Bachelorstudiengang Informatik

Bachelorarbeit

TSC2OpenX - Realisierung einer Werkzeugkette zur Simulation abstrakter Verkehrsszenarien

vorgelegt von

Vincent Kalwa

Gutachtende:
Prof. Dr. Martin Fränzle
Dr. Christian Neurohr

Oldenburg, den 20. September 2021

Zusammenfassung

Bei der Entwicklung autonomer Fahrzeuge ist das Testen der Fahrzeuge unabkömmlich zur Garantie ihrer Sicherheit. Das Testen auf öffentlichen Straßen oder auf dem Testgelände ist jedoch nicht nur zeit- und kostenintensiv, sondern kann bei gefährlichen Verkehrsszenarien leicht zu Sach- oder Personenschäden führen. Unter anderem aus diesen Gründen tendieren Fahrzeughersteller zur digitalen Simulation von Verkehrsszenarien.

Zur eleganten Beschreibung dieser Szenarien kann man Traffic Sequence Charts (TSCs), eine visuelle Beschreibungssprache mit formaler Semantik, benutzen. Die Spezifikation eines Verkehrsszenarios als „abstraktes Szenario“ mittels TSCs kann jedoch nicht direkt von Simulatoren eingelesen werden. In dieser Arbeit wird eine Werkzeugkette vorgestellt, welche TSCs in einer Sprache abbildet, die von Simulatoren verstanden wird und damit TSCs simulierbar macht.

Abstract

For the development of autonomous vehicles, extensive testing is crucial to ensure their safety. Testing on public roads or proving grounds is not only time-consuming and expensive but can easily lead to damaged property or personal injury in dangerous traffic scenarios. To accommodate for these problems, vehicle manufacturers have turned to digitally simulating traffic scenarios.

Traffic Sequence Charts (TSC), a visual description language with formal semantics, can be used to elegantly describe these scenarios. However, the specification of a traffic scenario as an „abstract scenario“ using TSC cannot be read directly by simulators. In this thesis, a tool chain is presented which maps TSCs into a language that is understood by simulators and thus enables TSCs to be properly simulated.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Zielsetzung . . . 2

2 Grundlagen **3**

2.1 Verkehrsszenarien . . . 3

2.2 Beschreibung von Verkehrsszenarien mittels Traffic Sequence Charts . . . 4

2.2.1 Weltmodell . . . 4

2.2.2 Symbolbibliothek . . . 6

2.2.3 Aufbau eines TSCs . . . 7

2.3 Beschreibung konkreter Verkehrsszenarien mittels OpenX-Standards . . . 8

2.3.1 OpenDRIVE . . . 8

2.3.2 OpenSCENARIO . . . 9

2.4 Simulieren konkreter Verkehrsszenarien . . . 10

3 TSC2OpenX - Idee der Werkzeugkette **11**

3.1 TSC Editor . . . 12

3.2 TSC2OpenX . . . 13

3.3 esmini . . . 13

4 TSC2OpenX - Realisierung der Werkzeugkette **13**

4.1 Anforderungen . . . 14

4.2 Modellierung . . . 14

4.3 Lösen des TSCs . . . 16

4.4 Erweiterung des Weltmodells und der Symbolbibliothek . . . 17

4.5 Einbindung der Standards OpenDRIVE und OpenSCENARIO . . . 18

4.6 Konfigurationsdateien . . . 19

4.7 Zuordnung von TSC auf OpenX . . . 19

4.8 Limitierungen . . . 23

5 Variationsmethoden zur Erzeugung verschiedener konkreter Szenarien **24**

5.1 Variation des Solver Seeds . . . 24

5.2 Variation des Weltmodells . . . 26

5.3 Variation existierender Constraints . . . 27

5.4 Variation via Blocking Clauses . . . 29

6 Evaluation der Werkzeugkette **31**
6.1 Definition eines Streuungsmaßes zur Evaluation der Variationsmethoden . . 32
6.2 Überholvorgang auf einer zweispurigen Straße 35
6.3 Überholvorgang auf einer mehrspurigen Straße mit verschiedenen Fahrzeugen 36
6.4 Straße mit einer zulässigen Höchstgeschwindigkeit 38
6.5 Inkonsistentes abstraktes Szenario 40
6.6 Auswertung 41

7 Zusammenfassung und Ausblick **47**

A Appendix **55**

B Eidesstattliche Erklärung **57**

1 Einleitung

Spurhalteassistenten, Einparkautomatik oder Totwinkel-Assistenten: Fahrerassistenzsysteme sind schon heute in vielen Neuwagen verbaut und erfreuen sich steigender Beliebtheit [13]. Diese Systeme sind Vorreiter für zukünftige Entwicklungen in der Automobilbranche [30]. Die Fahrzeuge sollen den Fahrer entlasten und immer selbstständiger agieren können: Entweder als teilautomatisiertes Fahrzeug, in dem der Fahrer in bestimmten Einsatzgebieten die Fahrzeugsteuerung abgeben kann, oder als vollautomatisiertes Fahrzeug, in dem Passagiere keine Kontrolle mehr über das Fahrzeug übernehmen können und stattdessen nur noch die gewünschten Zieldaten angeben. In beiden Fällen trifft das automatisierte Fahrsystem des Fahrzeugs sicherheitskritische Entscheidungen auf die der Mensch keinen Einfluss hat.

Die Mehrzahl der heutigen Verkehrsunfälle sind auf menschliches Versagen und Fehlverhalten des Fahrzeugführers zurückzuführen [5]. Durch Ausschluss des Menschen bei der Fahrzeugsteuerung wird automatisierten Fahrzeugen eine sicherheitsunterstützende Funktion im Straßenverkehr zugesprochen [29]. Um diese Sicherheit garantieren zu können, müssen Fahrzeughersteller ihre Fahrzeuge testen. In Realfahrten auf öffentlichen Straßen bzw. auf dem Testgelände ist dies jedoch nur begrenzt möglich, da zum Teil risikoreiche Szenarien, wie z.B. das Verhalten bei Unfällen, getestet werden müssen. Weiterhin ist das konstante Replizieren der Tests sehr aufwendig und kostenintensiv, weil Umgebungsverhältnisse (z.B. Nässe, Vereisung, u.Ä.) nicht jederzeit nachgestellt werden können. Eine konstante Wiederholung der Tests ist notwendig, damit man in späteren Entwicklungsstadien der Fahrzeuge sicher gehen kann, dass sich das Fahrzeug, z.B. durch Einführung neuer Funktionalitäten, wie gewünscht verhält (Regressionstesten [22]). Außerdem treten sicherheitskritische Ereignisse in der realen Welt zu selten auf, um statistisch belastbare Aussagen über die Sicherheit automatisierter Fahrzeuge - basierend auf gefahrenen Kilometern - zu treffen [19]. Aus diesen Gründen setzen Fahrzeughersteller vermehrt auf eine digitale Simulation der Verkehrsszenarien [21].

Die digitale Simulation wirft jedoch ein weiteres Problem beim Testen der Sicherheit autonomer Fahrzeuge auf: Man kann nicht alle realen Szenarien digital testen, da die Realität nicht vollständig digital abbildbar ist (Vollständigkeitsdilemma [4]). Entwickler automatisierter Fahrfunktionen müssen sich auf eine Menge verschiedener relevanter Eigenschaften

der Umwelt beschränken, welche das automatisierte Fahrzeug auswertet, um Entscheidungen über die zu setzenden Steuerbefehle zu treffen. Die Formalisierung dieser Eigenschaften kann jedoch zu großen Parameterräumen führen [1].

Ein kurzes Beispiel: Ein einem automatisierten Fahrzeug voraus fahrendes Fahrzeug kann sich in einem Abstand von 10-50m zum automatisierten Fahrzeug befinden. Theoretisch müsste für jeden reellen Abstand getestet werden, ob das automatisierte Fahrzeug seine Mobilitäts-, Performanz- und Sicherheitsziele erfüllen kann. Fügt man nun einen zweiten Parameter ein, z.B., dass die Fahrzeuge sich mit einer Geschwindigkeit zwischen 30-50km/h bewegen können, fällt schnell auf, dass sich die Anzahl der zu testenden Szenarien exponentiell mit jeder Eigenschaft erhöht. Es kommt zu einer Explosion des Parameterraums (State Explosion Problem [7]).

Die visuelle Szenarienbeschreibungssprache mit formaler Semantik Traffic Sequence Chart (TSC) [8] ermöglicht eine einfache Darstellung von Verkehrsszenarien und wirkt dem State Explosion Problem durch Abstraktion des Szenarios entgegen.

1.1 Zielsetzung

Mit aktuell verfügbarer Simulationssoftware kann man die in TSCs dargestellten abstrakten Szenarien nicht simulieren. Stattdessen muss das dargestellte Szenario in ein passendes Szenario zur Simulation umgewandelt werden. Ziel dieser Arbeit ist die Realisierung und Erprobung einer Werkzeugkette zur automatischen Konvertierung von abstrakten Szenarien in Form von TSCs zu konkreten Szenarien als OpenDRIVE und OpenSCENARIO, sowie deren anschließenden Simulation. Aufgrund der Komplexität des Straßenverkehrs wird sich dafür in dieser Arbeit auf eine Implementation im Szenarioumfeld „Autobahn“ beschränkt. Dazu werden als erstes einige Grundlagen erläutert. Danach wird die Entwicklung und der Aufbau eines Programms beschrieben, welches diese Konvertierung durchführen kann. Nachfolgend wird untersucht, wie man unterschiedliche Simulationsläufe aus einem TSC produzieren kann, die TSCs also variieren kann. Um diese Variationsmethoden zu bewerten, wird ein Variationsmaß definiert, welches Aufschluss gibt, in wie weit sich die unterschiedlichen Simulationen voneinander unterscheiden. Zum Schluss wird die Werkzeugkette an verschiedenen Szenarien erprobt und die erzeugten Ergebnisse ausgewertet.

2 Grundlagen

Im Folgenden werden zuerst einige Grundlagen vermittelt, welche im späteren Verlauf der Arbeit weiter benötigt werden. Dafür wird zuerst der Begriff „Szenario" definiert und erläutert. Es wird aufgezeigt, wie man Szenarien erstellen kann und welche Abstraktionsgrade diese besitzen können. Danach wird erklärt, wie man Verkehrsszenarien mittels „Traffic Sequence Charts" und „OpenX" Dateien definieren kann. Weiterhin wird erläutert, wie man diese Verkehrsszenarien erfolgreich simulieren kann.

2.1 Verkehrsszenarien

Nach [27] wird ein Szenario als eine zeitliche Entwicklung einer Reihe von Szenen definiert. Eine Szene wird als eine Momentaufnahme der Umgebung, inklusive dynamischer und statischer Objekte, interpretiert. Jedes Szenario startet mit einer initialen Szene und wird durch Aktionen und Ereignisse dieser Objekte zeitlich entwickelt. [27]

Beim Erstellen eines Szenarios kann als erstes ein *funktionales Szenario* skizziert werden. Dabei wird informell und leicht verständlich das Szenario via Text oder Bildern beschrieben. Aus funktionalen Szenarien kann man *abstrakte Szenarien* ableiten. Dafür wird das informelle Szenario ausformuliert und maschinenlesbar notiert. Dies ist möglich, indem man die auftretenden Relationen und Einschränkungen (Constraints) des Szenarios identifiziert und an eine Ontologie bindet. Eine Ontologie ist im Wesentlichen ein digitales Werkzeug zum Speichern von Wissen über Eigenschaften und Relationen zwischen Entitäten [6, S. 16]. Entitäten sind die im Verkehr vorkommenden Akteure. Wenn man die im abstrakten Szenario definierten Relationen und Constraints parametrisiert und auf Wertebereiche mit entsprechendem Verteilungen der Parameter beschränkt, erhält man ein *logisches Szenario*. Wählt man aus diesen Wertebereichen des logischen Szenarios jeweils einen validen Wert aus und instanziiert so diese Relationen und Constraints konkret, erhält man ein *konkretes Szenario*. [6, S. 19f] Aus einem abstrakten Szenario lassen sich also im Prinzip unendlich viele verschiedene logische Szenarien ableiten, dementsprechend auch unendlich viele konkrete Szenarien.

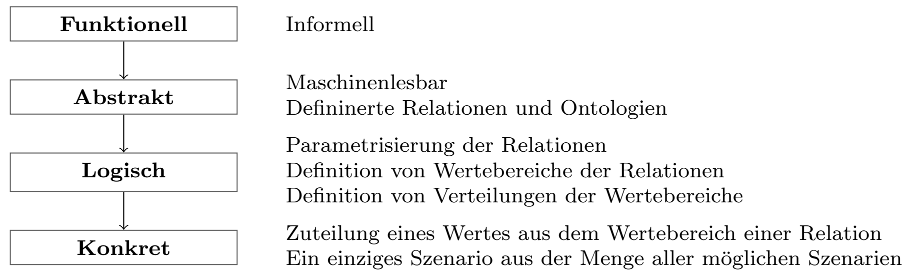


Abbildung 1: Verschiedene Abstraktionslevel von Szenariobeschreibungen.

2.2 Beschreibung von Verkehrsszenarien mittels Traffic Sequence Charts

Um Verkehrsszenarien einfach visuell und formal darstellen zu können, haben Damm et al. [8] die Entwicklung von Traffic Sequence Charts (TSCs) vorangetrieben. Es handelt sich um eine visuelle Beschreibungsmethode mit formaler Semantik, um abstrakte Verkehrsszenarien zu beschreiben. Eine TSC Spezifikation besteht aus TSC, Weltmodell und Symbolbibliothek. Ein TSC wird über das Weltmodell interpretiert. Dafür stellt die Symbolbibliothek eine Beziehung zwischen der grafischen Darstellung der Objekte (Symbole) im TSC und den Objekten im Weltmodell her.

2.2.1 Weltmodell

Im Weltmodell (world model) werden diverse Objekte der realen Welt (z.B. Fahrzeuge, Fahrräder, u.Ä.) zusammen mit ihren Eigenschaften (z.B. Länge, Breite, maximale Geschwindigkeit, u.Ä.) modelliert. Diese Objekte können häufig durch hybride Automaten beschrieben werden. In der Benutzung des TSC Editors (siehe Unterabschnitt 3.1) wird ein Dateiformat „Weltmodell“ eingeführt, welches das Modellieren der Objekte im XML Format ermöglicht.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wm:importWM xmlns:wm="http://www.offis.de/tsc/2019-09/wm"
       id="_H19aQKUOEeu6zP_Cqt1lOg">
  <wm:objectType id="_IH6s8KUOEeu6zP_Cqt1lOg" name="Car">
    <wm:property id="_Q_2-AKUOEeu6zP_Cqt1lOg" name="x" type="real"
               dynamics="continuous" unit="m"/>
    <wm:property id="_S9UJMKUOEeu6zP_Cqt1lOg" name="y" type="real"
               dynamics="continuous" unit="m"/>
  </wm:objectType>
  <wm:objectType id="_UtVVUKUOEeu6zP_Cqt1lOg" name="Road">
    <wm:property id="_VltHwKUOEeu6zP_Cqt1lOg" name="x" type="real"
               unit="m"/>
    <wm:property id="_WpuwkKUOEeu6zP_Cqt1lOg" name="y" type="real"
               unit="m"/>
  </wm:objectType>
</wm:importWM>
```

Abbildung 2: Ein einfaches Weltmodell: `simple.worldmodel`.

In diesem Beispiel enthält das Worldmodel eine Klasse Auto (Car) und eine Klasse Straße (Road). Beide Klassen besitzen die Eigenschaften einer x- und y-Koordinate. Die Eigenschaften können Werte einer realen Zahl annehmen (`type="real"`) und haben eine Einheit in Metern (`unit="m"`). Zudem haben beiden Klassen eine Angabe ihres zeitlichen Verhaltens (*dynamic*). Das Merkmal der Dynamik gibt an, ob sich die Eigenschaft der Klasse bei zeitlicher Betrachtung ändert. `dynamics="continuous"` bedeutet, dass sich die Eigenschaft, im Gegensatz zu `dynamics="constant"`, im zeitlichen Verlauf ändern kann. Da `"constant"` der Standardwert der Dynamik ist, wird sie bei der Road Klasse nicht noch einmal zusätzlich angegeben.

Die Eigenschaften der Klassen können im Weltmodell via Constraints eingeschränkt werden. Möchte man z.B., dass das in Abbildung 2 definierte Auto maximal 180 km/h fahren kann, fügt man die Eigenschaft v (hier für Velocity, Schnelligkeit) hinzu und schränkt die Eigenschaft ein:

```xml
<wm:objectType id="_IH6s8KUOEeu6zP_Cqt1lOg" name="Car">
   <wm:property id="_Q_2-AKUOEeu6zP_Cqt1lOg" name="x" type="real"
             dynamics="continuous" unit="m" />
   <wm:property id="_S9UJMKUOEeu6zP_Cqt1lOg" name="y" type="real"
             dynamics="continuous" unit="m" />
   <wm:property id="_InJsMN_eEeubhvx43yuU3Q" name="v" type="real"
             dynamics="continuous" unit="km/h"/>
   <base:constraint>v &lt;= 180[km/h]</base:constraint>
</wm:objectType>
```

Abbildung 3: Beschränkte Eigenschaften des Fahrzeug Objekts.

Um das spätere Einlesen des Weltmodells durch Programme zu vereinfachen, wird hier die HTML Notation verwendet: z.B. wird das Sonderzeichen < zu `lt;`.

2.2.2 Symbolbibliothek

Damit man die im Weltmodell erstellten Objekte im TSC benutzen kann, muss eine Grafik dem Objekt zugeordnet werden. Diese Zuordnung geschieht in der Symbolbibliothek (im Original symbol dictionary). Für die Verwendung des TSC Editors wird ein Dateiformat „sdict“ eingeführt, welches die Zuordnung im XML Format ermöglicht.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wm:dictionary xmlns:wm="http://www.offis.de/tsc/2019-09/wm"
       id="_UUkoUKUUEeu6zP_Cqt1lOg">
  <wm:importWM>simple.worldmodel#_H19aQKUOEeu6zP_Cqt1lOg</wm:importWM>
  <wm:symbol id="_W7tCoKUUEeu6zP_Cqt1lOg" name="Car"
       type="simple.worldmodel#_IH6s8KUOEeu6zP_Cqt1lOg"
       visualization="car.png"/>
  <wm:symbol id="_hn_mMKUUEeu6zP_Cqt1lOg" name="Road"
       type="simple.worldmodel#_UtVVUKUOEeu6zP_Cqt1lOg"
       visualization="road.png"/>
</wm:dictionary>
```

(a) car.png

(b) road.png

Abbildung 4: Eine einfache Symbolbibliothek: `simple.sdict`.

Hier wird ein jeweiliges Objekt im Weltmodell anhand der Identifikation einer Grafik zugeordnet. Die Identifikation eines Objektes geschieht in der Form
<Weltmodell-Datei>#<Weltmodell-Objekt ID>.

2.2.3 Aufbau eines TSCs

Beim Erstellen eines TSCs wird zuerst die Symbolbibliothek importiert. Danach können im *BulletinBoard* benötigte Symbole registriert werden.

Die grafische Darstellung eines TSCs findet in *Spatial Views* statt. Diese sind als kurze Zeitausschnitte des Verkehrsszenarios zu verstehen. Das Definieren von Symbolen im BulletinBoard ermöglicht eine „globale“ Fixierung der Symbole, sodass diese über alle Spatial Views gleich identifizert werden. Die Spatial Views können mittels *Invariantenknoten* verknüpft werden und ermöglichen so eine zeitlich nahtlose Darstellung des Szenarios. Zeitsprünge innerhalb den Invariantenknoten sind allerdings auch erlaubt. Im Kontext dieser Arbeit werden nur lineare Zeitabläufe betrachtet.

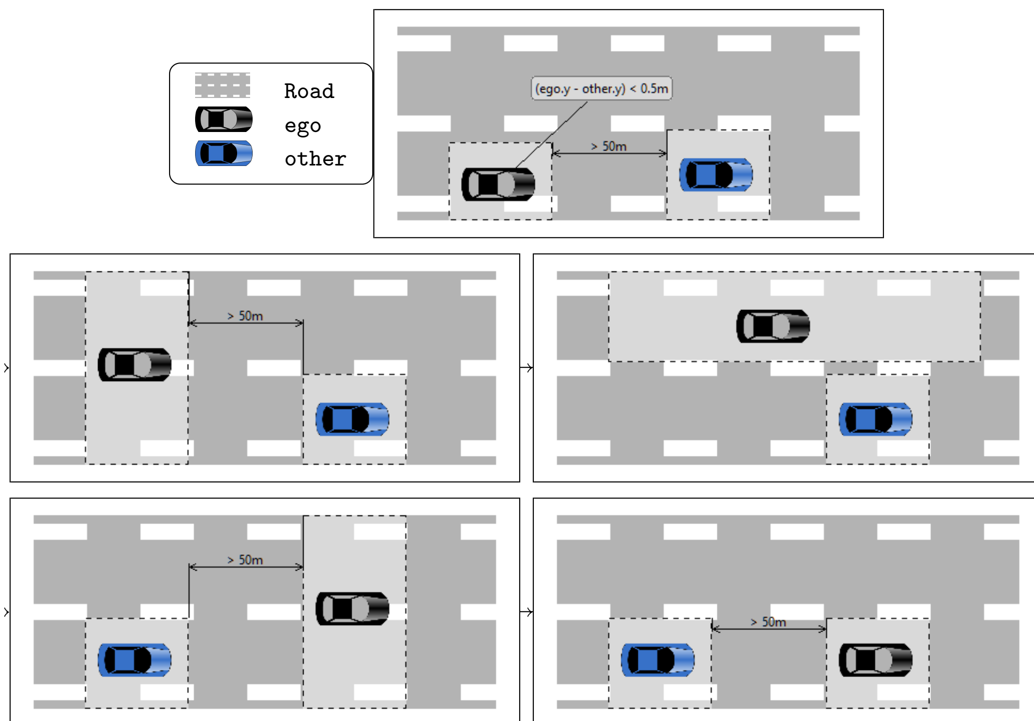


Abbildung 5: TSC: Ein Überholvorgang.

Ein TSC ließt sich von links nach rechts bzw. von oben nach unten. Das erste Fenster gibt das BulletinBoard an, also welche Objekte welcher grafischen Repräsentation und Namen innerhalb des TSC zugeordnet werden. Danach folgen die Invariantenknoten, die die Reihenfolge der Spatial Views angeben. In den Spatial Views wird das eigentliche Szenario beschrieben. Das schwarze Fahrzeug (im Folgenden *ego*) befindet sich hinter dem blauen Fahrzeug (im Folgenden *other*). Die gestrichelten Kästchen um die Symbole geben an, dass diese Fahrzeuge sich überall innerhalb des Kästchens befinden dürfen. Das wird im nächsten Invariantknoten besonders wichtig. Da sich die Box auf beiden Spuren befindet, darf sich *ego* auf beiden Spuren befinden. Es wird also ein Spurwechsel angedeutet, da sich *ego* im nachfolgenden Spatial View auf der linken Spur befindet.

2.3 Beschreibung konkreter Verkehrsszenarien mittels OpenX-Standards

OpenSCENARIO (OSC) [10] und OpenDRIVE (ODR) [9] sind standardisierte, von ASAM [11] gepflegte Dateiformate zum Erstellen von Verkehrszenarien und Straßennetzwerken im XML Format. Sie werden von einer wachsenden Zahl von Simulationsprogrammen unterstützt. In dieser Arbeit werden OpenSCENARIO und OpenDRIVE unter dem Term OpenX zusammengefasst.

2.3.1 OpenDRIVE

Mit OpenDRIVE kann man statische Straßennetzwerke definieren. In einer validen ODR-Datei wird mindestens eine Straße definiert. Diese Straße muss einen Mittelstreifen und mindestens eine Fahrbahn enthalten. Weiterhin kann man das Terrain der Umgebung formen, Straßenmarkierungen hinzufügen, Straßenschilder aufstellen oder Ampelanlagen definieren.

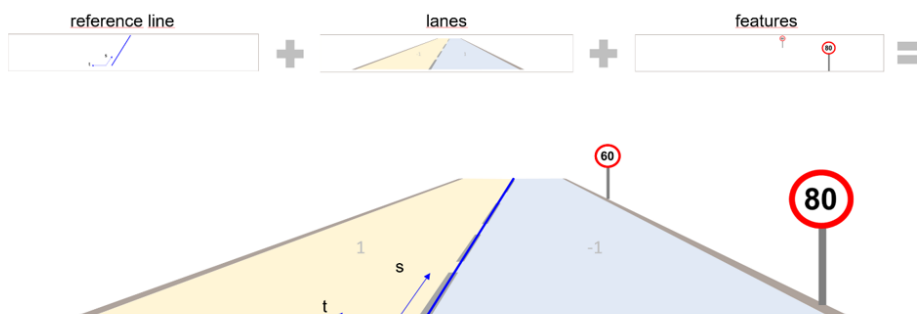


Abbildung 6: „The individual parts of a road“ [28].

In Abbildung 6 werden einige Funktionen von OpenDRIVE visualisiert. Wie man sehen kann, identifiziert man die einzelnen Fahrspuren relativ zum Mittelstreifen einer Straße. So haben Fahrspuren links des Mittelstreifens eine positive Identifikationsnummer, die rechts vom Mittelstreifen eine negative. Wenn es nicht anders eingestellt wird, werden die Spurrichtungen direkt vorgegeben und Rechtsverkehr angewendet.
Hat man ein Straßennetzwerk definiert, können weitere Merkmale hinzugefügt werden, wie z.B. Straßenschilder: Hier wurden Straßenschilder für die Fahrspur -1 definiert.

2.3.2 OpenSCENARIO

Ein vollständiges OpenSCENARIO enthält eine Fahrzeugbibliothek, die Angabe eines Straßennetzwerkes, eine Registrierung aller Entitäten im Szenario und die eigentliche Definition des Szenarios.

In einem *VehicleCatalog* werden Fahrzeuge definiert, welche in einem Szenario auftreten können. Dort wird jedem Fahrzeug eine Bezeichnung und ein 3D-Modell zugeordnet. Weiterhin können Eigenschaften des Fahrzeugs angegeben werden, wie maximale Geschwindigkeit, Beschleunigung und Verzögerung. Ein VehicleCatalog kann man als Äquivalent einer Symbolbibliothek in OpenX sehen.

Damit Fahrzeuge auf Straßen fahren können, wird ein Straßennetz via *RoadNetwork* eingebunden. Hier wird zwischen logischem und grafischem Straßennetz unterschieden. Zum einen kann man das logische Straßennetzwerk als OpenDRIVE-Datei einbinden, zum anderen kann man eine 3D-Welt im OpenSceneGraph-Format [24] einbinden.

Nun können Entitäten, die im Szenario vorkommen sollen, definiert werden. Den Entitäten wird ein Name und ein Fahrzeug aus dem VehicleCatalog zugeordnet. Man kann diese Definitionen mit dem BulletinBoard eines TSCs vergleichen. Diejenigen Entitäten, die vor dem Szenario definiert werden, sind während des Szenarios statisch zu behandeln.

Wurden alle gewünschten Entitäten definiert, kann mit dem Erstellen des Szenarios begonnen werden. Dies geschieht in einem *StoryBoard*. Im StoryBoard wird zuerst der Initial-Zustand eines Szenarios beschrieben. In diesem Fall werden alle Fahrzeuge an ihre Startkoordinaten der Straße gesetzt. Danach wird eine *Story* definiert, welche vorgibt, was während

des Szenarios geschehen soll. Hier soll jedes Fahrzeug einer definierten Trajektorie folgen, d.h. es wird eine *FollowTrajectoryAction* definiert. Um den Simulatoren vorzugeben, wie die Trajektorien während des Szenarios interpoliert werden sollen, kann die Trajektorie mit verschiedenen *Shapes* angegeben werden. Derzeit sind die Methoden *Polyline*, *Clothoid* und *Nurbs* definiert. Während die Polyline-Methode die Vertices der Trajektorie nur linear verbindet, benutzen die anderen Methoden jeweils Klothoide und Splines, um die Trajektorie glatter zu verlaufen zu lassen. Da die Trajektorie jedoch schon mit Splines berechnet wird (Siehe Unterabschnitt 4.3), wird trotzdem die Polyline-Methode verwendet. Damit stellt man sicher, dass sich die Fahrzeuge wirklich durch die Trajektorienknoten bewegen und nicht durch unterschiedliche Implementationen der Interpolationsmethoden durch den Simulator nebenher fahren.

2.4 Simulieren konkreter Verkehrsszenarien

Geeignete Simulatoren können OpenX Dateien einlesen, die Trajektorien der Objekte berechnen und den Verlauf der Trajektorien in Log-Dateien ausgeben oder zusätzlich visuell darstellen. Einige Simulatoren sind Open Source und kostenfrei erhältlich, wie z.B. die Simulatoren esmini [12] oder CARLA [26]. esmini ist ein simples Programm, welches sich auf die visuelle Darstellung der Szenarien konzentriert. CARLA ist ein fortgeschrittener, in der Unreal Engine programmierter Simulator. Neben dem Abspielen der Szenarien via dem „Scenariorunner“ unterstützt CARLA auch weitere Funktionen, wie z.B. die Simulation einzelner Fahrzeugsensoren. Da der CARLA ScenarioRunner aber in der Version 0.9.10 die FollowTrajectoryAction nicht unterstützt, wurde der Simulator nicht weiter getestet.

Andere gängige Simulatoren sind Virtual Test Drive (VTD) [18], welcher derzeit jedoch nicht frei zugänglich ist, und OpenPASS [16], welcher derzeit jedoch keine kostenfreie Visualisierung besitzt. Deshalb wurden diese Simulatoren nicht weiter untersucht.

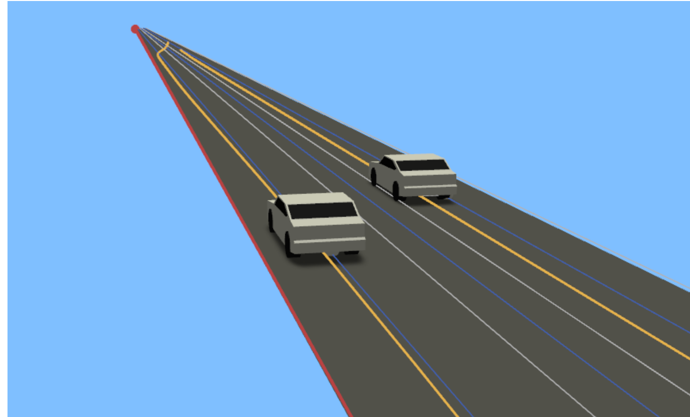


Abbildung 7: Eine Momentaufnahme aus der Simulation generierter OpenX Dateien durch den Simulator esmini.

3 TSC2OpenX - Idee der Werkzeugkette

Die komplette Werkzeugkette besteht aus insgesamt drei Programmen, welche das Erstellen des TSCs, die Konvertierung des TSCs und die Simulation der OpenX Dateien enthält.

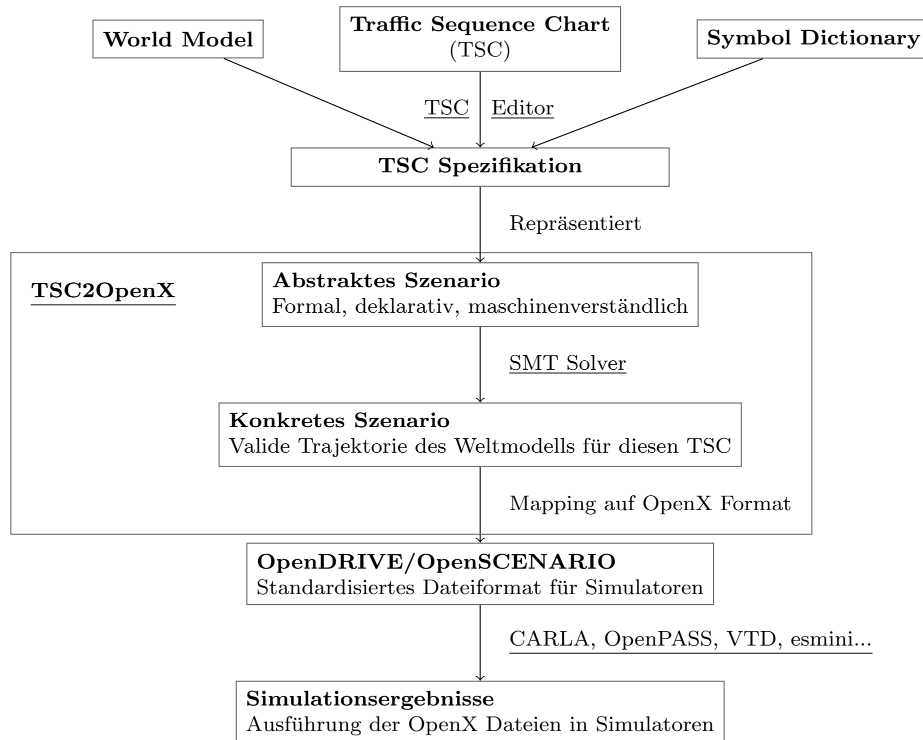


Abbildung 8: Die komplette TSC2OpenX Werkzeugkette.

Die einzelnen Programmkomponenten in Abbildung 8 wurden unterstrichen hervorgehoben und werden im Folgenden weiter erläutert.

3.1 TSC Editor

Der von Becker et al. [2] entwickelte TSC Editor liefert eine Sammlung von Erweiterungen für die integrierte Entwicklungsumgebung Eclipse [14] zum bequemen Erstellen und Bearbeiten von TSCs. Das Programm beinhaltet, neben einem grafischen Editor zum Erstellen der TSCs, die Möglichkeit zum Erstellen von Weltmodellen und Symbolbibliotheken im XML Format. Darüber hinaus kann das Programm erstellte Dateien bezüglich auf die Korrektheit eines TSCs validieren und fertig gestellte TSCs in verschiedenen Formaten grafisch exportieren. In dieser Arbeit wird der Build „2021-08-23“ benutzt.

3.2 TSC2OpenX

Das Programm TSC2OpenX wird die Konvertierung abstrakter Verkehrsszenarien zu konkreten Verkehrsszenarien durchführen. Die Umsetzung des Programms wird in Abschnitt 4 weiter erläutert. Das Programm hat einen Umfang von ungefähr 3000 Zeilen Programmcode.

3.3 esmini

Environment Simulator Minimalistic (esmini)[12] ist ein simpler quelloffener OpenSCENARIO Player. Seit der Version 1.5 unterstützt esmini die OpenSCENARIO Version 1.0, welche in dieser Arbeit benutzt wird. Während der Entwicklung von TSC2OpenX wurden verschiedene Versionen von esmini veröffentlicht. Generierte OpenX-Dateien laufen sicher auf der Version 2.8.1.

Um eine OpenSCENARIO Datei auszuführen, startet man den „EgoSimulator“ mit einigen Parametern:

- `--window [start.x] [start.y] [windows width] [window height]`: Die gewünschte Position und Größe des Simulationsfensters
- `--osc [filename]`: Angabe der OpenSCENARIO Datei
- `--info_text [on/off]`: Anzeige von Laufzeitinformationen der Fahrzeuge (z.B. Koordinaten, Geschwindigkeit usw.)
- `--road_features [on/off]`: Anzeige der Straßenbegrenzungen der OpenDRIVE Dateien

Eine beispielhafte Ausführung eines OpenSCENARIOs `"scenario.xosc"` stellt sich wie folgt dar: `"EgoSimulator.exe --window 50 50 1024 576 --osc scenario.xosc --info_text on --road_features on"`.

4 TSC2OpenX - Realisierung der Werkzeugkette

Das in dieser Arbeit realisierte Programm TSC2OpenX ist das Kernstück der Werkzeugkette und ermöglicht die Konvertierung von einem abstrakten zu einem konkreten Verkehrsszenario. Dafür liest das Programm das Ausgangsszenario in Form eines TSCs ein und generiert entsprechende OpenSCENARIO und OpenDRIVE Dateien.

4.1 Anforderungen

Vor der Entwicklung des Programms wurden einige Anforderungen formuliert.

Tabelle 1: Anforderungen an das Programm TSC2OpenX.

ID	Anforderung	Begründung
A1	Das Programm muss in Java programmiert werden.	Benötigte Bibliotheken sind bereits in Java verfügbar.
A2	Das Programm muss dem Benutzer die Möglichkeit bieten, TSCs einlesen zu können.	Endbenutzerfreundlichkeit erhöhen, TSCs sollen nicht hart kodiert werden.
A3	Das Programm muss valide OpenX Dateien erzeugen.	Dateien müssen den OpenX Standards entsprechen.
A4	Das Programm muss das Einbinden verschiedener Weltmodelle unterstützen.	Das Programm darf den Benutzer nicht auf ein vorgegebenes Weltmodell festlegen.
A5	Das Programm muss Trajektorien in grafischer Form visualisieren können.	Die generierten Trajektorien dürfen nicht nur in Simulatoren einsehbar sein.
A6	Das Programm muss verschiedene OpenX Dateien aus einer TSC Datei erzeugen.	Es soll nicht nur eine mögliche Lösung aus der Menge aller Lösungen gefunden werden.
A7	Das Programm muss erfüllende Simulationsdateien bezüglich des TSCs erzeugen.	Die Simulationsdateien dürfen die Semantik des TSCs nicht verletzen.

In den folgenden Kapiteln wird erläuert, wie die in Tabelle 1 definierten Anforderungen umgesetzt wurden.

4.2 Modellierung

Vor Beginn der Entwicklung wurde ein Aktivitätsdiagramm zur Modellierung des Programms erstellt.

Wie in Abbildung 9 beschrieben, muss dem Programm zuerst eine Datei übergeben werden, für das die Konvertierung durchgeführt werden soll (siehe Tabelle 1, Anforderung A2). Ist die Datei nicht im TSC Format, wird das Programm direkt terminiert. Nach erfolgreicher Übergabe der Datei im richtigen Format wird die Hauptkonfigurationsdatei

eingelesen, die wichtige Informationen über den weiteren Verlauf des Programms beinhaltet (siehe Unterabschnitt 4.6). Ist keine Hauptkonfigurationsdatei zu finden, wird eine grundlegende Konfigurationsdatei erstellt. Aus der Hauptkonfigurationsdatei wird entnommen, inwiefern das eingelesene TSC variiert werden soll. Wurde eine Variationsmethode angeben, werden diese TSC-Variationen generiert (siehe Abschnitt 5). Für jede erstellte Variante wird nun eine Solver-Instanz (siehe Unterabschnitt 4.3) erstellt und versucht, eine lösbare Belegung des TSC zu finden. Kann eine solche Belegung gefunden werden, erzeugt das Programm entsprechende OpenSCENARIO- und OpenDRIVE-Dateien. Nach Bearbeitung aller Varianten durch den Solver werden die Trajektorien visualisiert. Zum Schluss wird für alle Variationen ein Variationsmaß berechnet, um die Effektivität der Variationsmethoden zu vergleichen (siehe Unterabschnitt 6.1). Je höher das Variationsmaß, desto besser ist die Variationsmethode.

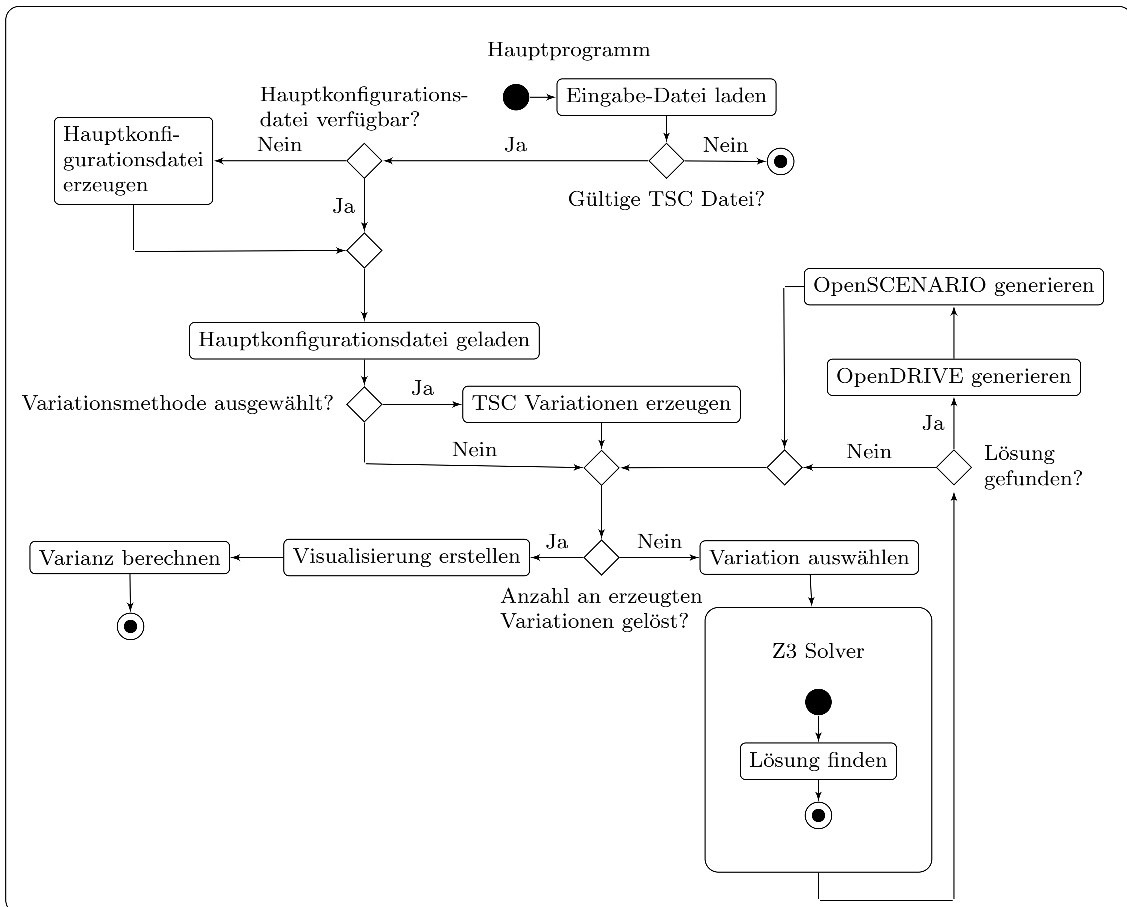


Abbildung 9: Aktivitätsdiagramm des Programms TSC2OpenX.

4.3 Lösen des TSCs

Ziel von TSC2OpenX ist das Ableiten eines abstrakten Szenarios in Form eines TSCs, zu konkreten Szenarien im OpenX Format. Dafür muss vom Programm mindestens eine passende Lösung für das abstrakte Szenario gefunden werden.

Nach Becker et al. [3] wird das TSC dafür in eine satisfiability modulo theories (SMT)-Formel im SMTLib-Format [23] umgewandelt und danach von einem SMT-Solver versucht zu lösen. Hier wird der Z3 Solver [31] verwendet. Heuristische Verfahren des Z3 Solvers ermöglichen ein schnelles Finden einer erfüllenden Lösung [17, p. 178]. Durch das Setzen eines - in diesem Fall numerischen - Wertes, dem sogenannten *Seed*, kann die Heuristik beeinflusst werden. Der Solver untersucht dann unterschiedliche

Teilzweige des Problems zuerst und findet möglicherweise unterschiedliche Lösungen.

In dieser Arbeit wird das Erzeugen der SMT-Formeln als Blackbox behandelt. Entsprechendes Java-Interface wurde von Becker nach [3] entwickelt. Wird eine erfüllende Belegung des TSCs gefunden, werden Trajektorien mit *Bounded Model Checking*(BMC) über endliche Zeitpunkte in einzelne Schritte abgerollt. Die maximalen BMC Schritte kann man manuell setzten. Wird nach dem letzten Schritt keine erfüllende Trajektorie gefunden, bricht der Solver die Berechnung ab.

Die Lösung des Solvers beinhaltet für jeden abgerollten Zeitschritt die Werte für die Eigenschaften der TSC Objekte. Jede Sekunde des Szenarios wird in einen Stützpunkt übersetzt. Je nachdem, ob die Dynamik (siehe Unterunterabschnitt 2.2.1) einer Eigenschaft im Weltmodell *constant* oder *continuous* gesetzt wurde, generiert der Solver entweder einen festen Wert (z.B. die Länge einer Straße) oder zeitvariable Werte (z.B. die X- und Y-Positionen eines Fahrzeugs).

4.4 Erweiterung des Weltmodells und der Symbolbibliothek

Damit der Solver weiß, wie bestimmte Variablen im Weltmodell spezifiert werden, wird im Weltmodell die Erweiterung *Property Semantic* eingeführt. Diese weißt den Variablen eine von den sechs verschiedenen Typen
`"X, Y, VELOCITY, ACCELERATION, BB_LENGTH, BB_WIDTH"` zu, wenn diese benutzt werden sollen. Nicht jeder Eigenschaft eines Objektes im Weltmodell muss eine Property Semantic zugeordnet werden. Die Property Semantics unterstützen das Erstellen von Weltmodellen für Autobahn-Szenarien. Andere Eigenschaften, wie z.B. Wenderadien definierter Objekte, werden deshalb nicht betrachtet.

Damit der Solver die Relationen zwischen den Objekten identifizieren kann, ist es nötig, dass die Ankerpunkte der Objekte bekannt sind. Dafür muss das Weltmodell um einige Eigenschaften erweitert werden: zum einen werden Eigenschaften für die Länge und Breite (im folgenden Beispiel über die Variablen *width* und *length*) den Objekten hinzugefügt. Durch eine *Derived Property* kann man Eigenschaften angeben, die sich aus anderen Eigenschaften ableiten lassen. Durch Angabe der Länge und Breite können abgeleitete Eigenschaften für die minimalen und maximalen X- und Y-Koordinaten (hier: min_x,

max_x, ...) des Objektes erstellt werden.

Sind diese abgeleiteten Eigenschaften bekannt, können im Symbol Dictionary zugehörige Ankerpunkte durch die Angabe von *Anchor Def*s definiert werden. In Abbildung 10 sind vier Ankerpunkte an den Ecken des Objektes definiert.

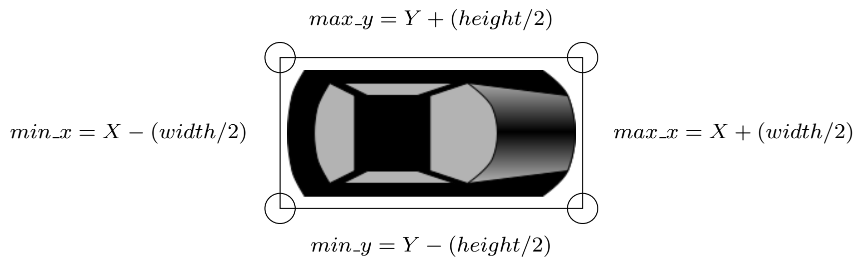


Abbildung 10: Visualisierung der Ankerpunkte eines Objekts.

Anhand der Ankerpunkte des Objekts kann der Solver nun erkennen, wie Objekte zueinander in Relation stehen, z.B. ob sich ein Auto-Objekt innerhalb eines Straßen-Objektes befindet.

Eine beispielhafte Implementation der Erweiterung des Weltmodels ist in Anhang A Abbildung 32 beschrieben, die implementierte Erweiterung der Symbolbibliothek in Anhang A Abbildung 33.

4.5 Einbindung der Standards OpenDRIVE und OpenSCENARIO

Als die Entwicklung des Programms TSC2OpenX begann, wurde gerade ASAM OpenSCENARIO V1.0.0 veröffentlicht. Dementsprechend wurde die neuere Version der älteren Version 0.9.1 vorgezogen. Die Version 1.0.0 benutzt gegenüber der Version 0.9.1 einige veränderte Beschreibungsmethoden, weshalb eine Rückwärtskompatibilität nicht möglich ist.
Weiterhin wurde OpenDRIVE V1.6.0 ausgewählt.

Von beiden Modellierungsstandards wurden die XML Schemata mittels dem Eclipse Modelling Framework (EMF) [15] in fertige Java Klassen übertragen. Dies ermöglicht eine leichte Generierung der XML Dateien. Weiterhin ist es möglich, mit den XML Schemata

die Korrektheit der generierten Dateien gegen das XML-Schema zu validieren (siehe Tabelle 1, Anforderung A3).

4.6 Konfigurationsdateien

Einige Einstellungen des Programms TSC2OpenX können in Konfigurationsdateien getroffen werden. Für das Konfigurationsformat wurde die leicht lesbare Sprache YAML [25] gewählt. Derzeit gibt es zwei wichtige Konfigurationsdateien:

In der Hauptkonfigurationsdatei `"configuration.yaml"` werden Einstellungen für das Programm und den Solver getroffen. So gibt man dem Solver über den Konfigurationspunkt `solver_max_steps` vor, wie viele Schritte der Solver maximal gehen soll, um eine Lösung zu finden. Weiterhin kann hier, über den Punkt `num_of_initiations`, angegeben werden, wie viele verschiedene konkrete Szenarien generiert werden sollen. Über `export_folder` wird definiert, in welchem Ordner generierte Dateien gespeichert werden sollen oder via `visualization` welche Art von Visualisierung von den generierten Trajektorien erstellt werden soll. Weitere Konfigurationsmöglichkeiten werden in Abschnitt 5 ergänzt.

In der Konfigurationsdatei `"mapping.yaml"` werden Einstellungen für die Zuordnung von TSC Objekttypen auf OSC oder ODR getroffen. Dies wird im folgenden Unterabschnitt 4.7 erläutert.

4.7 Zuordnung von TSC auf OpenX

Da für das Programm TSC2OpenX eine hohe Modularität angestrebt wird und das Programm sich nicht auf ein einziges Weltmodell beschränken soll (Siehe Tabelle 1, Anforderung A4), muss dem Programm eine Zuordnung bekannt sein, welche Objekttypen im TSC in OpenDRIVE oder OpenSCENARIO umzuwandeln sind. Diese Zuordnung wird in der Konfigurationsdatei `"mapping.yaml"` getroffen. Im BulletinBoard eines TSCs werden alle vorkommenden Entitäten des TSCs einem Objekt aus der Symbolbibliothek zugeordnet. Alle Objekte im BulletinBoard müssen im Programm TSC2OpenX den entsprechenden eindeutigen OpenX Objekten zugeordnet werden. Ansonsten sind die Entitäten nicht übersetzbar.

```
!!de.offis.vvm.tsc2osc.configuration.MappingConfiguration
mapping: {
    Car: {
        openscenario: vehicle,
        visualization: car_white,
        parameters: {
          x: Car.X,
          y: Car.Y
        }
    }
}
```

Abbildung 11: Eine Zuordnungskonfiguration für die Klasse PKW.

Beim Übertragen der Entität vom TSC Objekt zum OpenSCENARIO oder OpenDRIVE Objekt wird für jede OpenX Entität zuerst ein Blick in die Zuordnungskonfiguration geworfen. Das Objekt `Car` wird nach der Konfiguration ein OpenSCENARIO Objekt vom Typ Fahrzeug (`openscenario: vehicle`) zugeordnet. Fahrzeuge werden in OSC in einem so genannten `"VehicleCatalog"` definiert. In der Zuordnungskonfiguration wird nun definiert, welches Fahrzeug das Objekt im OSC aus dem VehicleCatalog repräsentieren soll, in diesem Beispiel ein weißes Auto (`visualization: car_white`). Ein weiteres Beispiel folgt mit der Konfiguration eines Busses in Abbildung 12.

```
!!de.offis.vvm.tsc2osc.configuration.MappingConfiguration
mapping: {
    Bus: {
    openscenario: vehicle,
    visualization: bus_blue,
    parameters: {
      x: Bus.X,
      y: Bus.Y
    }
  }
}
```

Abbildung 12: Eine Zuordnungskonfiguration für die Klasse Bus.

Der entsprechende TSC Objekttyp in der Symbolbibliothek ist nun `"Bus"`. Man kann

leicht erkennen, dass sich eigentlich nur die Visualisierung geändert hat, da im OSC nun kein weißes Auto, sondern ein blauer Bus dargestellt werden soll. Die Eigenschaften eines Objekttypen innerhalb der Simulation werden also allein durch das im VehicleCatalog definierte Fahrzeug bestimmt.

OpenDRIVE Objekte werden in der Zuordnungskonfiguration ähnlich behandelt. Da je nach TSC, insbesondere auch für andere Anwendungsdomänen (z.B. urbane Kreuzung oder Autobahn), eine andere Komplexität des Worldmodels gewünscht oder benötigt wird, gibt es derzeit zwei verschiedene Methoden, wie man ein Straßenobjekt nach ODR portieren kann.

In der ersten Methode definiert man das Objekt direkt als ODR Lane Objekt.

```
!!de.offis.vvm.tsc2osc.configuration.MappingConfiguration
mapping: {
  DoubleLaneHighway: {
      opendrive: lane,
      quantity: 2,
      type: driving,
      parameters: {
          width: DoubleLaneHighway.X
      }
  }
}
```

Abbildung 13: Eine Zuordnungskonfiguration für eine einfache Straße.

Die zweite Methode benötigt zwei TSC Weltmodell-Objekte. Zum einen eine Straße, zum anderen eine Fahrspur. Diese werden in der Zuordnungskonfiguration verschachtelt definiert:

```
!!de.offis.vvm.tsc2osc.configuration.MappingConfiguration
mapping:
  Road: {
    opendrive: road,
    parameters: {
      length: Road.bb_length,
      width: Road.bb_width
    },
    Lane: {
     opendrive: lane,
     type: driving,
     parameters: {
       width: Lane.lane_width
     }
    }
  }
}
```

Abbildung 14: Eine Zuordnungskonfiguration für eine komplexe Straße.

Da man die Straße als *Road* definiert (`opendrive: road`), erwartet TSC2OpenX eine weitere Definition einer Fahrspur innerhalb des Road Objektes. Diese muss dann, wie beim einfachen Straßen-Mapping, vom Typ Lane sein.
In OpenDRIVE gibt es verschiedene Fahrspurtypen. Diese werden auch im Programm TSC2OpenX unterstützt. Für eine Lane gibt man zusätzlich den Typen der Fahrspur an, z.B. `type: driving` für eine befahrbare Spur oder `type: sidewalk` für einen Fußgängerweg.
Um komplexere Szenarien zu erstellen, wurde die Möglichkeit hinzugefügt, Straßenschilder zu generieren:

```
!!de.offis.vvm.tsc2osc.configuration.MappingConfiguration
mapping:
 Sign: {
    opendrive: signal,
    type: 205,
    subtype: -1,
    parameters: {
      x: Sign.X
    }
  }
```

Abbildung 15: Eine Zuordnungskonfiguration für ein Straßenschild.

In OpenDRIVE werden die Straßenschilder per *Type* und *Subtype* definiert. So ist in Deutschland z.B. das Schild mit `Type: 274` und `Subtype: 100` das Schild „Zulässige Höchstgeschwindigkeit 100 km/h“.

4.8 Limitierungen

Im Ansatz dieser Arbeit wurde das Programm TSC2OpenX für die Generierung von Verkehrsszenarien in der Domäne „Autobahn“ implementiert. Unter anderem sind beim Erstellen von TSCs auf Grund der Implementierung des Programms TSC2OpenX einige Annahmen, Limitierungen und Sonderfälle zu beachten. Ansonsten kann der Solver keine Lösung finden oder das Programm keine valide OpenX Dateien generieren.

- Das Straßennetz wird auf eine Straße begrenzt. Das schließt Kreuzungen, Kreisverkehr usw. aus.
- Fahrzeuge müssen sich im TSC immer von links nach rechts bewegen. D.h. es kann kein Gegenverkehr simuliert werden.
- Fahrzeuge haben einen begrenzten Lenkwinkel und können deshalb nicht abbiegen. Das schließt die Anwendung der Werkzeugkette in urbanen Gebieten, wie z.B. Kreuzungen aus.
- Bei der Generierung der OpenX Dateien werden die im TSC definierten Objekte in der im BulletinBoard festgelegten Reihenfolge generiert. Es muss darauf geachtet werden, dass z.B. ein Fahrstreifen-Objekt vor einem Straßenschild definiert wird. Ansonsten kann dem Straßenschild keine Straße zugeordnet werden, da diese noch

nicht generiert wurde.

- Falls ein Streuungsmaß berechnet werden soll, dürfen Objekte sich nur im Parameterraum $x > 0$ und $y > 0$ befinden, da sonst das in Unterabschnitt 6.1 definierte Streuuungsmaß nicht richtig errechnet werden kann.

5 Variationsmethoden zur Erzeugung verschiedener konkreter Szenarien

Da nun Simulationsdateien aus einem abstrakten Verkehrsszenario generierbar sind, ist das nächste Ziel, aus einem abstrakten Verkehrsszenario verschiedene Simulationen mit möglichst hoher Streuung zu erzeugen (Siehe Tabelle 1, Anforderung A6). Dafür werden im Folgenden vier verschiedene Ansätze untersucht:

1. Variation des Solver Seeds
2. Variation des Weltmodells
3. Variation existierender Constraints
4. Blocking Clauses

Alle genannten Varianten wurden in TSC2OpenX implementiert. Jedoch erzeugen nicht alle Variationsmöglichkeiten ein breites Spektrum an verschiedenen Lösungen, sondern unterscheiden sich meist nur marginal. Eine Bewertung der Variationsmethoden wird in Abschnitt 6 durchgeführt. Im Folgenden wird erläutert, wie die Konzepte realisiert wurden.

5.1 Variation des Solver Seeds

Der Z3 Solver (siehe Unterabschnitt 4.3) benutzt einen sogenannten „Seed“ um eine Lösung zu finden. Diesen kann man mit der Einstellung `random-seed` des Solvers manuell am Anfang der zu lösenden Formel setzen.

Wird in der Konfigurationsdatei von TSC2OpenX die Konfiguration `variation_type: seed` gesetzt, wird die Variation über Ändern des Solver Seeds angewandt. Bei jedem Durchlauf des Solvers wird dann der Seed zufällig neu gesetzt. Dabei wird nicht darauf

geachtet, ob ein vorheriger Seed schon einmal genutzt worden ist. Generationen von genau gleichen Simulationsdateien sind deshalb nicht auszuschließen.

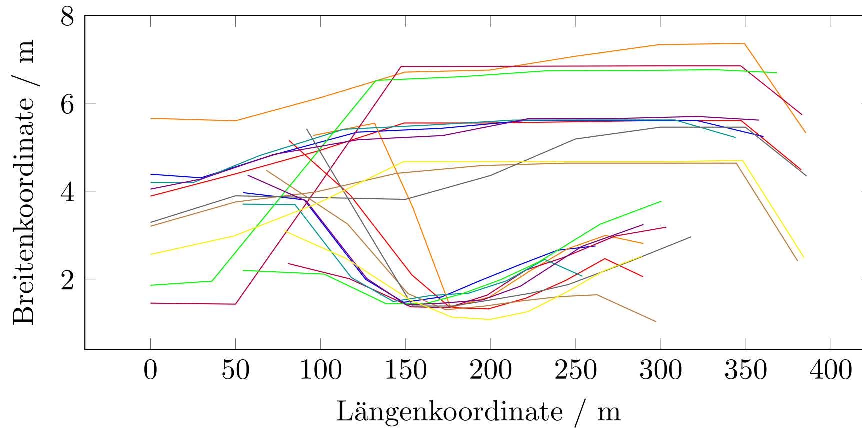


Abbildung 16: Visualisierung der Variation bei Änderung des Solver Seeds.

Die Abbildung zeigt ein straßenfestes Koordinatensystem. Die horizontale Achse beschreibt eine Längenkoordinate relativ zum Beginn einer Straße. Die vertikale Achse gibt die Breite einer Straße an. Beide Achsen sind jeweils in Metern angegeben.

In diesem Beispiel wurde ein einfaches Überholmanöver zehn Mal variiert. Die unteren Linien deuten die Trajektorien des Fahrzeugs an, das überholt wird. Die oberen Linien die Trajektorien des Fahrzeugs, welches überholt. Jede unterschiedliche Farbe der Trajektorie bedeutet eine eigene Variation, also Lösung des Solvers. Wie man sehen kann, unterscheiden sich die einzelnen Varianten oft nur marginal und durchlaufen ähnliche Koordinaten.

Zusammengefasst ist diese Methode der Variation die simpelste, bringt aber auch viele Nachteile mit sich. So hat der Anwender keinen Einfluss, welche Eigenschaften genau variiert werden sollen. Weiterhin ist es Zufall, ob sich die Lösung von vorherigen unterscheidet und in wie weit die Lösung abweicht.

5.2 Variation des Weltmodells

Im Weltmodell werden *globale Constraints* gesetzt, die für den gesamten Verlauf eines TSCs gültig sind. Diese können während der Variation beliebig verändert werden, solange die ursprünglich definierten Constraints nicht verletzt werden. Wird im Weltmodell z.B. festgelegt, dass ein Fahrzeug mit einer Geschwindigkeit im Intervall $80\text{km/h} \leq v \leq 180\text{km/h}$ fahren darf, kann dieses Intervall mit einer beliebigen Schrittweite eingeschränkt werden. Für gegebenes Beispiel kann die Einschränkung mit neuen Intervallen z.B. $[80, 100], [100, 120], ..., [160, 180]\text{km/h}$ ersetzt werden, ohne dass die beabsichtigte Semantik des Weltmodells verletzt wird. Für jede Variation des Weltmodells kann nun ein neues konkretes Szenario generiert werden.

TSC2OpenX implementiert die Methode der Weltmodell-Variation via der Konfiguration `variation_type: worldmodel_constraints`. Zusätzlich muss eine Liste von Eigenschaften angegeben werden, die variiert werden sollen. Dies geschieht mit dem Konfigurationspunkt `variation_properties`, der eine Liste an Objekt-Eigenschaften annimmt. Diese müssen im Format `[Objekt-Name].[Eigenschaft]` definiert werden. Zusätzlich muss über den Konfigurationspunkt `num_of_initiations` angeben werden, wie viele Variationen erzeugt werden sollen. Das Programm wählt dann automatisch die passende Schrittweite, um eine vollständige Abdeckung der Constraints zu erzielen. Möchte man also die Geschwindigkeit eines *Car*s zehn Mal im Weltmodell variieren, muss folgende Konfiguation getroffen werden:

```
variation_type: worldmodel_constraints,
num_of_initiations: 10,
variation_properties: [
  Car.v
]
```

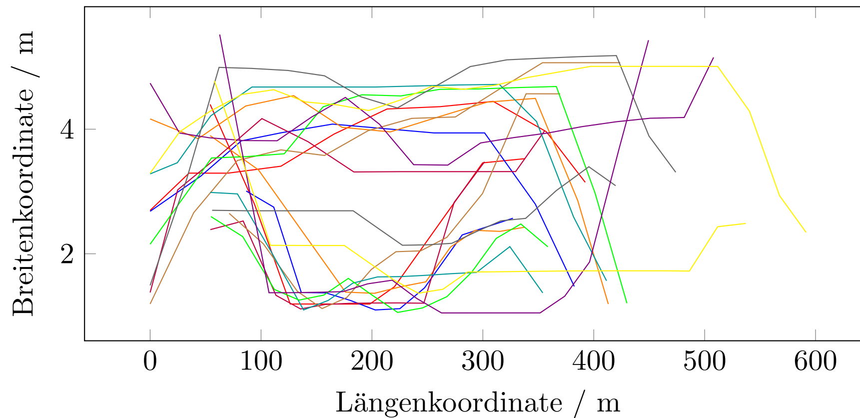
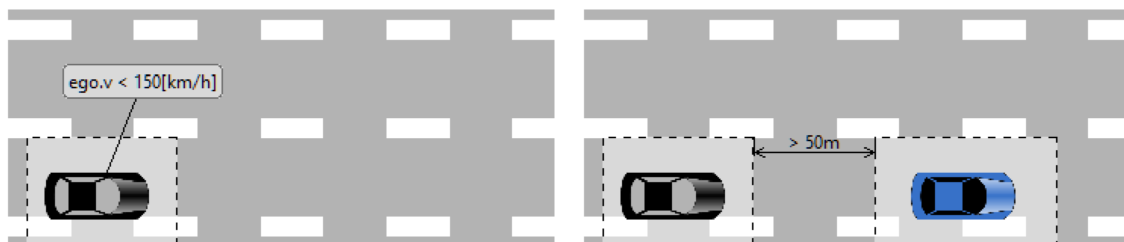


Abbildung 17: Visualisierung der Variation des Weltmodells.

Kreuzkombinationen werden nicht berücksichtigt, d.h. sollte ein $Car.v$ und $Bus.v$ variiert werden, werden keine Constraints wie z.B. $Car.v \in [80, 100] \wedge Bus.v \in [80, 90]$ kombiniert, sondern einzeln betrachtet.

5.3 Variation existierender Constraints

Neben globalen Constraints können auch *lokale Constraints* direkt im TSC variiert werden. Dafür werden die einzelnen Invariantenknoten variiert. Es werden zwei Variationsmethoden betrachtet: Die Variation bestehender Constraints von Objekten durch `Attached Predicates` (Abbildung 18a) und die Variation bestehender Constraints zwischen Objekten durch `Distance Arrows` (Abbildung 18b).



(a) Verwendung eines `Attached Predicates`. (b) Verwendung eines `Distance Arrows`.

Abbildung 18: Variierbare Definitionsmethoden von lokalen Constraints.

Die Variation existierender Constraints kann mit der Konfiguration `variation_type: existing_annotation_constraints` angewendet werden. Das Programm variiert alle sich

im TSC befindlichen `Attached Predicates` und `Distance Arrows`. Sollen bestimmte Constraints nicht variiert werden, so können diese mit dem Konfigurationspunkt `exclude_predicates` ausgeschlossen werden. Dort wird eine Liste mit den Identifikationen der Constraints angegeben, welche nicht variiert werden sollen:

```
variation_type: existing_annotation_constraints,
num_of_initiations: 10,
variation_properties: [
Car.v
],
exclude_predicates: [
  _FjKGOMnREEuT4ZYDTWbm3Q,
  _gWiCsMnREEuT4ZYDTWbm3Q
]
```

Hier sieht man eine beispielhafte Konfiguration, in der die Geschwindigkeit des „Car“s zehn Mal variiert werden soll. Weiterhin werden nach der Konfiguration zwei bestimmte lokale Constraints ausgeschlossen. Wie in 5.2 wird die Schrittweite der Belegung der Constraints dynamisch abhängig von der Anzahl der Initiierungen `num_of_initiations` gewählt. Werden passende Constraints im Weltmodell definiert, liegen die möglichen Wertebelegungen jeweils im Intervall zwischen lokal definiertem Constraint und den globalen Constraints. Andernfalls wird ein Intervall um den lokal definierten Constraint gewählt.

Werden im kompletten TSC keine `Attached Predicates` definiert, werden nur die `Distance Arrows` variiert, auch wenn diese keinen Einfluss auf die Eigenschaften der `variation_properties` haben.

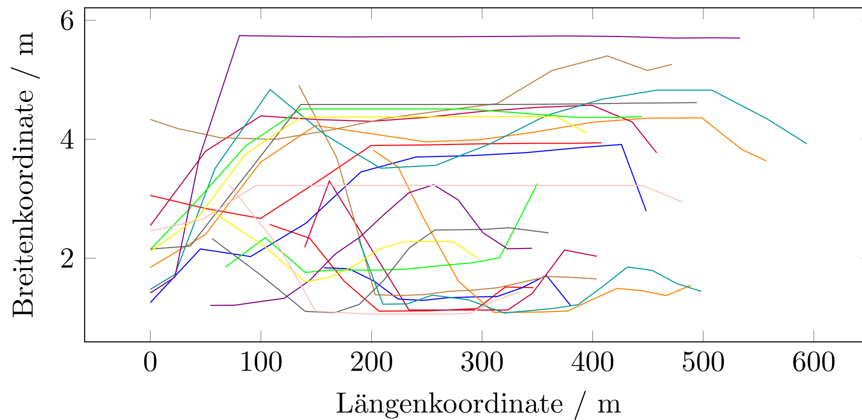


Abbildung 19: Visualisierung der Variation existierender Constraints.

Eine weitere Methode TSCs lokal zu variieren, besteht in der Möglichkeit neue Constraints in die Invariantenknoten einzufügen. Beispielsweise kann man mit dieser Methode bestimmte Werte für Eigenschaften an bestimmten Zeitpunkten forcieren. Diese Methode wurde noch nicht implementiert und wird auch im Rahmen dieser Arbeit nicht weiter betrachtet.

5.4 Variation via Blocking Clauses

Wenn der Solver eine Lösung findet, kann man im nächsten Solver-Durchlauf explizit verlangen, dass die gefundene Lösung nicht noch einmal gefunden werden soll. Im Falle der Fahrzeug-Trajektorien exkludiert man alle Stützpunkte der Trajektorie mit einem kleinen Puffer.

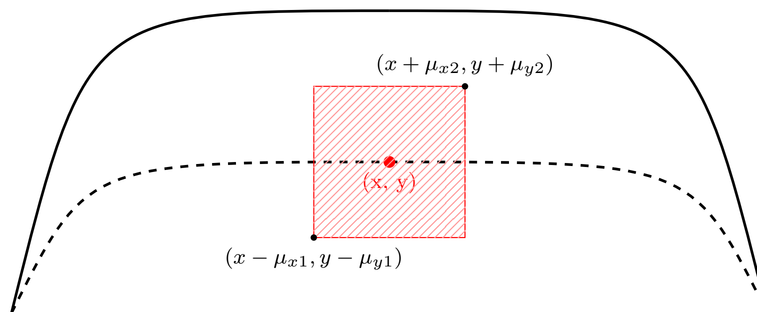


Abbildung 20: Umsetzung einer Blocking Clause für eine Trajektorie.

Im besten Fall bedeutet das, dass der Solver eine neue Trajektorie findet, welche

außerhalb des blockierten Bereichs liegt. Zu beachten ist, dass der Solver aber auch einen Stützpunkte jeweils vor und hinter dem blockierten Bereich wählen kann. So erfüllt der Solver zwar die Bedingungen, die Trajektorie führt dann trotzdem durch den gesperrten Bereich. In Abbildung 20 wird ein Stützpunkt der vorherigen Trajektorie (gestrichelte Linie) gesperrt. Die neu gefundene Trajektorie verläuft oberhalb der alten.

Zum Exkludieren eines Stützpunktes benötigt der Solver eine aussagenlogische Formel

$$b_n := \neg((\text{object}.x > x - \mu_{x1}) \wedge (\text{object}.x < x + \mu_{x2}) \wedge (\text{object}.y > y - \mu_{y1}) \wedge (\text{object}.y < y + \mu_{y2})) \tag{1}$$

Es wird dem Objekt *object* also verboten, sich durch den vorher bereits durchquerten Bereich erneut zu bewegen. Es wird nun eine weitere Einschränkung getroffen, dass sich das Objekt zu einem bestimmten Zeitpunkt nicht in diesem Bereich befinden darf:

$$c_n := (time = n) \implies b_n \tag{2}$$

Die Formeln für jeden Zeitpunkt des Szenarios (Gleichung 2) können dann konjugiert werden, um die vollständige Trajektorie auszuschließen $bc_n := c_1 \wedge c_2 \wedge ... \wedge c_n$. Für jedes weitere Objekt und Szenario kann diese weiter konjugiert werden $bc_{ges} := bc_1 \wedge bc_2 \wedge ... \wedge bc_n$.

TSC2OpenX unterstützt Blocking Clauses und ist mittels der Konfigurationsmöglichkeit `variation_type: block_surrounding_solutions` aktivierbar.

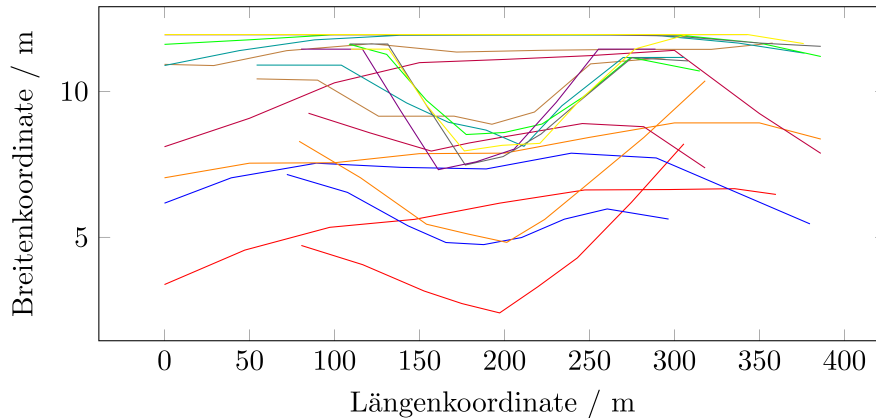


Abbildung 21: Visualisierung der Variation mittels Blocking Clauses.

In Abbildung 21 sieht man gut, dass es immer schwieriger wird, valide Trajektorien zu finden: Durch das Hinzufügen zusätzlicher Constraints wird immer weniger der eigentlichen Straßenbreite für neue Variationen benutzt, da ein vorheriges Fahrzeug dort bereits lang gefahren ist. So sieht man, dass die rote Trajektorie als erstes erzeugt worden ist und alle anderen Lösungen sich immer weiter nach oben verschieben.

Um trotzdem noch möglichst viele verschiedene Szenarien zu generieren, wird diese Methode entschärft. Erstens werden jeweils sehr kleine $\mu \leq 0.01$ gewählt, damit Fahrzeuge noch relativ dicht an vorherigen Trajektorien vorbei fahren dürfen. Zweitens wird nicht zu jedem Zeitpunkt die Gleichung 2 c_n aufgestellt, sondern nur zu jedem zweiten/dritten/.../n-ten Zeitpunkt.

Das Erstellen von Blocking Clauses ist auch auf andere Objekte und deren Eigenschaften anwendbar. Im Rahmen dieser Arbeit wird sich auf die Variation der Trajektorien beschränkt.

6 Evaluation der Werkzeugkette

Zur Erprobung der Werkzeugkette werden verschiedene TSCs erstellt und simuliert. Bei jedem TSC-Beispiel wird die Komplexität des Szenarios erhöht. Für jedes Beispiel wird jede Variationsmethode *seed* (Unterabschnitt 5.1), *worldmodel_constraints* (Unterabschnitt 5.2), *existing_annotation_constraints* (Unterabschnitt 5.3) und

block_surrounding_solutions (Unterabschnitt 5.4) jeweils drei Mal für $n = 10, 100, 1000$ durchgeführt. Es wird die Laufzeit des Programms TSC2OpenX gemessen und ein Maß zur Streuung der Variationen berechnet. Für alle Beispiele wird das während der Arbeit entwickelte Weltmodell verwendet.

Für die Methode *block_surrounding_solutions* wurde für jedes $\mu = 0.01$ gewählt. Weiterhin wird $t = 0 \land 0 \equiv t \mod 3$ nicht blockiert, damit mehr Lösungen gefunden werden können.

Die folgenden Messungen wurden auf einem Windows 10 Computer mit einem Intel Core i9-9900KF und 24 GB RAM durchgeführt. Es wurde die 64 Bit Version 4.6.0 des Z3 Solvers benutzt.

6.1 Definition eines Streuungsmaßes zur Evaluation der Variationsmethoden

Damit die Variationsmethoden verglichen werden können, muss gemessen werden, in wie weit sich die einzelnen Szenarien voneinander unterscheiden. Je höher der Unterschied, desto besser ist die Variationsmethode.

Um zu verstehen, wie ein Streuungsmaß für die Variationsmethoden definiert werden kann, wird zuerst das Format der Ausgabe des Solvers erläutert. Dazu wird das Beispiel aus Abbildung 16 herangezogen. Zur Vereinfachung wird sich auf zwei Variationen beschränkt:

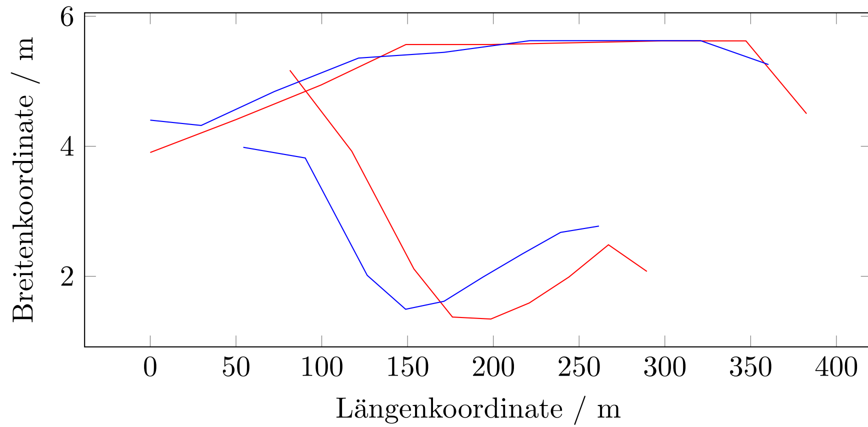


Abbildung 22: Vereinfachte Variationstrajektorien.

Der Solver gibt für eine Reihe an Zeitpunkten unter anderem aus, an welcher Position sich jedes Fahrzeug befindet. Zur weiteren Vereinfachung wird im Folgenden nur jeder dritte Zeitpunkt betrachtet. Die Trajektorie des *ego*-Fahrzeugs befindet sich jeweils über der des *other*-Fahrzeugs:

			Zeitpunkt t				
Variation	Objekt	Eigenschaft	0	3	6	9	12
rot	ego	x	0	99.63	149.06	248.35	347.23
		y	3.9	4.94	5.56	5.62	4.5
	other	x	81.39	153.70	198.57	244.04	289.51
		y	5.168	2.11	1.34	1.98	2.07
blau	ego	x	0.0	72.36	171.27	270.94	360.40
		y	4.40	4.84	5.44	5.62	5.25
	other	x	54.26	126.48	171.25	216.75	261.57
		y	3.98	2.01	1.61	2.34	2.77

Man kann sehen, dass sich jedes Fahrzeug zu jedem Zeitpunkt an unterschiedlichen x- und y-Koordinaten befinden kann. Es wurde daher entschieden, die x- und y-Koordinaten unabhängig voneinander zu betrachten und jeweils die Varianz aller Variationen-Koordinaten zu jedem Zeitpunkt t zu ermitteln. Es wird also die Stichprobenvarianz für ego.x, ego.y, other.x und other.y wie folgt berechnet:

$$Var(x,t) = \frac{1}{n-1} \sum_{i=0}^{n} (x_i - \overline{x_t})^2. \tag{3}$$

Dabei bildet $\overline{x}$ den arithmetischen Mittelwert der Stichprobe $x_1, \ldots, x_n$. n beschreibt die Anzahl der Stichprobenelemente. x für ego.x zum Zeitpunkt $t = 3$ ist z.B. $[99.63, 72.36]$.

Aus der Varianz einer Trajektorienvariation kann nun der Variationskoeffizient v errechnet werden, ein relatives Streuungsmaß:

$$v(x,t) = \frac{\sqrt{Var(x,t)}}{\overline{x_t}}. \tag{4}$$

Da ein Variationskoeffizient nur für $\overline{x} \neq 0$ bestimmt werden kann, wird eine Limitierung eingeführt, dass sich Objekte der OpenX-Dateien nur im Parameterraum $x > 0$ und $y > 0$ bewegen dürfen. Der Variationskoeffizient ist ein Intervall [0.0, 1.0]. Je höher der Variationskoeffizient, desto mehr unterscheiden sich die verschiedenen Trajektorien voneinander.

Der Variationskoeffizient ist von der Anzahl der Werte n abhängig. Deshalb können unterschiedlich große Datensätze nicht miteinander verglichen werden [20, p. 70]. Dies kann auftreten, falls sich zu einem späten Zeitpunkt t weniger Fahrzeuge bewegen, als zu einem vorherigen Zeitpunkt $t - 1$. Damit man den Variationskoeffizienten trotzdem verwenden kann, muss dieser als Nächstes normiert werden:

$$v^*(x,t) = \frac{v(x,t)}{\sqrt{n}}. \tag{5}$$

Zum Schluss muss nun aus jedem normierten Variationskoeffizienten für x- und y zu jedem Zeitpunkt t für jedes Fahrzeug *entity* das arithemtische Mittel berechnet werden. Dies bildet den finalen Variationskoeffitienten, die Gesamtvarianz s, zur Beschreibung der Variation der Trajektorien:

$$s = \frac{\sum_{t=0}^{T} \sum_{\text{entity} \in \text{entities}} \sum_{m \in \{\text{entity}.x, \text{entity}.y\}} v^*(m,t)}{T \cdot |entities| \cdot 2} \tag{6}$$

In dem oben erläuterten Beispiel wäre entities $= \{\text{ego}, \text{other}\}$.

Derzeit wird nur das Streuungsmaß der einzelnen Trajektorien berechnet. Andere Variationsmöglichkeiten, wie Straßen- oder Fahrzeuglänge und -breite, werden nicht beachtet. Je höher die Gesamtvarianz im Intervall $[0.0, 1.0]$, desto besser ist die Variationsmethode, da sich die einzelnen Trajektorien der Fahrzeuge mehr unterscheiden.

6.2 Überholvorgang auf einer zweispurigen Straße

Das erste TSC ist ein simples Überholmanöver auf einer zweispurigen Straße. Das *ego*-Fahrzeug soll mindestens 50m hinter dem anderen Fahrzeug starten und sich am Ende des Szenarios mindestens 50m vor dem anderen Fahrzeug befinden.

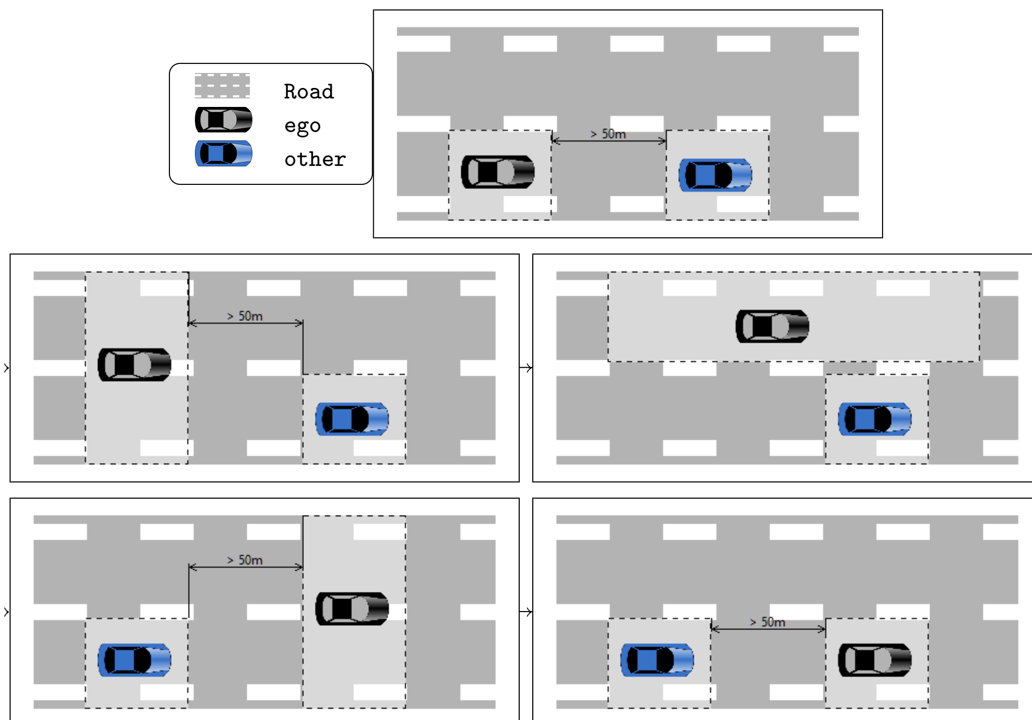


Abbildung 23: TSC: ein Überholvorgang auf einer zweispurigen Straße.

Der Solver kann zwischen linker und rechter Spur unterscheiden indem man ein Fahrzeug entweder an die linke oder rechte Seite der Straße platziert.

Tabelle 2: Variationsmessungen für einen Überholvorgang auf einer zweispurigen Straße.

Variationsmethode	Initialisierungen n	Ausführungszeit t	Gesamtvarianz s
seed	10	22.31s	0.06409
	100	3m 32s	0.01964
	1000	8m 35.97s	0.00650
worldmodel_constraints	10	59.73s	0.0932
	100	8m 52.95s	0.06324
	1000	42m 34.85	0.03871
existing_annotation_constraints	10	24.01s	0.06391
	100	3m 23.98s	0.02798
	1000	33m 50.95s	0.00961
block_surrounding_solutions	10	20.76s	0.05175
	22	59.39s	0.03461
	1000	-	-

Für dieses TSC wurden nur 22 Variationen mit der *block_surrounding_solutions* gefunden.

6.3 Überholvorgang auf einer mehrspurigen Straße mit verschiedenen Fahrzeugen

Komplizierter wird es, wenn man mehr als zwei Spuren benutzt. Dann reicht ein einziges Straßenobjekt nicht mehr aus, da man die mittlere Spur weder durch den linken noch den rechten Rand der Straße identifizieren kann. Stattdessen muss für jede Spur ein eigenes Objekt im BulletinBoard registriert werden. Damit eine Zuordnung stattfinden kann, welche Spur zu welcher Straße gehört, benötigt man zusätzlich ein Wrapper-Objekt, welche alle Spuren beinhaltet.

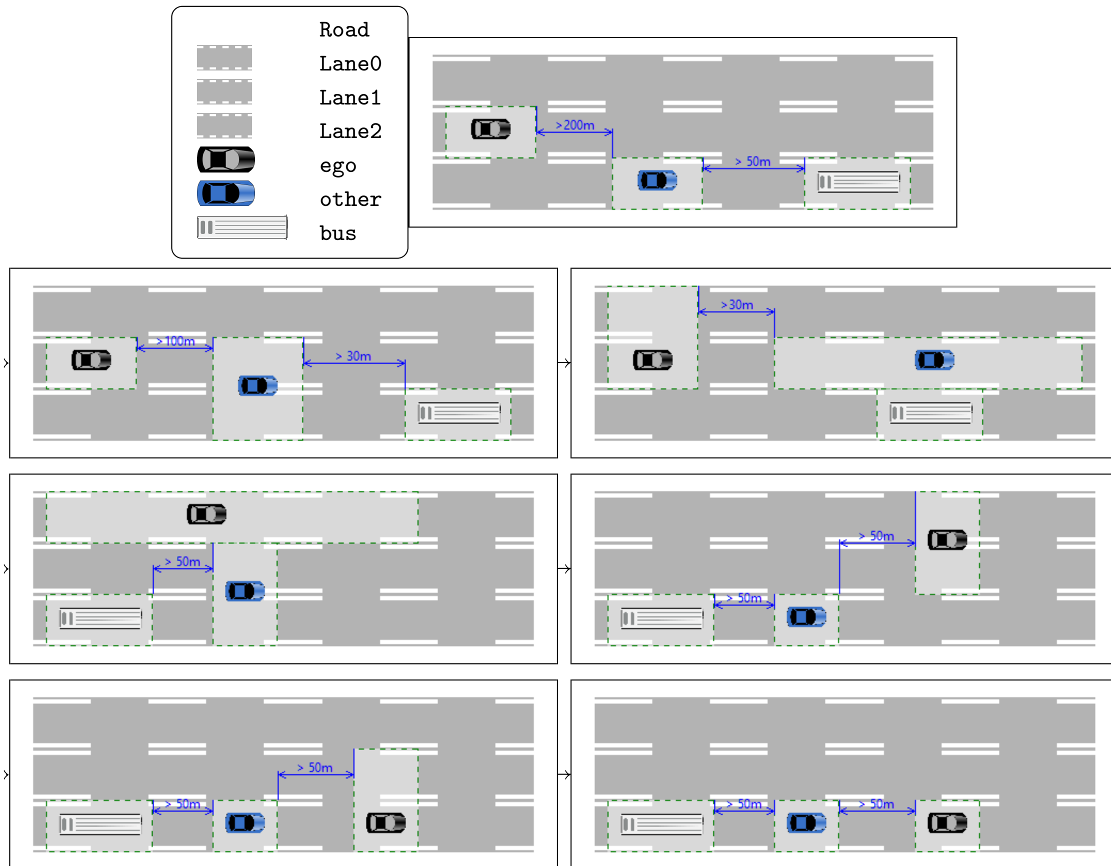


Abbildung 24: TSC: Überholvorgänge auf einer mehrspurigen Straße mit verschiedenen Fahrzeugen.

In diesem Beispiel befahren drei verschiedene Fahrzeuge die Straße: ein Bus und zwei Autos. Das *other* Auto überholt den Bus. Währenddessen überholt das *ego* Fahrzeug das *other* Fahrzeug, sodass sich das *ego* Fahrzeug am Ende des Szenarios vor den beiden anderen Fahrzeugen befindet. Da zwei Überholvorgänge gleichzeitig stattfinden, werden alle drei Fahrbahnen benutzt.

Tabelle 3: Variationsmessungen für einen Überholvorgang auf einer mehrspurigen Straße mit verschiedenen Fahrzeugen.

Variationsmethode	Initialisierungen n	Ausführungszeit t	Gesamtvarianz s
seed	10	1m 10.27s	0.05037
	100	11m 21.61s	0.01827
	1000	1h 53m 3.35s	0.00566
worldmodel_constraints	10	2m 2.05s	0.07015
	100	20m 56s	0.06018
	1000	3h 54m 29.73s	0.03290
existing_annotation_constraints	10	18.7s	0.03665
	100	2m 34.85s	0.02111
	1000	25m 51.36s	0.00716
block_surrounding_solutions	6	1m 12.57s	0.07959
	100	-	-
	1000	-	-

Für dieses TSC wurden nur 6 Variationen mit der *block_surrounding_solutions* gefunden.

6.4 Straße mit einer zulässigen Höchstgeschwindigkeit

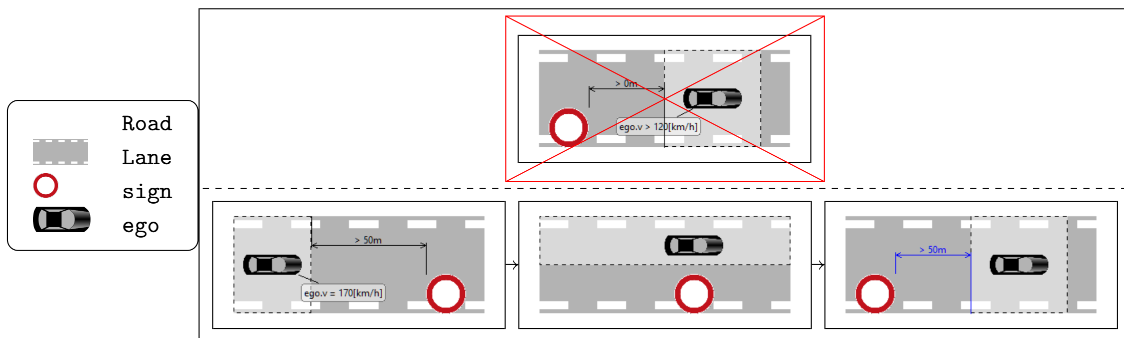


Abbildung 25: TSC: Setzen einer zulässigen Höchstgeschwindigkeit.

In diesem Beispiel wird die Komplextität des TSC deutlich erhöht. Es wird eine *And Node* benutzt, d.h. zwei unabhängig voneinander vorliegende Szenariosequenzen sind innerhalb

des TSCs gleichzeitig gültig. Die *And Node* wird durch die gestrichelte Linie zwischen oberer und unterer Szenariosequenz angedeutet.
Zuerst wird die obere Szenariosequenz betrachetet, in der eine weitere Besonderheit zu finden ist: eine *Not Node*. Die *Not Node* negiert den Inhalt des Invariantenknoten. Während der Invariantenknoten zeigt, dass ein Fahrzeug mit einem Abstand von mehr als 0 Meter zu dem Schild, mehr als 120 km/h fahren darf, bedeutet der negierte Invariantenknoten, dass das Fahrzeug nicht mehr als 120 km/h fahren darf. Es wird also eine zulässige Höchstgeschwindigkeit gesetzt, sobald an dem Schild vorbei gefahren wird. Es wurde eine Verkehrsregel aufgestellt, welche für den Rest des Szenarios gültig sein wird.
Die untere Szenariosequenz deutet an, dass das Fahrzeug an dem Schild vorbeifährt. Dies ist also das eigentliche Verkehrsszenario. Besonders ist, dass im ersten Invariantenknoten die Geschwindigkeit des *ego* Fahrzeugs auf 180 km/h gesetzt wird. So muss das Fahrzeug im Verlauf des Szenarios auf jeden Fall verzögern.

Zum Anwenden der Variationsmethode *existing_annotation_constraints* muss die Identifikation der Annotation der maximalen Geschwindigkeit, sowie der Abstand von $> 0m$ des oberen Invariantenknoten auf die *exclude_predicates* Liste der Hauptkonfigurationsdatei hinzugefügt werden. Ansonsten würde die zulässige Höchstgeschwindigkeit im Szenario variiert werden.

Tabelle 4: Variationsmessungen für ein Szenario auf einer Straße mit einer zulässigen Höchstgeschwindigkeit.

Variationsmethode	Initialisierungen n	Ausführungszeit t	Gesamtvarianz s
seed	10	24.51s	0.01399
	100	2m 31.57s	0.00743
	1000	26m 6.19s	0.00229
worldmodel_constraints	10	-	-
	100	-	-
	1000	-	-
existing_annotation_constraints	10	18.28s	0.06151
	100	2m 32.45s	0.02155
	1000	25m 25.02s	0.00758
block_surrounding_solutions	10	16.27s	0.0384
	64	1m 44.41s	0.02468
	1000	-	-

Für dieses TSC konnte die *worldmodel_constraints*-Methode nicht angewendet werden. Es wurden nur 64 Variationen mit der *block_surrounding_solutions* gefunden.

6.5 Inkonsistentes abstraktes Szenario

In Abbildung 26 wird gezielt ein nicht lösbares Szenario als Eingabe des Programms TSC2OpenX betrachtet. Wie in Unterabschnitt 6.4 werden zuerst wieder Regeln für ein definiertes Straßenschild aufgestellt. Für *ego* und *bus* wird festgelegt, dass die Fahrzeuge hinter dem Schild nicht mehr als 80 km/h fahren dürfen. Das Szenario ist nun so aufgebaut, dass *bus* am Schild vorbei fährt und *ego* daraufhin versucht, den Bus zu überholen. Da im Weltmodell die minimale Geschwindigkeit der Fahrzeuge auf 80 km/h festgesetzt wurde, müssen beide Fahrzeuge exakt 80 km/h fahren. Der Überholvorgang des *ego* Fahrzeugs ist nicht mehr möglich, da keine Geschwindigkeitsdifferenz gegeben ist. Die durch den letzten Invariantenknoten gestellte Anforderung, dass sich *ego* vor *bus* befinden muss, ist somit nicht erfüllbar.

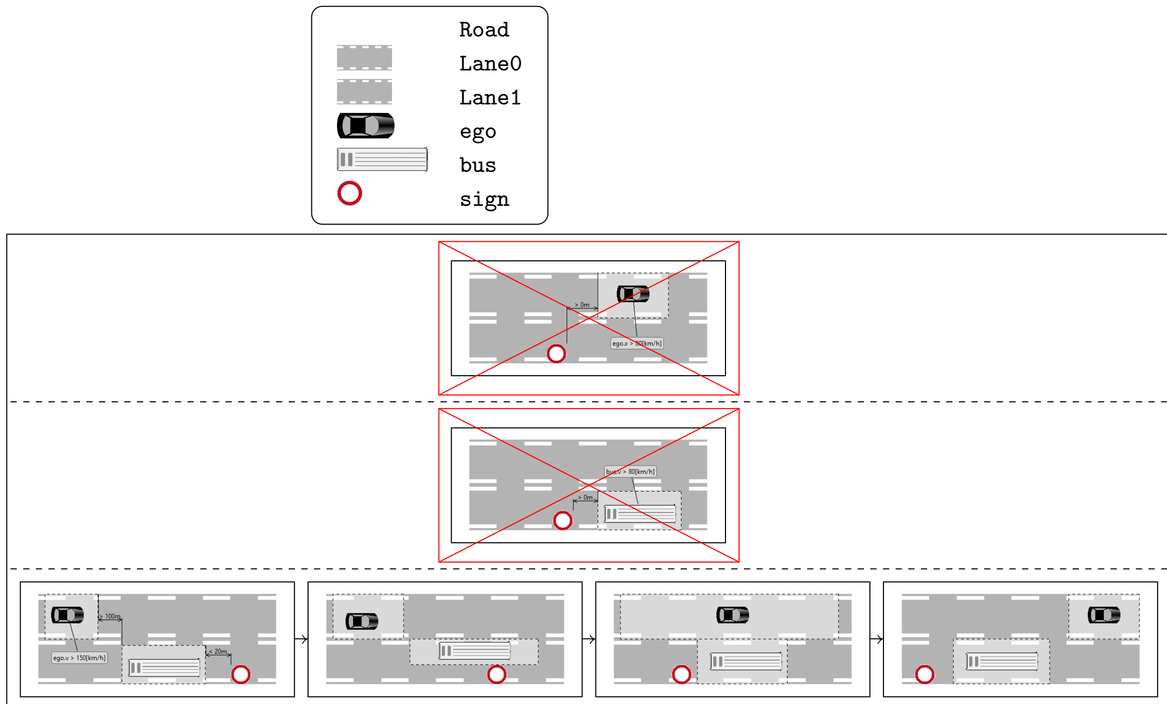


Abbildung 26: TSC: Inkonsistentes Szenario.

Der Solver kann die Inkonsistenz nicht von allein erkennen und versucht normal, wie in den anderen Beispielen, eine Lösung zu finden. Nach 30 Minuten wurde das Programm manuell gestoppt. Der Solver hat in dieser Zeit nicht die maximale Schrittanzahl erreicht.

6.6 Auswertung

Zur Auswertung der Variationsmethoden werden Tabelle 2, Tabelle 3 und Tabelle 4 herangezogen.

Bei der Durchführung der Evaluation war festzustellen, dass das hier definierte Streuungsmaß sehr leicht verfälscht werden kann: Nimmt man den Fall an, dass sich zu einem beliebigen Zeitpunkt nur noch ein Fahrzeug bewegt, ist $Var(x,t)$ nicht definiert (da $n = 1$ und $\frac{1}{n-1} = \frac{1}{0}$) und wird deshalb nicht in die Gesamtvarianz eingerechnet. Andererseits ist dies der beste Fall, der auftreten kann: Die Variation des TSCs ist zu diesem Zeitpunkt maximal, und $Var(x,t)$ sollte dementsprechend 1 sein. Wie man das

Streuungsmaß verbessern kann, damit diese Sonderfälle mit einbezogen werden können, müsste weiter untersucht werden. Der Fall, dass ein Fahrzeug in nur einer Variation fährt, kommt häufig in der *worldmodel_constraints*- und *existing_annotation_constraints*-Methode vor, da mit der Geschwindigkeits- oder Abstandsvariation Fahrzeuge in einigen Varianten langsamer fahren und damit länger unterwegs sind.

Fast alle hier verwendeten Variationsmethoden skalieren in ihrer Ausführungszeit linear. Die vollständige Ausführungszeit einer Variationsmethode hängt von der Komplexität des TSCs und der Anzahl der Initialisierungen ab. Als Beispiel ist das TSCs aus Abbildung 24 (im Folgenden *complex*-TSC) gegenüber des TSCs aus Abbildung 23 (im Folgenden *simple*-TSC) wesentlich komplexer. So benutzt das *complex*-TSC vier Entitäten mehr und enthält zwei Invariantenknoten mehr. Wie in den Tabellen abzulesen ist, braucht das Generieren von 10 *seed*-Variationen vom *complex*-TSC circa dreifach so lange als die Generierung des *simple*-TSCs. Der TSC aus Abbildung 25 (im Folgenden *sign*-TSC) ist zwar von der Logik her komplexer, (es enthält weniger Invariantenknoten, jedoch zwei Logikknoten), ist aber mit der Laufzeit des *simple*-TSC vergleichbar. Vermutlich, weil ähnlich viele Eigenschaften über das Bounded Model Checking berechnet werden müssen.

Die *block_surrounding_solutions*-Methode skaliert nicht linear in ihrer Ausführungszeit (siehe Tabelle 2). Das lässt sich damit erklären, dass das gegebene Problem, das der Solver lösen muss, für jede Variation komplexer wird. Weitherhin kann diese Methode für gegebene TSCs nicht annähernd 100 Lösungen finden. Auch eine weitere Entschärfung dieser Methode (z.B. noch mehr Zeitschritte ausschließen) wird dem vermutlich nicht entgegenwirken.

Anscheinend gibt es einen wesentlichen Unterschied zwischen der Variation von lokalen und globalen Constraints. So braucht die Generation mit der *worldmodel_constraints*-Methode auffällig länger als die *existing_annotation_variation*-Methode. Dies kann daran liegen, dass durch eine globale, langsame Geschwindigkeit der Fahrzeuge der Solver mehr BMC-Schritte ausführen muss.

Wie in Tabelle 3 auffällt, konnte die *worldmodel_constraints*-Methode für dieses TSCs nicht ausgeführt werden. Das liegt daran, dass im TSC explizit gefordert wird, dass das

ego-Fahrzeug zu Beginn des Szenarios 170km/h fährt. Da in den Testfällen nur die Geschwindigkeit der Fahrzeuge variiert wurde, wurden die Geschwindigkeitsintervalle schnell so gering, dass 170 km/h nicht mehr im Intervall lag. Die TSC-Variationen konnten vom Solver nicht mehr gelöst werden. Es zeigt sich also, dass nicht alle Variationsmethoden für alle Arten von TSC geeignet sind. Eine Lösung wäre hier vor der Variation des Weltmodells das zu variierende TSC zu untersuchen und die Intervalle auf diese Art von Constraint anzupassen.

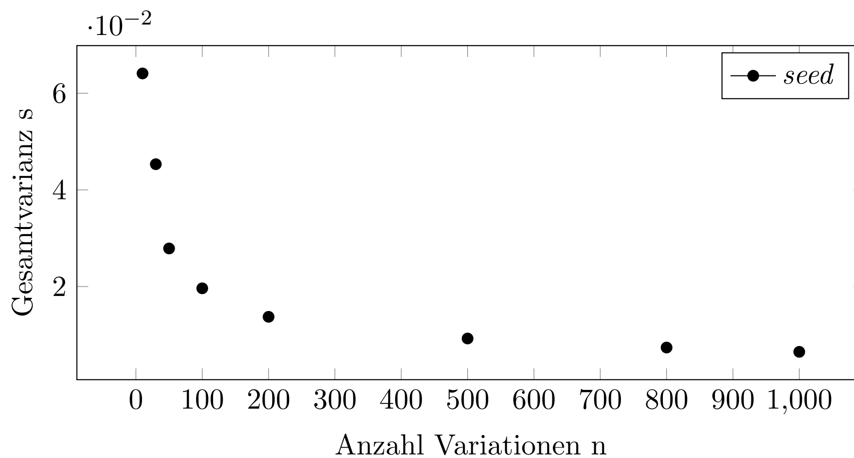


Abbildung 27: Anwendung der *seed*-Variationsmethode auf das *simple*-TSC.

In Abbildung 27 wurde die *seed*-Variationsmethode für verschiedene n durchgeführt. Hier kann man beobachten, dass sich die Gesamtvarianz s mit steigender Anzahl an Initialisierungen n asymptotisch dem Wert 0 nähert. Allgemein lässt sich das Verhalten bei allen hier eingeführten Variationsmethoden erkennen. Durch die in der Berechnung der Gesamtvarianz angewendeten Normierung lässt sich darauf schließen, dass im Verhältnis zu n immer weniger unterschiedliche Variationen generiert werden, bzw. sich alle Variationen nur wenig voneinander unterscheiden.

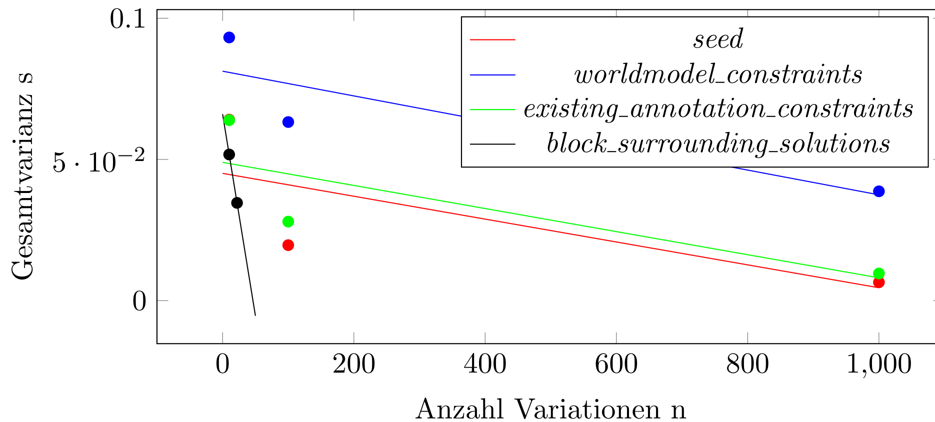


Abbildung 28: Vergleich aller Variationsmethoden anhand des *simple*-TSCs.

Die Abbildung 28 vergleicht die verschiedenen Variationsmethoden anhand des *simple*-TSCs. Die Punkte stellen jeweils die Messwerte der Variationen dar. Die Linien sind Regressionsgeraden. Zwar empfiehlt es sich nach Abbildung 27 eine hyperbolische Regression durchzuführen, um die Gesamtvariationsunterschiede jedoch leichter darzustellen, wurde eine lineare Regression durchgeführt. Die *block_surrounding_solutions* konnte leider nur 22 Variationen erzeugen und erreicht trotzdem, im Vergleich zu den anderen Methoden, keine hohe Gesamtvarianz. Auch für die beiden anderen TSCs wurden maximal 64 Variationen erzeugt. Es wurde von der Blocking Clauses Methode zwar die höchste Gesamtvarianz von ~ 0.08 erreicht (siehe *complex*-TSC), da aber nur 6 statt 10 Messwerte vorliegen, ist die eigentliche Gesamtvarianz vermutlich geringer. Damit erfüllt diese Methode nicht die Erwartungen: da vorherige Lösungen ausgeschlossen werden, müsste die Gesamtvariation insgesamt höher sein, als die der anderen Methoden. Vermutlich wäre dies auch der Fall, wenn die Methode nicht entschärft worden wäre.

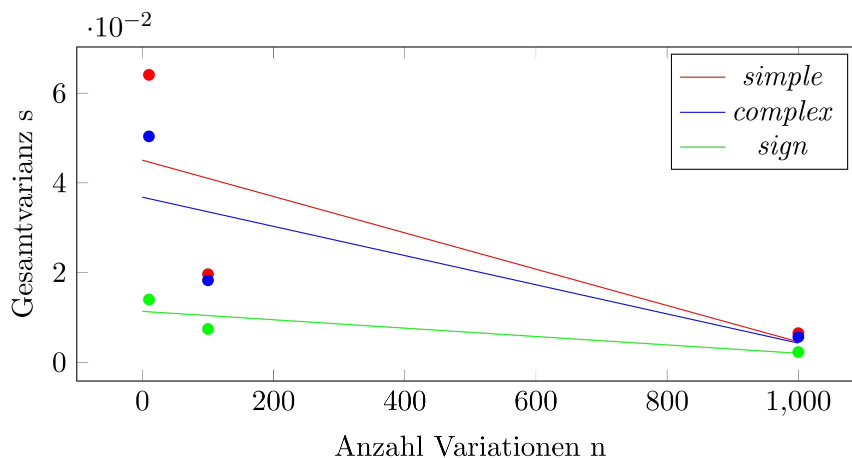


Abbildung 29: Variationsmethode *seed*.

Die Abbildung 29, Abbildung 30 und Abbildung 31 vergleichen die Variationsmethoden jeweils bezüglich der verschiedenen TSCs. Die Messungen des *simple*-TSCs ist in roter Farbe dargestellt, das *complex*-TSC in blau und das *sign*-TSC in grün. Anhand der Abbildungen kann man sehen, dass nicht jedes TSC gleich effizient mittels den Variationsmethoden variiert werden kann. Das *complex*-TSC lässt sich z.B. nicht gut mit der *seed*-Methode variieren, aber besser mit der *existing_annotation_constraints*-Methode.

Wie zu erwarten, schneidet die *seed*-Methode von allen Variationsmethoden auch schlecht ab: zwar werden zum Beginn der Variationserzeugung bei einigen TSCs relativ hohe Gesamtvarianzen erreicht, diese fallen jedoch mit größer werdenden n sehr schnell ab (siehe Tabelle 2). Das liegt daran, dass die Eigenschaften des TSCs nicht aktiv moduliert werden, sondern immer wieder das gleiche TSC anhand beeinflusster Heuristiken (siehe Unterabschnitt 4.3) gelöst wird.

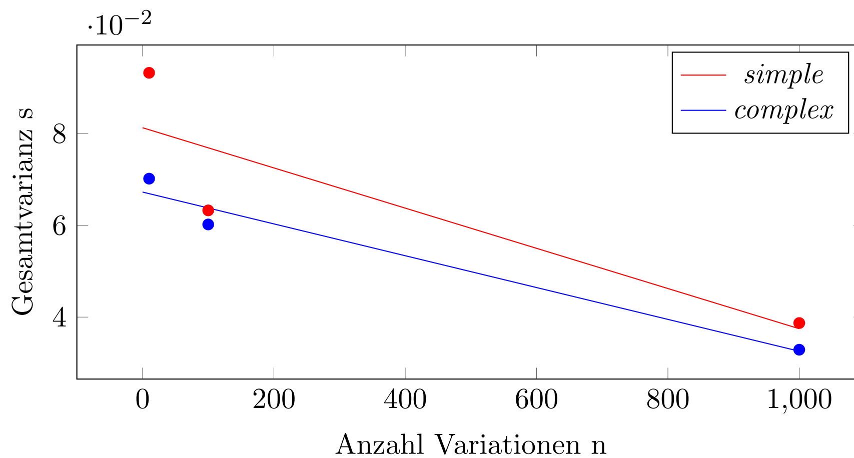


Abbildung 30: Variationsmethode *worldmodel_constraints*.

Im Vergleich zu den anderen Variationsmethoden erreicht die *worldmodel_constraints*-Methode die beste Gesamtvarianz. Wie benannt, muss man dafür eine längere Ausführungszeit erwarten und darauf achten, dass die Methode für das TSC überhaupt angewendet werden kann.

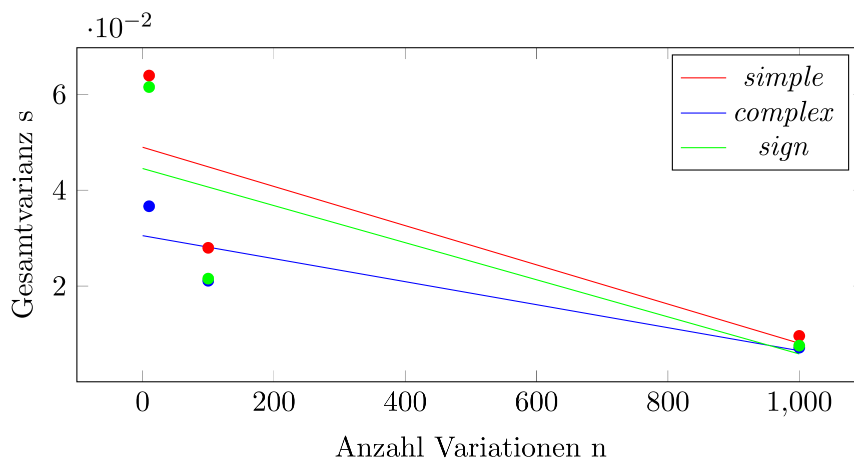


Abbildung 31: Variationsmethode *existing_annotation_constraints*.

Die *existing_annotation_constraints*-Methode liegt im Mittelfeld der Variationsmethoden. Sie schneidet in der Gesamtvarianz knapp besser als die *seed*-Variation ab, variiert die verschiedenen TSCs aber nicht über einen zufälligen Wert, sondern moduliert die Inhalte

des TSCs und ist damit besser kontrollierbar. Weiterhin wird die Gesamtvarianz der Methode leicht verfälscht und liegt deshalb eigentlich höher. Diese Methode ist gut einzusetzen, falls die *worldmodel_constraints*-Methode versagt.

Nach der Auswertung dieser Ergebnisse wird beschlossen, dass die *worldmodel_constraints*- und *existing_annotation_constraints*-Methoden in den beschriebenen Implementationen am effektivsten sind. Die *worldmodel_constraints*-Methode erzeugt, wenn diese anwendbar ist, bessere Variationen, während die *existing_annotation_constraints*-Methode stabiler gegenüber unterschiedlichen TSCs ist.

7 Zusammenfassung und Ausblick

Mit Hilfe der vorgestellten Werkzeugkette, bestehend aus den Programmen TSC Editor, TSC2OpenX und dem Simulator esmini, ist ein Nuter in der Lage, einfache Verkehrsszenarien in der Domäne Autobahn schnell zu definieren und sich verschiedene valide Versionen des gleichen Szenarios generieren zu lassen. Dafür implementiert das Programm TSC2OpenX insgesamt vier verschiedene Methoden, um unterschiedliche Simulationsdateien zu erstellen. Um die Effizienz dieser Verfahren zu messen, wurde ein Streuungsmaß, die *Gesamtvarianz*, definiert und ausgewertet. Nach dessen Auswertung wurde festgestellt, dass die *worldmodel_constraints*- und die *existing_annotation_constraints*-Methoden am effektivsten zum Variieren von TSCs geeignet sind. Enttäuschtend war die Anwendung von Blocking Clauses, um bereits befahrende Trajektorien auzuschließen. Hier müsste weiter untersucht werden, ob es vorteilhaft wäre, die Eigenschaften der Objekte ausgeschlossen werden, statt dessen Trajektorien.

Weiterhin kann das Variationsmaß weiter verfeinert werden: statt nur die Trajektorien der Fahrzeuge als Maß der Variation der Simulationen zu nehmen, könnten noch weitere Eigenschaften der Straße hinzugenommen werden. Zudem sollte das Variationsmaß stabiler gegenüber den genannten Sonderfällen werden.

Das Programm TSC2OpenX ist eine prototypische Realisierung zum Erzeugen von konkreten Verkehrsszenarien. Besonders an den genannten Limitierungen sollte man zukünftig weiterarbeiten, sodass diese weiter reduziert werden. Aber auch an den

Variationsmethoden selbst kann noch gefeilt werden: so kann daran gearbeitet werden, dass möglichst jede Methode auf jegliche Art von TSC angewandt werden kann.

Abkürzungsverzeichnis

TSC Traffic Sequence Chart

OSC OpenSCENARIO

ODR OpenDRIVE

esmini Environment Simulator Minimalistic

SMT satisfiability modulo theories

BMC Bounded Model Checking

Index

Bounded Model Checking, 17
BulletinBoard, 7

Constraint, 3, 5, 26, 27

Dynamik, 5, 17

Globaler Constraint, 26, 42

Invariantenknoten, 7

Lokaler Constraint, 27, 42

OpenDRIVE, 8
OpenSCENARIO, 9
OpenX, 8

Solver, 19, 32
Spatial View, 7
Symbol Dictionary, 6
Symbolbibliothek, 6

TSC Editor, 6

Weltmodell, 4
World model, 4

Z3 Solver, 16

Literatur

[1] Christian Amersbach und Hermann Winner. *Funktionale Dekomposition - Ein Beitrag zur Überwindung der Parameterraumexplosion bei der Validation von höher automatisiertem Fahren.* URL: https://www.uni-das.de/images/pdf/veroeffentlichungen/2018/03_Walting_Am_Wi_20180621.pdf (besucht am 06. 09. 2021).

[2] Jan Steffen Becker, Niklas Holtz, Linus Hestermeyer und Boris Wirtz. *Der TSC Editor ist eine interne Entwicklung von OFFIS e.V.*

[3] Jan Steffen Becker, Tjark Koopmann, Birte Kramer, Christian Neurohr, Lukas Westhofen, Boris Wirtz, Eckard Böde und Werner Damm. „Simulation of Abstract Scenarios: Towards Automated Tooling in Criticality Analysis“. In: 2021.

[4] Eckard Böde, Matthias Büker, Werner Damm, Martin Fränzle, Birte Kramer, Christian Neurohr und Sebastian Vander Maelen. *Identifikation und Quantifizierung von Automationsrisiken für hochautomatisierte Fahrfunktionen.* 2019. URL: https://www.pegasusprojekt.de/files/tmpl/pdf/PEGASUS_TechnicalReport_Automationsrisiken_17.07.2019.pdf (besucht am 25. 05. 2021).

[5] Statistisches Bundesamt. *Fehlverhalten der Fahrzeugführer bei Unfällen mit Personenschaden.* URL: https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Verkehrsunfaelle/Tabellen/fehlverhalten-fahrzeugfuehrer.html;jsessionid=84DF0299829A860B108084FC8D47EA09.internet741 (besucht am 25. 05. 2021).

[6] Christian Neurohr, Lukas Westhofen, Martin Butz, Martin Bollmann, Ulrich Eberle und Roland Galbas. „Criticality Analysis for the Verification and Validation of Automated Vehicles“. In: *IEEE Access* 9 (2021), S. 18016–18041. DOI: 10.1109/ACCESS.2021.3053159.

[7] Edmund M. Clarke, William Klieber, Miloš Nováček und Paolo Zuliani. „Model Checking and the State Explosion Problem“. In: (2011). URL: https://link.springer.com/chapter/10.1007/978-3-642-35746-6_1 (besucht am 12. 08. 2021).

[8] Werner Damm, Stephanie Kemper, Eike Möhlmann, Thomas Peikenkamp und Astrid Rakow. *Traffic Sequence Charts - From Visualization to Semantics.* Techn. Ber. Okt. 2017. URL: http://www.avacs.org/fileadmin/Publikationen/Open/avacs_technical_report_117.pdf.

[9] ASAM e.V. *ASAM OpenDRIVE®*. URL: https://www.asam.net/standards/detail/opendrive/ (besucht am 23. 08. 2021).

[10] ASAM e.V. *ASAM OpenSCENARIO®*. URL: https://www.asam.net/standards/detail/openscenario/ (besucht am 23. 08. 2021).

[11] ASAM e.V. *Association of Standardization of Automation and Measuring Systems*. URL: https://www.asam.net (besucht am 23. 08. 2021).

[12] esmini. *Environment Simulator Minimalistic - a basic OpenSCENARIO player*. URL: https://github.com/esmini (besucht am 25. 05. 2021).

[13] *Fahrerassistenzsysteme werden immer beliebter*. 2019. URL: https://www.springerprofessional.de/automatisiertes-fahren/fahrzeugsicherheit/fahrerassistenzsysteme-werden-immer-beliebter/16613236 (besucht am 21. 08. 2021).

[14] Eclipse Foundation. *Desktop IDEs*. URL: https://www.eclipse.org/ide/ (besucht am 02. 09. 2021).

[15] Eclipse Foundation. *Eclipse Modeling Framework*. URL: https://www.eclipse.org/modeling/emf/ (besucht am 29. 08. 2021).

[16] Eclipse Foundation. *OpenPASS*. URL: https://openpass.eclipse.org (besucht am 23. 08. 2021).

[17] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras und Cesare Tinelli. „DPLL(T): Fast Decision Procedures". In: *Computer Aided Verification*. Hrsg. von Rajeev Alur und Doron A. Peled. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, S. 175–188. ISBN: 978-3-540-27813-9.

[18] MSC.Software GmbH. *Virtual Test Drive*. URL: https://www.mscsoftware.com/de/virtual-test-drive (besucht am 23. 08. 2021).

[19] Nidhi Kalra und Susan M. Paddock. *Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?* RAND Corporation, 2016. URL: http://www.jstor.org/stable/10.7249/j.ctt1btc0xw.

[20] Wolfgang Kohn und Riza Öztürk. *Statitik für Ökonomen*. 2013. DOI: 10.1007/978-3-642-37352-7.

[21] Deutsches Zentrum für Luft- und Raumfahrt e. V. *SET LEVEL - Simulationsbasiertes Entwickeln und Testen von automatisiertem Fahren.* URL: https://setlevel.de (besucht am 25. 05. 2021).

[22] Alexander Pretschner, Florian Hauer und Tabea Schmidt. *Tests für automatisierte und autonome Fahrsysteme.* 2021. URL: https://link.springer.com/content/pdf/10.1007/s00287-021-01364-w.pdf (besucht am 21. 09. 2021).

[23] The jSMTLIB Project. *jSMTLIB Project.* URL: https://smtlib.github.io/jSMTLIB/ (besucht am 13. 09. 2021).

[24] The OpenSceneGraph Project. *OpenSceneGraph.* URL: http://www.openscenegraph.org (besucht am 13. 09. 2021).

[25] The YAML Project. *YAML: YAML Ain't Markup Language.* URL: https://yaml.org (besucht am 05. 09. 2021).

[26] CARLA Team. *CARLA: Open-source simulator for autonomous driving research.* URL: https://carla.org (besucht am 23. 08. 2021).

[27] Simon Ulbrich, Till Menzel, Andreas Reschka, Fabian Schuldt und Markus Maurer. *Defining and Substantiating the Terms Scene, Situation, and Scenario for Automated Driving.* 2015. DOI: 10.1109/ITSC.2015.164.

[28] ASAM e. V. *The individual parts of a road.* URL: https://releases.asam.net/OpenDRIVE/1.6.0/ASAM_OpenDRIVE_BS_V1-6-0.html (besucht am 06. 09. 2021).

[29] Walther Wachenfeld und Hermann Winner. „Die Freigabe des autonomen Fahrens“. In: *Autonomes Fahren.* Springer Vieweg, Berlin, Heidelberg, 2015, S. 439–464.

[30] „Quo vadis, FAS?“ In: *Handbuch Fahrerassistenzsysteme: Grundlagen, Komponenten und Systeme für aktive Sicherheit und Komfort.* Hrsg. von Hermann Winner, Stephan Hakuli, Felix Lotz und Christina Singer. Wiesbaden: Springer Fachmedien Wiesbaden, 2015, S. 1167–1186. ISBN: 978-3-658-05734-3. DOI: 10.1007/978-3-658-05734-3_62. URL: https://doi.org/10.1007/978-3-658-05734-3_62 (besucht am 25. 05. 2021).

[31] Z3. *Z3 Theorem Prover.* URL: https://github.com/Z3Prover (besucht am 10. 06. 2021).

A Appendix

```
<wm:objectType id="_IH6s8KUOEeu6zP_Cqt1lOg" name="Car">
    <wm:property id="_Q_2-AKUOEeu6zP_Cqt1lOg" name="x" type="real"
        dynamics="continuous" unit="m">
      <base:extensions>
        <tsc2openx:propertySemantics kind="X"/>
      </base:extensions>
    </wm:property>
    <wm:property id="_S9UJMKUOEeu6zP_Cqt1lOg" name="y" type="real"
        dynamics="continuous" unit="m">
      <base:extensions>
        <tsc2openx:propertySemantics kind="Y"/>
      </base:extensions>
    </wm:property>
    <wm:property id="_InJsMN_eEeubhvx43yuU3Q" name="v" type="real"
        dynamics="continuous" unit="m/s">
      <base:extensions>
        <tsc2openx:propertySemantics kind="VELOCITY"/>
      </base:extensions>
    </wm:property>
    <wm:property id="_iiB2gOZEEeuUdb8tXvb8ig" name="width" type="real"
        dynamics="continuous" unit="m">
      <base:extensions>
        <tsc2openx:propertySemantics kind="BB_HEIGHT"/>
      </base:extensions>
    </wm:property>
    <wm:property id="_lvpCAOZEEeuUdb8tXvb8ig" name="length" type="real"
        dynamics="continuous" unit="m">
      <base:extensions>
        <tsc2openx:propertySemantics kind="BB_HEIGHT"/>
      </base:extensions>
    </wm:property>
    <wm:derivedProperty id="_vAuL0uZEEeuUdb8tXvb8ig" name="min_x" type="real"
              definition="x - length / 2"/>
    <wm:derivedProperty id="_04oaYuZEEeuUdb8tXvb8ig" name="max_x" type="real"
              definition="x + length / 2"/>
    <wm:derivedProperty id="_2-IuUuZEEeuUdb8tXvb8ig" name="min_y" type="real"
              definition="y - width / 2"/>
    <wm:derivedProperty id="_5mw9cuZEEeuUdb8tXvb8ig" name="max_y" type="real"
              definition="y + width / 2"/>
    <base:constraint>v &lt;= 180[km/h]</base:constraint>
</wm:objectType>
```


Abbildung 32: Erweiterte Car Klasse.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wm:dictionary xmlns:wm="http://www.offis.de/tsc/2019-09/wm"
        id="_UUkoUKUUEeu6zP_Cqt1l0g">
  <wm:importWM>simple.worldmodel#_H19aQKUOEeu6zP_Cqt1l0g</wm:importWM>
  <wm:symbol id="_W7tCoKUUEeu6zP_Cqt1l0g" name="Car"
        type="simple.worldmodel#_IH6s8KUOEeu6zP_Cqt1l0g"
        visualization="../car.png">
    <wm:anchor id="_5hQnsAA1EeyPj-0qRUQetg" name="NORTH_WEST" x="min_x" y="max_y">
      <wm:location x="0" y="1"/>
    </wm:anchor>
    <wm:anchor id="__wBQAAA1EeyPj-0qRUQetg" name="NORTH_EAST" x="max_x" y="max_y">
      <wm:location x="1" y="1"/>
    </wm:anchor>
    <wm:anchor id="_DEbAAAA2EeyPj-0qRUQetg" name="SOUTH_WEST" x="min_x" y="min_y">
      <wm:location x="0" y="0"/>
    </wm:anchor>
    <wm:anchor id="_GbaJ8AA2EeyPj-0qRUQetg" name="SOUTH_EAST" x="max_x" y="min_y">
      <wm:location x="1" y="0"/>
    </wm:anchor>
  </wm:symbol>
</wm:dictionary>
```

Abbildung 33: Erweiterter Car-Eintrag der Symbolbibliothek mit Anker-Definitionen.

B Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Oldenburg, den 20. September 2021

Vincent Kalwa