Länge hängt dabei von der Modellierung der einzelnen Testfälle ab. Dabei können alle Testfälle vollautomatisch hintereinander ausgeführt werden. Dabei fiel vor allem auf, dass alle Testfälle unfallfrei absolviert wurden und die meisten Vor- bzw. Nachbedingungen erfolgreich erfüllt wurden. Allerdings sind auch einige negative Aspekte aufgefallen. Unter den Monitoren befinden sich einige, die die aus der *ACC-ISO-Norm* gegebenen Spezifikationen, wie beispielsweise die maximal erlaubte Beschleunigung oder Verzögerung, überwachen. Diese Monitore sind in beinahe allen Testfällen unerfüllt, denn das innerhalb der Projektgruppe entwickelte ACC hält die Grenzen für die maximale Beschleunigung und die maximale Verzögerung nicht ein, wie es bereits in Abschnitt 12.3 gezeigt wurde. Außerdem sind einige Testfälle vorhanden, bei denen die *TimeDistance* bei 1,0 s liegt und die Abstände zwischen den Fahrzeugen so gering sind, dass mit dem in der Norm spezifizierten maximalen Wert für die Bremskraft ein Auffahrunfall erzeugt worden wäre. Da das entwickelte ACC diese Limitierung nicht aufweist, sind in den Test dadurch keine Auffahrunfälle entstanden.

Mit diesen Möglichkeiten konnten Zustandswechsel des ACC erfolgreich getestet und validiert werden. Zu keinem Zeitpunkt der Testfälle befand sich das ACC für einen längeren Zeitraum in einem falschen Zustand und somit waren die Vor- und Nachbedingungen der Testfälle erfüllt. Jedoch kam es in manchen Fällen zu einem „flackern“der Monitore. Das bedeutet, dass er Monitor in kurzen Zeitabständen erfüllt und nicht aktiv oder unerfüllt und nicht aktiv war. Dieses Phänomen trat auf, da das ACC in gewissen Brems- oder Beschleunigungsvorgängen eingestellte Abstände kurzzeitig unter- bzw. überschritten hat.

Des Weiteren sind erst durch die Simulation und die Validierungsmethoden während des Entwicklungsprozesses einige Fehler aufgefallen. Bei diesen Fehlern handelt es sich größtenteils um Modellierungsfehler am ACC. Beispielsweise gab es den Fehler, dass das Ego-Fahrzeug im Zustand *Following* über seine eingestellte Geschwindigkeit hinaus beschleunigt hat, wenn das vorausfahrende Fahrzeug dies ebenfalls tat. Dies lag an dem Versuch den Zeitabstand auch über die eingestellte Geschwindigkeit beizubehalten. Ein anderer Fehler war, dass sich das ACC während eines automatischen Bremsvorgangs ausgeschaltet hat, ohne den Bremsvorgang zu Ende durchzuführen.

Alles in allem hat sich das Testen durch Simulationen mit Hilfe von Monitoren als sehr hilfreich erwiesen. Auch während des Entwicklungsprozesses oder mit einem Prototyp des Systems ist dieses Vorgehen geeignet, um frühzeitig Fehler zu entdecken oder Funktionen bzw. Eigenschaften des Systems zu validieren. In unserem Fall wurden die oben erwähnten Modellierungsfehler erst dadurch sichtbar und konnten behoben werden.

14.2 Realfahrten im Innovationslabor

Unter einer Realfahrt ist der Betrieb des Modellautos zu verstehen. Im einfachsten Fall ist das Modellauto aufgebockt, sodass es weiter stationär betrieben werden kann, das heißt, dass Bildschirm und Eingabegeräte direkt am Auto verbleiben.

Während der Projektlaufzeit war es der Projektgruppe aber auch möglich, auf räumliche sowie technische Ressourcen des „Innovationslabor Hybride Dienstleistungen in der Logistik“ [1] zurückzugreifen, um Realfahrten durchzuführen, die in normalen Seminarräumen nicht möglich sind.

Zunächst wird die Technik der Versuchshalle skizziert, anschließend genauer auf die Lidarvisualisierung eingegangen. Abschließend werden insbesondere Herausforderungen von realen ACC-Fahrten erläutert.

[1] http://www.innovationslabor-logistik.de/

14.2.1 Versuchshallenaufbau und Fahrzeugkonfiguration

Für Realfahrten unabdingbar ist eine möglichst große befahrbare Versuchsfläche. Die Versuchshalle des Innovationslabors bietet eine prinzipielle Versuchsfläche von 466 m^2. Dadurch war ein Ausfahren des Fahrzeugs auf längeren Geraden möglich.

Während solcher Realfahrten wurde der auf dem Auto montierte Rechner per Akku betrieben, welcher eine Betriebszeit von gut 90 min ermöglicht. Da die Halle weiterhin über ein WLAN-Netz verfügt, konnte über einen weiteren Rechner das Fahrzeug via SSH gesteuert werden. Es ergab sich dadurch die Möglichkeit, ADTF-Konfigurationen oder Plugins zügig abzuändern, während das Fahrzeug noch auf der Versuchsfläche stand. Auch fortlaufendes Auswerten von Log-Ausgaben war möglich. Umlaufzeiten während der Entwicklung blieben so relativ kurz.

Besonderheit der Versuchshalle ist ihr Motion-Capturing-System sowie ihr Laserprojektionssystem. Mithilfe des Motion-Capturing-Systems lässt sich die Position markierter Objekte innerhalb des Beobachtungsbereichs millimetergenau erfassen. Das Laserprojektionssystem ermöglicht das Projizieren von Formen und Markierungen auf den Hallenboden und unterstützt die Echtzeitauswertung von laufenden Versuchen.

Das Motion-Capturing-System setzt sich aus 38 Kameras zusammen und ermöglicht das Erfassen von auf Infrarot-Licht reagierenden Markern in einem Beobachtungsbereich von 23 m x 13 m x 4 m. Objekte werden mit mehreren dieser Marker versehen und lassen sich so von den Kameras erfassen. Angegeben werden kann dabei nicht nur die Position im Koordinatensystem der Halle, sondern auch ihre Ausrichtung. Im einzelnen besteht das System aus 8 Kameras mit einer Auflösung von 5 Megapixeln sowie aus 30 Kameras mit 2,2 Megapixeln Auflösung, welche im Deckenbereich der Versuchsfläche montiert sind. Möglich ist das Tracking von mehr als 100 Objekten gleichzeitig bei einer maximalen Bildrate von 330 Hz. Die Kameralatenz liegt bei 4,7 ms.

Das Laserprojektionssystem besteht aus 8 an der Decke befestigten Lasern und deckt einen Projektionsbereich von 23 m x 13 m ab. Das System kann nicht beliebig viele Pixel gleichzeitig darstellen, sondern fährt in erster Linie definierbare Formen ab. Bei der Darstellung bestimmter Objekte kommt es aufgrund der begrenzten Projektionsrate zum Flackern. Beim Betreten des Projektionsbereichs während aktivierter Laserprojektion muss eine Schutzbrille getragen werden, daher verfügt das System zudem über Not-Aus-Schalter, die beim unbefugten Betreten von Personen manuell ausgelöst werden können.

Um das Modellauto tracken zu können, mussten zunächst Marker auf das Fahrzeug aufgebracht werden. Diese wurden so auf das Auto geklebt, dass sie möglichst von allen Richtungen aus von Kameras erfasst werden können, also nicht durch andere Fahrzeugaufbauten verdeckt werden. Darüber hinaus muss das Fahrzeug mit den so aufgebrachten Markern ins

Motion-Capturing-System eingelernt werden. Dazu wird das Fahrzeug in definierter Richtung in den Koordinatenursprung der Versuchshalle gestellt. Der Mittelpunkt des Objekts kann in der Capturing-Software konfiguriert werden. Gewählt wurde dabei der Mittelpunkt des Lidars. Das System lieferte so zuverlässig die Position des Lidars innerhalb des Beobachtungsbereichs.

Wurde die Position der Marker auf dem Fahrzeug geändert, beispielsweise durch Neuanbringen einzelner Marker, so bedurfte es einem erneuten Einlernen, da die Relativabstände der einzelnen Marker verändert wurden.

14.2.2 ACC-Fahrt

Während Versuchsfahrten mit zwei Modellautos gelang es kaum, eine ACC-Fahrt, also ein Erreichen des *Following*-Zustands (siehe Kapitel 13), durchzuführen. Problematisch war das Ausrichten der Fahrzeuge hintereinander. Dies stellte sich herausfordernder dar als angenommen, da kleinste Lenkbewegungen bei einer manuellen Steuerung dafür sorgten, dass das vorausfahrende Fahrzeug nicht mehr oder nicht optimal im Sensorfeld fuhr. Dadurch schwankten die vom Sensor bzw. vom Lidar-Distance-Filter ermittelte Abstandswerte extrem. Die Folge waren permanente Zustandswechsel des ACCs. Die in Unterabschnitt 12.1.4 vorgestellte Controller-Steuerung mit geringerer Intensität konnte das Problem abschwächen. Gleichzeitig wurde durch den Umstieg vom Xbox 360 zum Xbox One Controller die haptische und visuelle Rückmeldung stark erschwert. Der aktuelle Zustand des ADAS war während der Fahrt nur via MQTT oder Debugausgaben auf der Konsole in Erfahrung zu bringen. Nicht wie vorher per LED-Visualisierung direkt auf dem Gamepad.

Ein weiteres Hindernis für alle sensorgestützten Realfahrten war die in Unterabschnitt 14.2.3 näher erwähnte schlechte Fahrzeugerkennung. Modellfahrzeuge werden vom Lidar kaum erkannt. Das Anbringen von Papier- bzw. Papp-Aufbauten konnte kaum Abhilfe schaffen. Notwendig wäre hier eine deutlich stärkere Fokussierung der Projektgruppe auf die Auswertung der vom Lidar gelieferten Rohdaten, um möglicherweise bessere Ergebnisse erzielen zu können.

Des Weiteren stellten die begrenzten Maße der Versuchsfläche beträchtliche Probleme dar. Da die Sensorwertverarbeitung in dem von der Projektgruppe entwickelten Distanz-Filter keine Kurvenfahrten berücksichtigt, aber die Strecke einer einzelnen Geradeausfahrt für umfangreiche Tests des ADAS zu kurz ist, sind Kurvenfahrten mit eingeschaltetem ACC zwingend nötig. Dadurch kam es wiederholt zu Problemen mit der Erkennung des vorausfahrenden Fahrzeugs und dadurch zu einem veränderten Verhalten des ACCs.

Die dargestellten Probleme sorgten dafür, dass ein für die Realfahrt angepasstes ACC-Modell modelliert wurde. Insbesondere in der *ACC-ISO-Norm* beschriebene Restriktionen für Zustandswechsel, beispielsweise Mindestgeschwindigkeiten für die Aktivierung des ACCs

wurden in dieser Modellvariante abgeschwächt, um überhaupt eine Funktion des ACC zu ermöglichen. Weiter wurde die Parameter des PID-Reglers anhand von manuellen Testfahren angepasst. Außerdem war die unmodifizierte Version des Tempomaten für eine Fahrt aufgrund der für das Modellauto großen Geschwindigkeitssprünge ungeeignet. Um dies zu korrigieren, wurde die Geschwindigkeitsschrittweite im Modell verringert. Auch fehlte ein direktes Feedback für die aktuell eingestellte Zielgeschwindigkeit sowie Zeitdistanz zum vorausfahrenden Fahrzeug. Eine für den Fahrer direkt und verzögerungslos ablesbare Information, im echten Auto über ein Multifunktionsdisplay gelöst, fehlte. Gerade bei der Erprobung, Validierung und Anpassung von prototypischen ADAS im Realbetrieb ist eine direkte Anzeige notwendig.

14.2.3 Lidarvisualisierung

Wie im vorangegangenen Unterabschnitt 14.2.2 angemerkt, ist die Datenvisualisierung bei der Erprobung und Entwicklung von ADAS äußerst wichtig. Da die Fahrzeuge selbst keine direkt Visualisierung bieten und konventionelle, textuelle Ausgaben, bspw. über eine Konsolenterminal, nur im geringen Maße brauchbar sind und die Halle des Innovationslabors die Möglichkeiten einer direkten Visualisierung über das Laserprojektionssystem verfügt, wurde davon Gebrauch gemacht.

Eine erste Umsetzung erfolgte durch den in Unterabschnitt 12.1.7 vorgestellten Lidar-Visualisierung-Filter.

Es wurde die Darstellung einzelner Hindernispunkte sowie die Darstellung von Strahlen vom Lidar aus zu den Hindernispunkten erprobt. Letztere Darstellung erwies sich als anschaulicher. Es lässt sich dadurch der Eindruck gewinnen, die dargestellten Strahlen gingen tatsächlich vom Lidar aus.

Durch die Lidarvisualisierung wurde ersichtlich, dass die Erfassung von Realfahrzeugen wie einem weiteren BFFT-Modellfahrzeug oder einem funkferngesteuerten Spielzeugauto eine große Problematik bei der Erprobung der modellierten ADAS darstellt. Vorausfahrende Fahrzeuge solcher Art erzeugen in der Regel nur sehr wenig Lidar-Hindernispunkte. Sie sind dadurch kaum von falsch-positiv erkannten Hindernispunkten zu unterscheiden.

Teil V

Zusammenfassung

Kapitel 15

Fazit und Ausblick

Das Ziel war es, mit Hilfe einer DSL eine Modellierungsumgebung für Fahrassistenzsysteme zu entwickeln. Aus den darin erstellten Modellen sollte dann ausführbarer Code generiert werden. Das Fahrassistenzsystem soll anschließend auf den uns zur Verfügung stehenden Testfahrzeugen ausgeführt werden. Weiterhin sollte es möglich sein, die Korrektheit des Modells vor der Übersetzung und während der Ausführung zu testen und zusätzlich Testszenarien zu generieren.

15.1 Fazit

Mithilfe eigens definierter DSLs ist es uns möglich, die oben genannten Ziele der Projektgruppe zu erreichen. Zum einen können die in der AutoDSL verwendeten Modelle in einen mit ADTF ausführbaren Code übersetzt werden. Zusätzlich werden Monitore mit der von uns entwickelten textuellen DSL definiert, welche wie Invarianten in der Programmierung agieren und den inneren Zustand des modellierten ADAS zu überwachen. Weiterhin ist es möglich, mit der entwickelten SzenarienDSL Testszenarien zu definieren und automatisch in der Simulationsumgebung VTD zu simulieren. Dabei kann die Einhaltung der Monitore auch im Nachgang überprüft werden.

Nach der initialen Fertigstellung der DSLs lässt sich mit deren Hilfe schneller und sicherer ein ADAS entwickeln. Die AutoDSL ermöglicht es, das zu entwickelnde ADAS ohne Programmiererfahrung, bspw. durch einen Maschinenbauingenieur, zu erstellen. Der Modellierer wird durch syntaktische und semantische Überprüfung des Modells während des Erstellungsprozesses unterstützt. Weiterhin können Monitore definiert werden, um den Systemzustand zu überwachen. So ist es mit der AutoDSL möglich ein ADAS effizienter und sicherer zu entwickeln, als es mit einer direkten Implementierung möglich wäre. Durch kleine Änderungen am Generator kann schnell auf Änderungswünsche oder Fehler im ADAS reagiert werden, ohne das Modell anpassen zu müssen. Die SzenarienDSL ermöglicht es

Testszenarien zu definieren, die in einer Simulationsumgebung ausgeführt werden. Diese Testszenarien sind einfach zu modellieren und leicht zu modifizieren, so dass mehrere Varianten eines Testfalls beispielsweise mit unterschiedlichen Geschwindigkeiten erstellt werden können. Weiterhin ist es möglich, mehrere Szenarien in einer Sammlung zusammenzufassen und automatisch hintereinander auszuführen. Zusammen mit den definierten Monitoren lassen sich so verschiedene Szenarien ausführen und das Verhalten des ADAS bewerten. Die gewinnbringenden Funktionen der DSLs unterstützen einen Ersteller eines ADAS stark. Dadurch sind ADASs einfacher und effizienter zu erstellen.

15.2 Ausblick

Es gibt einige Punkte, die über den Rahmen der Projektgruppe hinausgehen und aufgrund niedriger Priorität nicht umgesetzt wurden. Dazu gehören Erweiterungen der AutoDSL, der SzenarienDSL, ADTF und des ACC-Modells, die in den folgenden Abschnitten betrachtet werden.

15.2.1 Weiterentwicklung der AutoDSL

Informationen über Fahrspuren können nicht bei der Modellierung eines ADASs verwendet werden. So gibt es zum Beispiel keine Möglichkeit zu überprüfen, ob ein Spurwechsel aktuell möglich wäre. Um die ACC-ISO-Norm [13] komplett umsetzen zu können, wären solche Daten allerdings notwendig.

Eine sinnvolle Ergänzung wäre hier also die Erweiterung der DSL um Funktionalitäten, welche es ermöglichen zusätzliche Daten über Spurinformationen oder Kurven zu verwenden und zu verarbeiten.

Außerdem unterstützt die DSL aktuell keine hierarchischen Zustände. So wird zum Beispiel in [13] ein Zustand Active definiert. Dieser fasst die beiden Zustände SpeedControl und FollowingControl zusammen. Eine Ergänzung wäre hier denkbar, wobei hier gewisse Probleme mit der Semantik entstehen können, welche im Voraus ausführlich besprochen und geklärt werden müssten. So ist zum Beispiel eine Frage, ob ausgehende *Guards* aus dem Subzustand höher priorisiert werden, oder ob die übergeordneten *Guards* höher priorisiert werden. Außerdem muss entschieden werden, ob man sich in einem übergeordneten Zustand befinden kann, ohne in einem seiner Subzustände zu sein. Eine weitere Frage ist, ob übergeordnete Zustände auch Referenzen auf *Rule*-Modelle besitzen dürfen und ob diese auch durchgeführt werden, wenn man sich in einem Subzustand und nicht nur im übergeordneten Zustand selbst befindet. Zuletzt ist noch zu klären, ob übergeordnete Zustände sich merken können sollen, welcher ihrer Subzustände zuletzt aktiv war und ob dieser bei einem Übergang in den übergeordneten Zustand wieder aktiviert wird, oder ob es eine Möglichkeit zur Definition eines Startsubzustands geben können soll.

Bei *Rule*-Modellen wird vom Nutzer ein expliziter Kontrollfluss vorgegeben, hier stellt sich die Frage, ob es zielführend wäre, diesen über den Datenfluss implizit zu schlussfolgern, um somit durch das Weglassen nicht benötigter Kanten die Modelle übersichtlicher zu gestalten.

Beim Monitoring wäre interessant zu forschen, ob es möglich ist, für definierte Tests und Monitore automatisch entsprechende Szenarien für Grenzfälle zu generieren, welche das Verhalten eines ADASs durch Simulationen testen. Außerdem ist die Syntax im Moment nicht sehr natürlichsprachlich und an einigen Stellen etwas ungünstig gewählt, hier könnten gut weitere Alternativen getestet und verglichen werden. Des Weiteren können aktuell keine Realzeitangaben verwendet werden, hier wäre eine Ergänzung, welche Konstrukte wie „Wenn vor 100ms [...], dann [...]“ oder „[...] Geschwindigkeit vor 2 Sekunden [...]“ hinzufügt, sehr sinnvoll. Aktuell werden historische Werte nur über Zyklen der Zustandsmaschine des ADAS gehandhabt.

15.2.2 Weiterentwicklung des ACC-Modells

Im Laufe der Projektgruppe wurde die AutoDSL immer im Hinblick auf ein modellierbares ACC entwickelt. Von Anfang an war ein lauffähiges ACC mit zusätzlicher Funktionalität als *Proof-Of-Concept* angedacht.

Das ACC++-Modell enthält neben den Kernfunktionalitäten eines ACC unter anderem einen Notbremsassistenten. Dadurch erhöht sich das Sicherheitsverhalten, gleichzeitig wird jedoch gegen maximale Bremsbeschleunigungsvorschriften der ACC-ISO-Norm verstoßen.

Weiterhin kann unser ACC-Modell, um Unterstützung von Fahrspuren erweitert werden und dadurch beispielsweise einen Spurhalteassistenten bereitstellen.

15.2.3 Weiterentwicklung der SzenarienDSL

Zur Unterstützung von alternativen Simulationsumgebungen könnten neue Generatoren angelegt werden.

Weiterhin kann über eine Verwendung der SzenarienDSL für die Generierung von realen Fahrszenarien nachgedacht werden. Diese wären aufgrund von physikalischen Einschränkungen nur halbautomatisch ausführbar. So könnten Steuerungssignale für automatisch gesteuerte Testfahrzeuge generiert werden.

Zusätzlich kann eine automatische Szenarien-Generierung untersucht werden. Dabei können z.B. bestimmte Grenzwerte für Beschleunigungen und Bremsungen angegeben werden und mehrere Szenarien innerhalb dieser Intervalle erzeugt werden. Dabei könnten beispielsweise bestimmte Grenzwerte aus den Monitoren abgeleitet und für die Szenariengenerierung verwendet werden.

Darüber hinaus können im Augenblick lediglich Szenarien mit Autos erzeugt werden. Eine Erweiterung der Fahrzeugelemente um weitere Fahrzeugklassen, wie z.B. Fahrräder, Motorräder, LKW und Busse, wäre wünschenswert. Eine Erweiterung des Generators ist momentan recht einfach umzusetzen, da VTD bereits weitere vordefinierte Fahrzeugtypen besitzt.

15.2.4 Weiterentwicklung ADTF

Grundsätzlich muss die Funktionsweise der in Abschnitt 12.1 vorgestellten Filter in einem realitätsnäheren Umfeld weiter erprobt werden, da die stattgefundenen Testfahrten in den Hallen des Innovationslabors aufgrund von Kapazitätsdefiziten sehr kurz ausgefallen sind. Bisherige Tests wurden überwiegend mit aufgebockten Fahrzeugen durchgeführt, sodass eventuelle Einflüsse des dynamischen Fahrverhaltens unentdeckt blieben. Dazu zählen insbesondere Einschränkungen der Sensorik während der Fahrt.

Neben der Erprobung und Evaluation der bereits entwickelten Funktionalitäten ist die Implementierung weiterer Funktionen denkbar, welche im Folgenden vorgestellt werden.

Distanzerkennung bei Kurvenfahrten

Wie in Abbildung 15.1 dargestellt, könnte der Ansatz zur Berechnung der Distanz zu einem Hindernis erweitert werden, um auch bei Kurvenfahrten sinnvolle Distanzwerte zu erzeugen. Dazu könnten die einzelnen Buckets skaliert und in Richtung der Kurve verschoben werden. Im oben dargestellten Beispiel hätte keiner der verschobenen Buckets die Mindestzahl an Hindernispunkten, sodass kein Hindernis detektiert würde.

Einbeziehen der Tiefenkamera

Zusätzlich zu dem verbauten Lidar bestünde die Möglichkeit, Informationen der verbauten RealSense-Tiefenkamera zur Distanzmessung von Hindernissen zu verwenden. Dafür sind allerdings Techniken der erweiterte Mustererkennung von Nöten, deren Einsatz außerhalb des Umfangs der Projektgruppe liegt. Möglicherweise kann daher auf bereits fertige Bibliotheken zurückgegriffen werden, die ein einfaches Auswerten der Bildinformationen ermöglichen.

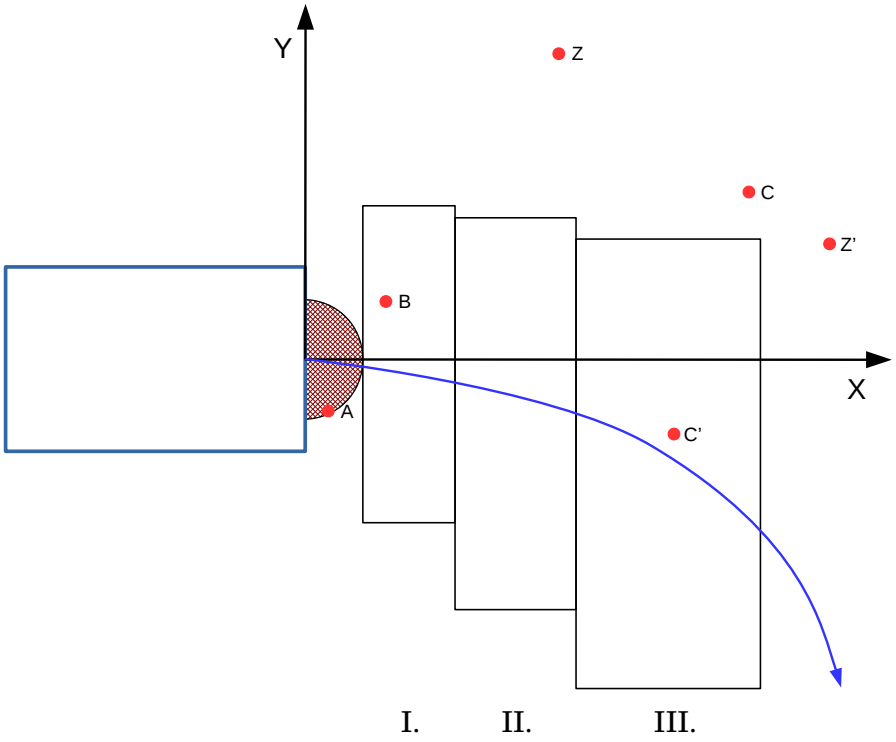


Abbildung 15.1: Schematische Darstellung von hypothetischen Distanz-Buckets während einer Kurvenfahrt entlang der blauen Kurve.

Computergestützte Steuerung

Die Steuerungsprobleme sowie der begrenze Fahrbereich führten zu der Erkenntnis, dass für Realfahrten eine computergestützte Steuerung nötig ist. Eine solche Steuerung muss die Fahrzeuge entlang eines festgelegten Pfad führen. Für eine ACC-Testfahrt ist die gleichzeitige Steuerung von zwei Modellfahrzeugen nötig. Ein Hauptbestandteil einer solchen Steuerung ist die Bahnplanungskomponente, welche aus einer gegebenen Punktmenge einen geschlossenen abzufahrenden Pfad errechnet. Weiterhin muss die Fahrzeugaktorik so angesteuert werden, dass der nächste Wegpunkt abhängig von der aktuellen Position erreicht wird und das Fahrzeug möglichst direkt dem Pfad folgt. Bei einer so autonomisierten Fahrt müssen zusätzliche Sicherheitsvorkehrungen getroffen werden. Dies wurde durch das Beschreiben eines räumlich begrenzten Sicherheitsbereichs, der den Pfad enthält, umgesetzt. Sollte das Fahrzeug diesen verlassen, wird sofort eine Vollbremsung eingeleitet.

Anhang A

Verfügbare Fahrzeugwerte in der Monitoring-Sprache

Zahlenwerte

- DistanceFront
- DistanceRear
- TimeDistanceFront
- LeadingCarRelativeSpeed
- CurrentSpeed
- PhysicalAcceleration
- Steering
- dTime
- InputThrottle
- InputBrake
- Throttle
- Brake
- SetSpeed
- SetDistance

Booleanwerte

- SystemOnButton

- SystemActiveButton
- HasLeadingCar
- HasEngineError
- HasSteeringError
- HasGearboxError
- DecrementSetDistanceButton
- IncrementSetDistanceButton
- DecrementSetSpeedButton
- IncrementSetSpeedButton
- ObstacleDetectedWarning
- ErrorWarning
- SystemOn
- SystemActive
- HeadlightsOn

Anhang B

ADTF-Komponenten

AADC-Arduino-Communication

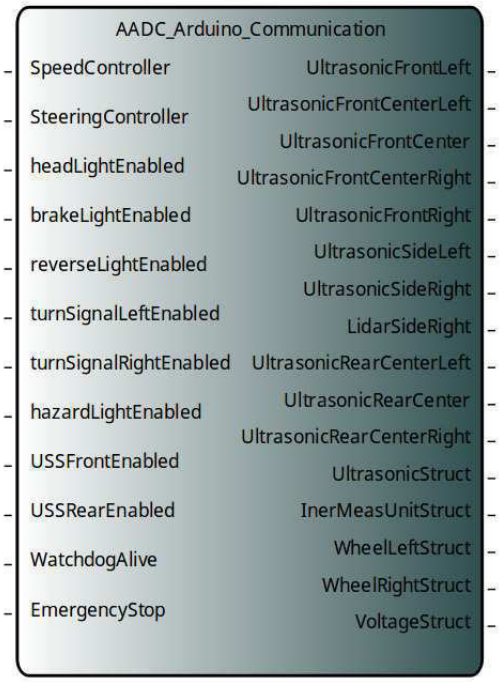


Abbildung B.1: AADC-Arduino-Communication-Filter

Der AADC-Arduino-Communication-Filter bewerkstelligt die Kommunikation zwischen der ADTF-Umgebung und den am Fahrzeug angebrachten Mikrocontrollern (Arduinos), welche im Wesentlichen den Antriebsstrang und die Lenkung des Fahrzeugs kontrollieren. Über die Eingänge *SpeedController* und *SteeringController* lassen sich Motor und Lenkung ansprechen. Zusätzlich stellt der Filter Werte der verbauten Ultraschallsensoren, des Beschleunigungssensors, des Magnetometers und der Radsensoren zur Verfügung.

AADC-Converter-Wheels

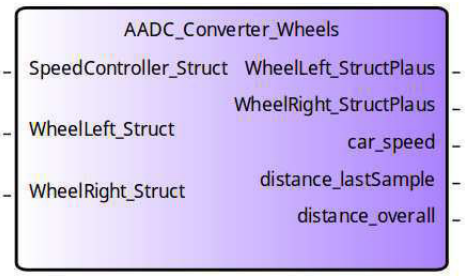


Abbildung B.2: AADC-Converter-Wheels-Filter

Das AADC-Converter-Wheels-Plugin erwartet für die hier verwendete Funktionalität als Eingänge die Radumdrehungszahlen *WheelLeft_ Struct* und *WheelRight_ Struct*. Daraus berechnet es unter anderem die aktuelle Fahrzeuggeschwindigkeit in m/s. Das Plugin muss dazu so konfiguriert werden, dass der korrekte Radumfang der Fahrzeugräder verwendet wird.

Ethernet-Device-TCP-Filter

Abbildung B.3: Ethernet-Device-TCP

Der Ethernet-Device-TCP-Filter bietet die Möglichkeit innerhalb von ADTF auf Netzwerksressourcen zuzugreifen. Über den Inputpin kann auf den Netzwerksocket geschrieben werden und über den Outputpin werden auf dem Socket empfangen Daten publiziert. Grundlage bildet das TCP [1].

Über die in Abbildung B.4 dargestellten Filterproperties müssen dazu der Hostname bzw. die IP-Adresse sowie der zu verwendende Port spezifiziert werden. Weiter kann eine konstante Paketgröße erzwungen werden.

[1] http://www.ietf.org/rfc/rfc0675.txt

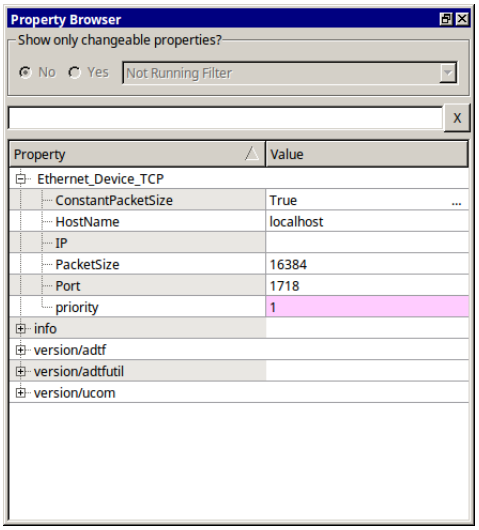


Abbildung B.4: Ethernet-Device-TCP-Properties

Abbildungsverzeichnis

Literaturverzeichnis

[1] *BMW 7er: Schöne neue Bedienwelt - Bilder - autobild.de*. http://www.autobild.de/bilder/bmw-7er-schoene-neue-bedienwelt-5908914.html#bild9. (Accessed on 2018-03-06).

[2] *Cinco Website*. https://cinco.scce.info/. (Accessed on 2018-03-05).

[3] *Eclipse Modeling Project | The Eclipse Foundation*. https://www.eclipse.org/modeling/emf/. (Accessed on 08/28/2018).

[4] Foundation, The Eclipse: *Classes and Members*. https://www.eclipse.org/xtend/documentation/202_xtend_classes_members.html.

[5] Foundation, The Eclipse: *Expressions*. https://www.eclipse.org/xtend/documentation/203_xtend_expressions.html.

[6] Foundation, The Eclipse: *Hello World*. https://www.eclipse.org/xtend/documentation/101_gettingstarted.html.

[7] Foundation, The Eclipse: *Introduction*. https://www.eclipse.org/xtend/documentation/.

[8] Foundation, The Eclipse: *Java Interoperability*. https://www.eclipse.org/xtend/documentation/201_types.html.

[9] Fowler, Martin: *Domain Specific Languages*. Addison-Wesley Professional, 1st Auflage, 2010.

[10] GmbH, Elektrobit: *EB Assist ADTF - Elektrobit*. https://www.elektrobit.com/products/eb-assist/adtf/. (Accessed on 2018-03-05).

[11] GmbH, Vires: *VTD - VIRES Virtual Test Drive*. https://vires.com/vtd-vires-virtual-test-drive/. (Accessed on 2018-03-05).

[12] International Business Machines Corporation (IBM), Eurotech: *MQTT V3.1 Protocol Specification*. http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html.

[13] International Organization for Standardization: *Intelligent Transport systems - Adaptive Cruise Control systems - Performance requirements and test procedures.* Standard, International Organization for Standardization, CH, april 2010.

[14] M. Dupuis, H.Grezlikowski: *OpenDRIVE, an open standard for the description of roads for driving simulations.* Proceedings of the Driving Simulation Conference, 2006.

[15] Mann, H., H. Schiffelgen und R. Froriep: *Einführung in die Regelungstechnik: analoge und digitale Regelung, Fuzzy-Regler, Regler-Realisierung, Software.* Hanser, 2009.

[16] Neumann-Cosel, Kilian von: *Virtual Test Drive: Simulation umfeldbasierter Fahrzeugfunktionen.* Doktorarbeit, Technische Universität München, 2013.

[17] Presagis: *OpenFLIGHT.* https://www.presagis.com/en/glossary/detail/openflight/. (Accessed on 2018-02-26).

[18] project, Eclipse Paho: *Eclipse Paho.* http://www.eclipse.org/paho/.

[19] Reif, K.: *Fahrstabilisierungssysteme und Fahrerassistenzsysteme.* Bosch Fachinformation Automobil. Vieweg+Teubner Verlag, 2010.

[20] RetroPie. https://github.com/RetroPie/RetroPie-Setup/wiki/Xbox-360-Controller. (Accessed on 2018-01-18).

[21] rubyrails.org: *Ruby on Rails unter Ruby.* [Online; accessed 10-January-2018].

[22] *Reglervergleich.* http://rn-wissen.de/wiki/index.php/Regelungstechnik. Eingesehen am 04.10.2017.

[23] VIRES: *Road Designer (ROD).* https://vires.com/docs/Rod201306.pdf. Accessed on 2018-02-26.

[24] volvocars.com: *Adaptive Geschwindigkeitsregelanlage (Tempomat) mit Geschwindigkeitsbegrenzer.* [Online; accessed 4-March-2018].

[25] Wirkner, Dominic: *Merge-Strategien für Graphmodelle am Beispiel von jABC und Git.* Diploma thesis, TU Dortmund, Februar 2015.

[26] *Xtext - Language Engineering Made Easy!* https://www.eclipse.org/Xtext/. (Accessed on 08/28/2018).

[27] Zech S., Weichhart A., Kuck A.: *Audi Autonomous Driving Cup 2017 Hardware Description.* https://www.elektrobit.com/products/eb-assist/adtf/, 2017.