technische universität dortmund

fakultät für informatik

Endbericht

eASy: Resilient Autonomous Systems

PG 664

9. März 2025

Betreuer:
Simon Dierl, M.Sc.
Daniel Busch, M.Sc.

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 14 für Software Engineering
Arbeitsgruppe AQUA
https://aqua.engineering/

AQUA automated quality assurance

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **5**
2.1 Formula Student . . . 5
2.2 Continuous Integration . . . 9
2.3 Git und GitLab . . . 10
2.3.1 Grundlagen von Git . . . 10
2.3.2 Grundlegende Funktionen von GitLab . . . 11
2.4 ROS . . . 12
2.5 JARVIC . . . 14

3 Konzept **17**

4 Vorgehensweise **22**
4.1 Voraussetzungen und Rahmenbedingungen . . . 22
4.2 Rollen und Kleingruppen . . . 23
4.3 Zeitliche Planung . . . 24
4.4 Zusammenfassung . . . 25

5 Developer Experience **27**
5.1 Simulatoren . . . 27
5.1.1 FSSIM . . . 28
5.1.2 PacSim . . . 29
5.1.3 CARLA . . . 29
5.1.4 FSDS . . . 30
5.1.5 Simulationsdeterminismus . . . 32
5.2 Integrierte Simulation . . . 32
5.2.1 RBB . . . 32
5.2.2 CI Integration . . . 33
5.3 Entwicklungsumgebung . . . 35
5.3.1 Containerbasierte Entwicklung . . . 37
5.3.2 Serverbasierte Entwicklung . . . 38

5.4 Weitere Unterstützung der Developer Experience 38
5.5 Migration auf ROS2 39
5.6 Fazit 39

6 Fehleranalyse 41
6.1 Fehleranalyse der PG 42
6.2 Resultat der Analyse 43

7 Fehlerinjektion 46
7.1 Grundlagen der Fehlerinjektion 46
7.1.1 Techniken der Fehlerinjektion 47
7.2 Fehlerinjektion auf Basis von ROS-Nachrichten 48
7.2.1 Implementierung von Fault-Nodes 50
7.2.2 Zentrale Steuerung durch Control-Node 52
7.2.3 Starten des Systems mit Fehlerinjektion 55
7.2.4 Implementierte Faults 57
7.2.5 Generische Fehlerinjektion 58
7.2.6 ROSMonitoring zur Fehlerinjektion 60
7.3 Fehlerinjektion in der Simulation durch Veränderung der Map 60
7.4 Fehlerinjektion in der Simulation durch Veränderung des Wetters 62
7.5 Integration der Fehlerinjektion in die CI-Pipeline 64

8 Domänenspezifische Sprache 66
8.1 Grundkonzept FLEX 66
8.2 Implementierung 69
8.3 Nutzer-Umfrage 73

9 Executor 78

10 Resilienzmaßnahmen 83
10.1 Fahren ohne SLAM 83
10.2 Unit Tests 87
10.3 Statische Codeanalyse 90
10.4 TopicParser 91
10.5 FastSLAM-Ausfall und TopicObserver 95
10.6 On-The-Fly-Kartenreperatur 96
10.7 Safety Envelopes 98

11 Diskussion 104
11.1 Zusammenfassung 104
11.2 Ausblick 106
11.3 Fazit 107

Abkürzungen 107

Literatur 108

A Lesernotizen 112

B FLEX 113

Kapitel 1

Einleitung

Das autonome Fahren ist ein aktuell viel diskutiertes Thema der Informatik und eines der großen Interessenfelder der Automobilindustrie. Die Gewährleistung von Fahrsicherheit selbstfahrender Autos ist ein Problem, das die Industrie noch nicht abschließend gelöst hat. Unfälle ereignen sich immer wieder und erwecken viel Aufmerksamkeit. Im Jahr 2016 gab es den ersten tödlichen Unfall mit einem autonomen Fahrzeug [Tad+20].

Das Ziel dieser Projektgruppe ist es, die Resilienz der Fahrsoftware selbstfahrender Fahrzeuge zu erhöhen. Es handelt sich dabei um die Fähigkeit eines Systems Risikosituationen ohne äußere Einflüsse zu bewältigen und auftretende Probleme zu minimieren. Dies wird erreicht, indem Fehler schnell erkannt und behandelt werden, sodass die Fahrsicherheit auch nach einem Fehlerzustand gewährleistet ist. Die betrachteten Fahrzeuge sind Rennwagen, welche durch den studentischen Rennverein „GET racing Dortmund e. V." der Technischen Universität Dortmund entwickelt und gebaut wurden. Die Fahrsoftware „JARVIC" wurde ebenfalls durch den Rennverein entwickelt und implementiert. GET racing nimmt an Wettbewerben der Formula Student teil und tritt in verschiedenen Disziplinen gegen Teams von anderen Universitäten an. Für den Fortschritt der Projektgruppe und die Entwicklung von autonomen Fahrsystemen sind die dynamischen Disziplinen relevant. Dabei handelt es sich um fahrbasierte Disziplinen. Seit einigen Jahren werden Rennen auch softwaregesteuert gefahren. Diese Variation wird Driverless genannt und ist eine praktische Anwendung von autonomen Resilienzaspekten.

Auch wenn diese Wettbewerbe in einer gesicherten Umgebung auftreten, kann es zu Unfällen kommen. Das Fahrzeug von GET Racing traf während den Disziplinen der letzten Formula Student mehrere Leitkegel. Teilweise haben sich diese innerhalb des Fahrzeugs verfangen oder wurden bei der Fahrt verschoben und umgeworfen. Ein Bild von diesem Ereignis ist in Abbildung 1.1 zu sehen. Mehrere Leitkegel befinden sich innerhalb des Fahrzeugs. Die fehlende Resilienz des Fahrzeuges ist ein Motivationsgrund für die Arbeit der Projektgruppe. Unfälle führen in der Praxis nicht nur zu hohen Sachschäden, sondern gefährden auch Menschen. Idealerweise kann das System aus jedem Fehlerzustand heraus wieder in einen risikominimalen Zustand wechseln. Ein Fehler bezeichnet in diesem Zusammenhang jegliches

Abbildung 1.1: Eine Fotoaufnahme des zuletzt verwendeten Fahrzeugs von GET racing, © FSG Lodholz.

Verhalten der Fahrsoftware oder der Hardware des Autos, welches vom gewollten Verhalten abweicht. Ein Fehlerzustand ist der Zustand, in dem sich die Software des Autos nach einem Fehler befindet. Dabei handelt es sich beispielsweise um einen Komponentenausfall oder eine Fehlberechnung. Als risikominimaler Zustand gilt zunächst jeder Zustand, indem kein Fahrzeugsystem, keine Objekte und keine Menschen zu Schaden kommen. Im Idealfall kann in diesem Zustand weitergefahren werden. Bei schweren Systemfehlern ist dies nicht immer möglich.

Dabei wird die Resilienz eines Fahrzeuges über die Fähigkeit, einen risikominimalen Zustand erreichen zu können definiert. Wenn ein Fehler auftritt, beispielsweise der Ausfall einer Komponente, muss das autonome System diesen zuverlässig registrieren können. Danach muss es eigenständig auf den Fehler reagieren und einen risikominimalen Zustand herstellen. Diese Fähigkeit des autonomen Systems wird im Folgenden als Fehlerbehandlung bezeichnet. Viele Fehlerfälle waren zu Beginn weder erkannt noch behandelt. Das Ziel der Projektgruppe ist, die Resilienz des Fahrzeuges zu erhöhen. Dafür liegt der Fokus auf dem Erkennen und Behandeln der Fehlerfälle, sowie weiteren Resilienzmaßnahmen, um Fehler zu verhindern.

Die Arbeit der Projektgruppe teilt sich in mehrere Bereiche auf: die Fehleranalyse, die Fehlerinjektion, die Erstellung einer Sprache zur Beschreibung von Fehlerszenarien, die automatisierte Simulation und die Fehlerbehandlung. Die Fehleranalyse fand statt, um einen Überblick über Fehlerquellen und Konsequenzen zu verschaffen. Dabei entstand eine grundlegende Struktur mit Fehlerszenarien, an denen gearbeitet wurde.

Weiterhin wurde die Fahrsoftware um Komponenten zur Fehlerinjektion erweitert, mit denen ein Fehlverhalten im System ausgelöst werden kann. Die Fehlerinjektion ist ein zentrales Werkzeug der Projektgruppe. Damit kann getestet werden, ob implementierte Resilienzmaßnahmen wie vorgesehen funktionieren. Sie bildet die Grundlage für die Arbeit an Resilienz.

Es existierte zu Beginn kein festgelegtes Vorgehen, um für die Fehlerinjektion gefundene Fehlerarten zu beschreiben. Zur Vereinfachung von Fehlerbeschreibung wurde eine eigene Szenariensprache entwickelt. Damit lassen sich insbesondere Kombinationen von Fehlerarten beschreiben, was eine Abdeckung von vielen Fehlertypen erlaubt. Die entwickelte Sprache ist eine nutzerfreundliche Methode, mit den entwickelten Fehlerinjektionsformen zu interagieren. Das Resultat ist ein Werkzeug für die Spezifikation von Fehlern und deren genauen Parametern für eine Injektion.

Für eine Visualisierung wurde die Integration und Automatisierung der Simulation des Fahrzeugverhaltens aufgesetzt. In der Simulation kann das Verhalten der Software während eines Rennens angenähert werden und erlaubt, die Qualität von Resilienzmaßnahmen ressourcensparend zu bewerten. Den besten Aufschluss über das Verhalten der Software liefern Testfahrten des physischen Rennautos, die jedoch sehr ressourcenaufwendig sind. Deshalb wurde die Simulation als Vorgehen zum Testen gewählt. Diese Entscheidung basiert auf dem Shift-Left Ansatz [Smi01], welcher besagt, dass möglichst früh im Entwicklungsprozess getestet werden muss. Es kann schneller, früher und automatisiert mithilfe der Simulation getestet werden. Dafür ist im Idealfall deterministisches Verhalten zu erwarten. Das bedeutet, dass bei einem identischen Fehler ein vergleichbares Ergebnis simuliert wird. Dadurch werden Fehler früher erkannt und können früher behandelt werden. Eine Automatisierung erlaubt die direkte Simulation und Analyse des Verhaltens von vorgegebenen Fehlerarten. Somit wird der Prozess für Nutzer:innen auf eine Eingabe eines Fehlers und den Ergebnissen der Simulation reduziert.

Zusätzlich wurden Methoden zur Fehlerbehandlung implementiert. Unter anderem wurden Tests geschrieben, um Programmierfehler (Bugs) zu erkennen. Durch diese Bugs treten Fehler im autonomen System auf, welche die Fahrsicherheit gefährden können und von dem System oft unbemerkt bleiben. Dementsprechend besteht ein Zusammenhang zwischen Bugs in der Software und der Resilienz. Mithilfe dieser Tests wurden bereits Bugs gefunden und behoben. Neben der Erstellung von Tests wurden Methoden entwickelt, um die analysierten Fehler zu behandeln. Mehrere Formen der Behandlung wurden von der Projektgruppe analysiert. Dabei wurden vielversprechende Behandlungsmethoden eingebaut. Das Resultat ist eine Zusammenstellung von verschiedenen implementierten Methoden. Gemeinsam stellen diese ein Proof-of-Concept von Fehlerbehandlung in mehreren Formen dar. Dabei entstanden eigens entwickelte Werkzeuge und zusätzlich wurden allgemein bekannte Maßnahmen implementiert.

Zusammenfassend wurden durch die Projektgruppe Resilienzmaßnahmen implementiert, um die Fahrsicherheit des Fahrzeuges zu gewährleisten und Unfälle zu vermeiden. Unter anderem handelt es sich dabei um Fehlerbehandlungen und Verhinderungen von Fehlerquellen. Dadurch wurde das Ziel, die Resilienz bei der Fahrsoftware zu erhöhen, erreicht. Dafür müssen verschiedene Bereiche betrachtet werden, welche im Folgenden vorgestellt werden. Es werden zuerst die wichtigsten Grundinformationen über die Bedingungen und Werkzeuge der Projektgruppe in Kapitel 2 beschrieben. Anschließend wird das konzeptuelle Vorgehen

der Projektgruppe in Kapitel 3 genauer aufgeschlüsselt und in Kapitel 4 die dafür entwickelte Vorgehensweise und Organisationsstruktur des Teams zusammengefasst. In Kapitel 5 wird auf die aufgebaute technische Infrastruktur zur Entwicklung eingegangen. Es folgen dabei insbesondere nähere Informationen zur Simulation. Informationen zum Prozess der Fehleranalyse befinden sich in Kapitel 6, zur Architektur der Fehlerinjektionskomponenten in Kapitel 7 und eine detaillierte Beschreibung der dafür entwickelten domänenspezifischen Sprache FLEX in Kapitel 8. Das Zusammenspiel von der domänenspezifischen Sprache, der Fehlerinjektion und der Simulation mithilfe des Executors wird in Kapitel 9 beschrieben. Kapitel 10 erläutert die Ansätze zur Fehlerbehandlung und der Implementierung von Resilienzmaßnahmen. In Kapitel 11 werden die bisherigen Resultate zusammengefasst und ein Ausblick auf die Nutzung der Ergebnisse der Projektgruppe durch GET Racing gegeben. Auch behandelt werden mögliche Folgearbeit durch künftige Projektgruppen oder Projekte des GET Racing-Teams.

Kapitel 2

Grundlagen

In diesem Kapitel werden die grundlegenden Informationen zur Ausgangssituation der Projektgruppe erklärt. Nach einer Übersicht über die Formula Student folgen theoretische Grundlagen der genutzten Entwicklerwerkzeuge. Genauere Informationen zu den gewählten Werkzeugen werden in Kapitel 5 gegeben. Es folgt eine Einführung in die Middleware ROS, welche von der betrachteten Fahrsoftware JARVIC verwendet wird. Eine allgemeine Betrachtung von JARVIC folgt zuletzt.

2.1 Formula Student

Die Formula Student ist ein internationaler Wettbewerb, bei dem Teams verschiedener Hochschulen Rennwagen konstruieren und in diversen Renndisziplinen gegeneinander antreten. Die Rennautos müssen von den Teams eigenständig, sowohl auf Hardware- als auch auf Software-Ebene, hergestellt werden. Außer Materialbereitstellung darf es keine Hilfe von außen geben [For25, Kap. A 2]. Der Fokus wird hier auf die Rennregeln des autonomen Fahrens gelegt. Als internationaler Wettbewerb ist die offizielle Sprache der Formula Student Englisch. Daher verbleiben Namen oder Titel auf Englisch, um Unklarheiten zu vermeiden.

Die Teams werden bei der Formula Student in Events bewertet, die in zwei Kategorien unterteilt sind: statische und dynamische Events. Dabei drehen sich statische Events um die Engineering- und die Business-Seite des Fahrzeug-Entwicklungsprozesses. Diese Events werden aufgrund mangelnder Relevanz für die Arbeit an resilienter Software nicht genauer betrachtet. Die dynamischen Events sind verschiedene Disziplinen, bei denen die Performance der Fahrzeuge bewertet wird. Bei den dynamischen Events wird zwischen dem Eletric Vehicle Cup mit Fahrern und dem Driverless Cup für das autonome Fahren unterschieden. Im Rahmen der PG wurde sich mit den Disziplinen des Driverless Cups befasst. Im Driverless Cup gibt es außer einem Not-Aus keine Möglichkeit für das Team während einer Fahrt einzugreifen, sodass das Fahrzeug selbsttätig die Disziplin absolvieren muss.

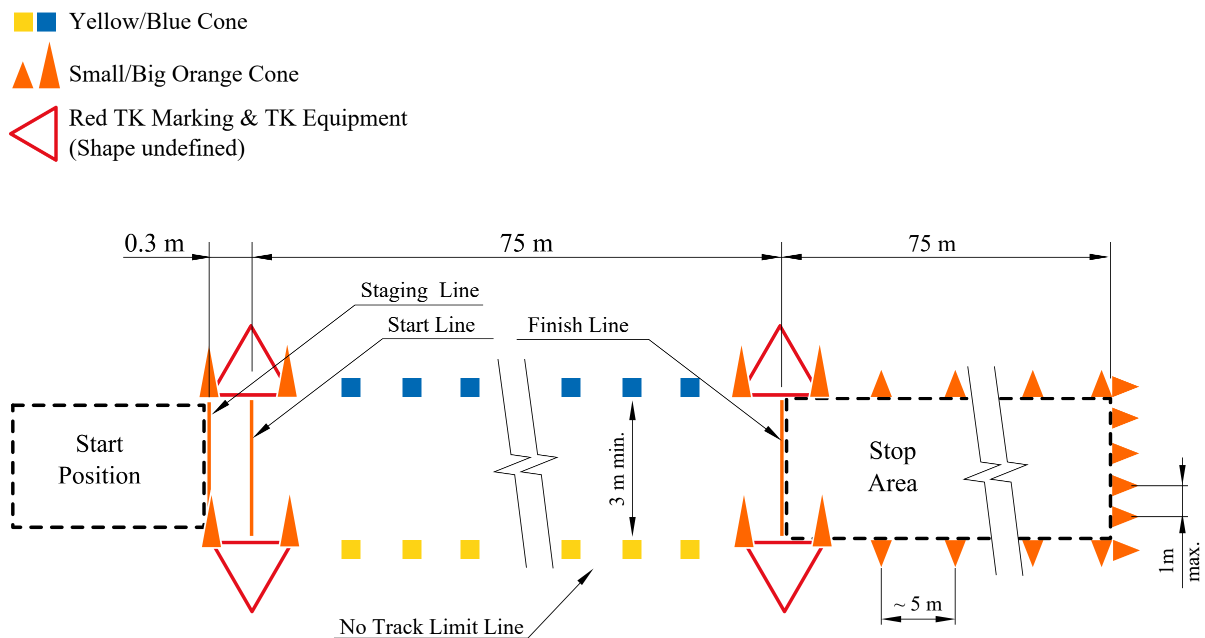


Abbildung 2.1: Der Aufbau eines Acceleration-Events nach Competition Handbook [For24]. TK: Time-Keeping-Equipment zur Zeitmessung.

Grundsätzlich sind im Driverless Cup die Ränder der Strecke durch Leitkegel markiert: der linke Rand blau, der rechte Rand gelb. Sofern nicht anders festgelegt, dürfen maximal 5 m zwischen zwei Leitkegeln gleicher Farbe liegen [For24, DE 7.5.1]. Sollte ein Fahrzeug während der Disziplin einen Leitkegel berühren, führt dies zu einer Zeitstrafe. Orangene Leitkegel dienen als Marker für Start-/Zielbereiche, oder in spezifischen Disziplinen für statische Referenzpunkte wie Zeitmessungsgeräte (Time-Keeping-Equipment; abgekürzt TK). Die Zeitmessung wird für die Bewertung der Performance in den Disziplinen verwendet. Für jede Disziplin ist eine Maximalpunktzahl definiert, welche in Tabelle 2.1 für die dynamischen Events aufgeführt ist. Je nach erreichter Zeit in der Disziplin werden die Punkte entsprechend vergeben.

Die dynamischen Events des Driverless Cups beinhalten die Disziplinen Skidpad, Acceleration, Autocross und Trackdrive und werden im folgenden kurz vorgestellt:

Acceleration Hierbei wird das Beschleunigungsverhalten des Rennwagens gemessen. Der Aufbau des Events ist in Abbildung 2.1 dargestellt. Zunächst wird auf einer geraden Strecke möglichst stark beschleunigt. Danach muss der Rennwagen innerhalb eines definierten 75 m langen Zielbereichs anhalten. Wird der Zielbereich nicht erreicht oder nach Eintritt verlassen, wird der Versuch als „Did Not Finish“ gewertet. Bei diesem Event wird die Begrenzung der Strecke ausschließlich durch Leitkegel markiert. [For25, Kap. D 5]

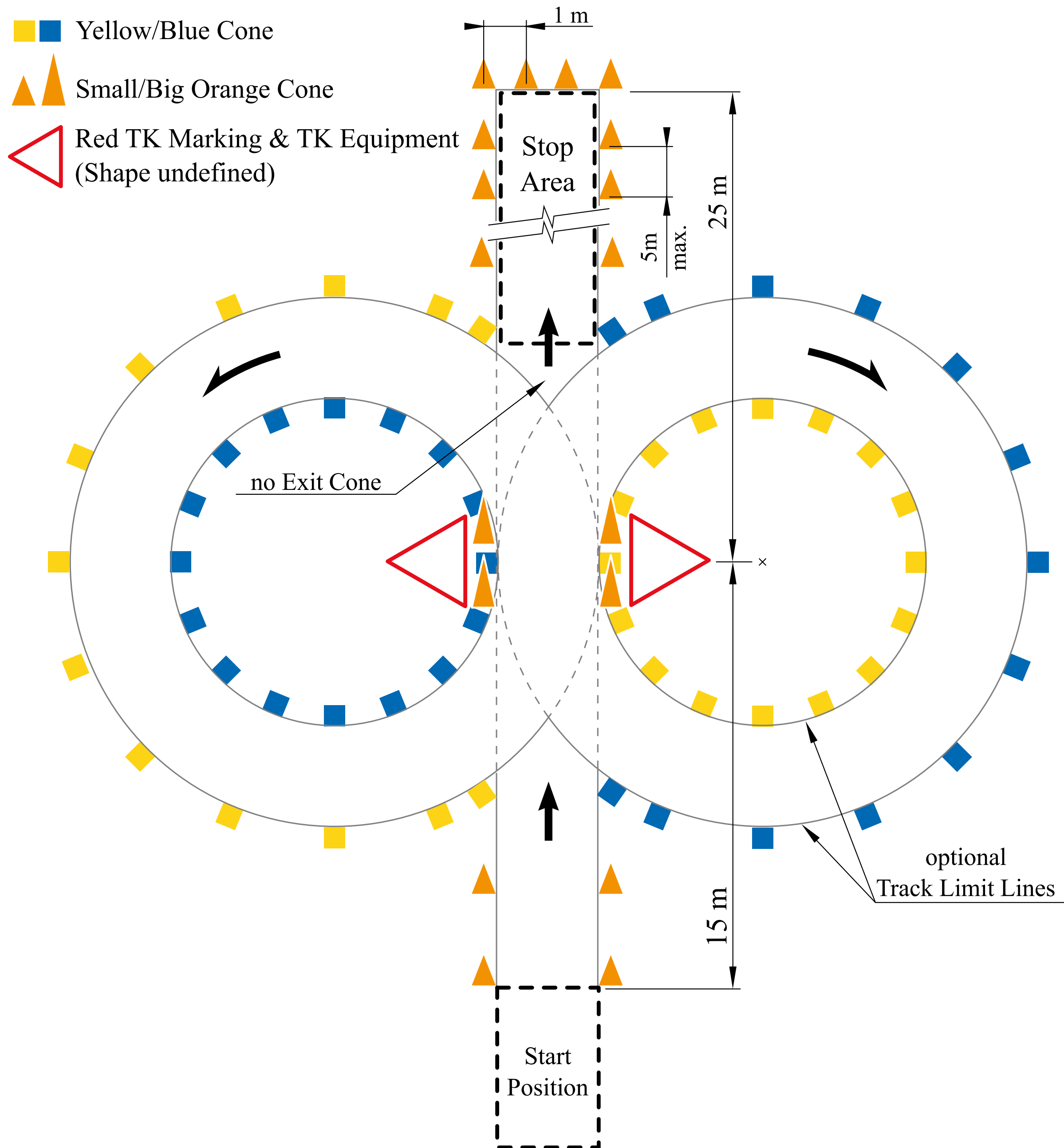


Abbildung 2.2: Der Aufbau eines Skidpad-Events nach Competition Handbook [For24]. Hier auch gezeigt sind die optionalen Randmarkierungen. TK: Time-Keeping-Equipment zur Zeitmessung.

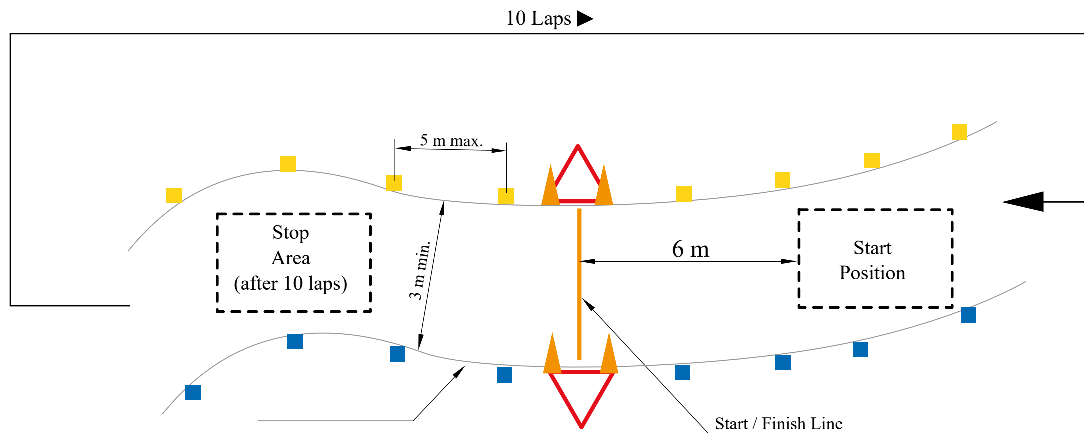


Abbildung 2.3: Der Aufbau eines Trackdrive-Events nach Competition Handbook [For24]. Hier auch gezeigt sind die optionalen Randmarkierungen. TK: Time-Keeping-Equipment zur Zeitmessung.

Skidpad In diesem Event soll die Lenkfähigkeit des Autos getestet werden. Der Rennwagen durchfährt dazu zwei sich überlappende Kreise, wie in Abbildung 2.2 abgebildet. Die Kreise müssen in Rechts-Rechts-Links-Links-Reihenfolge durchfahren werden. Gemessen wird jeweils die zweite Umkreisung pro Richtung. Die Start- und Zielbereiche befinden sich in der Mitte zwischen den beiden Kreisen und sind durch kleine, orangene Leitkegel markiert. Die Strecke kann optional durch eine gemalte Randmarkierung gekennzeichnet sein. Sobald begonnen wurde, darf es keine Pause geben, bis alle vier Runden gefahren wurden. Das Überschreiten der Streckengrenzen und Umstoßen von Leitkegeln führt zu Zeitstrafen. [For25, Kap. D 4]

Autocross und Trackdrive Autocross und Trackdrive laufen ähnlich ab. Sie fungieren als umfassender Test der Fahrzeuge, insbesondere hinsichtlich der Navigationsfähigkeit des Rennwagens. Beide Disziplinen verwenden denselben Streckenaufbau [For25, Kap. D 6.1.3]. Die Rennstrecke besteht aus geraden Abschnitten, (scharfen) Kurven sowie weiteren besonderen Hindernissen wie Schikanen oder eng aufeinanderfolgenden Kurven. Gesondert markiert ist nur die Startlinie, sichtbar in Abbildung 2.3.

Die beiden Disziplinen unterscheiden sich hauptsächlich durch den Ablauf: Bei Autocross wird die Strecke einmal befahren, während bei Trackdrive 10 Runden zum erfolgreichen Abschluss notwendig sind. Weiterhin ist bei Autocross ein Trackwalk erlaubt, bei dem die Fahrstrecke ohne technische Messgeräte vor der Disziplin abgelaufen werden kann. Dies erlaubt eine genauere Einschätzung der Strecke vor Beginn

Tabelle 2.1: Punktetabelle der dynamischen Events des FSG (nach [For25, Kap. A.1]).

Dynamische Events:	Electric Vehicle	Driverless Cup
Skidpad	50 Punkte	-
Driverless (DV) Skidpad	75 Punkte	75 Punkte
Acceleration	50 Punkte	-
Driverless (DV) Acceleration	75 Punkte	75 Punkte
Autocross	100 Punkte	-
Driverless (DV) Autocross	-	100 Punkte
Endurance	250 Punkte	-
Efficiency	75 Punkte	-
Trackdrive	-	200 Punkte
Overall	675 Punkte	450 Punkte

der Disziplin und somit eine detailliertere Vorbereitung der autonomen Software. Autocross und Trackdrive agieren als Hauptevents der FSG, da beide mehrere Aspekte des autonomen Systems auf einmal testen.

2.2 Continuous Integration

Ein manuelles Testen der autonomen Fahrsoftware ist aufwendig und fehleranfällig. Daher sollten diese Tests möglichst automatisch nach festen Testplänen ausgeführt werden. Eine mögliche Technik dafür ist Continuous Integration (CI), bei welcher der Code einer Software automatisiert regelmäßig gebaut und getestet wird, wodurch Fehler einfacher und früher im Entwicklungsprozess gefunden werden können [FS17]. Weitergehend kann mittels Continuous Delivery (CD) der Code automatisiert in einen Release ausgeliefert oder zumindest automatisiert für einen Release vorbereitet werden [Git24a]. Bei CD ist die Idee sicherzustellen, dass zu jeden Zeitpunkt voll funktionsfähige Software vorliegt, welche im Idealfall automatisch veröffentlicht wird, sodass z. B. schneller Feedback eingeholt werden kann [SAZ17]. Da Releases im Rahmen der autonomen Fahrsoftware jedoch nicht nötig sind, wird CD hier nicht weiter betrachtet.

Der Prozess von CI innerhalb eines Entwicklungszyklus wird häufig als eine Art Pipeline aufgefasst. Ein Beispiel eines solchen Entwicklungszyklus mit automatisierter CI-Pipeline ist in Abbildung 2.4 dargestellt. Der Prozess beginnt mit der Entwicklung der Software bzw. eines neuen Features. Nach Abschluss der Änderungen kann die Pipeline gestartet werden, um Rückmeldung zu den Anpassungen zu erhalten. Die Ausführung kann zum Beispiel automatisiert durch das Hinzufügen von neuem Code zu einem Repository angestoßen werden. In der Beispielpipeline wird als Eingabe der Code zuerst gebaut. Wenn dieser Abschnitt

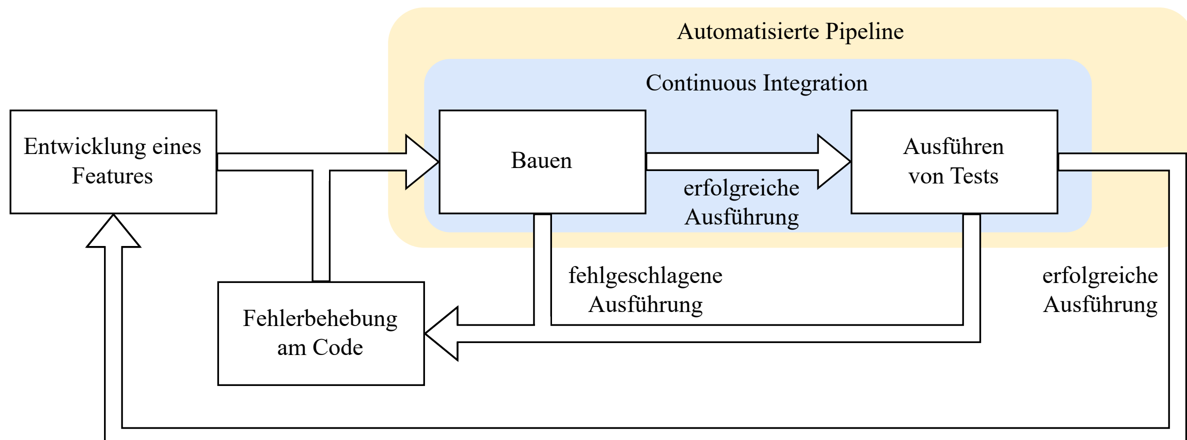


Abbildung 2.4: Beispiel eines Entwicklungsprozesses unter Verwendung einer automatisierten Pipeline.

fehlschlägt, sollte der Code so angepasst werden, bis der Abschnitt erfolgreich ausgeführt wird. Bei erfolgreichem Bauen werden Tests ausgeführt. Wenn die Tests fehlschlagen, muss der Code korrigiert werden. Andernfalls wurde die Pipeline erfolgreich ausgeführt und der Entwicklungsprozess kann in einer neuen Iteration von vorne beginnen. Daher wird dies häufig auch als ein iterativer Prozess aufgefasst [Git24b]. Die genauen Stufen, welche innerhalb der Pipeline ausgeführt werden, sind flexibel. Neben einfachem Bauen und Testen der Software können auch z. B. externe Tools aufgerufen werden, wie etwa zur statischen Codeanalyse.

Damit das automatisierte Bauen von Software vernünftig eingesetzt werden kann, müssen sinnvolle Tests oder Bedingungen existieren, welche Auskunft über den Zustand der Software geben. Wenn beispielsweise keine oder nicht ausreichend umfassende Unit Tests existieren, kann durch die CI-Pipeline nicht sicher bemerkt werden, wenn Teile der Software fehlerhaft sind. Allerdings muss auch trotz möglichst umfassender Tests beachtet werden, dass die Laufzeit der Pipeline gering gehalten wird, damit ein schnelles Feedback möglich ist [Fow24].

2.3 Git und GitLab

In diesem Abschnitt wird das von der Projektgruppe genutzte Versionskontrollsystem Git sowie für die Projektgruppe relevante Funktionen des Entwicklungssystem GitLab vorgestellt.

2.3.1 Grundlagen von Git

Git ist ein Versionskontrollsystem. Die Idee hinter Versionskontrollsystemen ist, dass der Änderungsverlauf von Dateien innerhalb eines Repositories gespeichert wird. Dadurch lassen

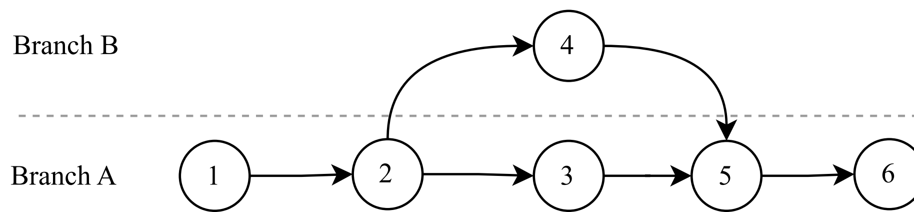


Abbildung 2.5: Git-Versionshistorie zweier Branches. Kreise stellen Commits dar. Nummern dienen zur Identifizierung der Commits.

sich Änderungen einfach nachvollziehen und bei Bedarf zurücksetzen. Die Versionshistorie eines Repositories besteht bei Git aus einer Verkettung von sogenannten Commits. Ein Commit ist dabei durch einen Hash gekennzeichnet und enthält unter anderem eine Menge an Änderungen (neue Dateien, Änderungen an Dateien usw.). Durch die Verkettung der Commits ergibt sich der aktuelle Stand des Repositories.

Sogenannte Branches sind benannte Verzweigungen in der Historie eines Repositories, die auf bestimmte Commits verweisen. Diese ermöglichen eine klare Strukturierung und erleichtern die Identifikation verschiedener Entwicklungszweige. Eine Versionshistorie mit mehreren Commits ist als Beispiel in Abbildung 2.5 dargestellt. Dort ist ebenfalls abgebildet, wie von einem Branch abgezweigt werden kann, indem der neue Branch B ausgehend von Commit 2 erstellt wird. Um die Änderungen eines Branches auf einen anderen Branch zurückzuführen, kann ein Merge oder Rebase verwendet werden. Dies ist ebenfalls in Abbildung 2.5 beispielhaft dargestellt, wobei durch Commit 5 Branch A und Branch B wieder zusammengeführt werden.

Git ermöglicht das parallele Arbeiten an einem Projekt mit mehreren Entwicklern. Dazu können die lokalen Stände der Repositories der Entwickler:innen mit Remote-Repositories synchronisiert werden. Dabei wird die Gefahr vermieden, dass Änderungen an derselben Datei versehentlich überschrieben werden.

2.3.2 Grundlegende Funktionen von GitLab

GitLab ist eine Serversoftware und bietet als wesentlichen Bestandteil den Dienst des zentralen Git-Repositories zur Synchronisation an. Andere Entwicklungswerkzeuge wie ein Issue-Management sowie das Bereitstellen von Projektdokumentation werden ebenfalls von GitLab bereitgestellt. GitLab ermöglicht zudem das Erstellen einer CI/CD-Pipeline (vgl. Abschnitt 2.2). Im Folgenden werden einige der Funktionen von GitLab kurz beschrieben.

Merge Requests und Code Reviews Mit einem Merge Request können Nutzer:innen eine Anfrage stellen, die Änderungen von einem Branch in einen anderen Branch zu mergen. Für Merge Requests können Code Reviews durchgeführt werden. Durch diese können andere Entwickler:innen (sogenannte Reviewer) Anmerkungen zu den Änderungen geben. Code Reviews

werden bei vielen größeren Projekten eingesetzt und dienen dazu, den Quellcode einheitlich zu halten und einen Wissenstransfer zwischen Entwickler:innen zu ermöglichen [Sad+18]. Ein Reviewer kann nach dem Review angeben, ob der Merge Request akzeptiert wird. Um die Codequalität eines Projekts sicherzustellen, kann forciert werden, dass für das Mergen von Merge Requests die Zustimmung von Reviewern notwendig ist. Für das Mergen eines Merge Request kann z. B. auch ein erfolgreiches Durchlaufen der CI/CD Pipeline (vgl. Abschnitt 2.2) erfordert werden.

Issue-Management Für Projekte, an denen mehrere Personen gleichzeitig arbeiten, ist die Übersicht und Verwaltung von Aufgaben essenziell. Dazu können in GitLab Issues verwendet werden, die Probleme oder Aufgabenstellungen repräsentieren. Diese können für ein Projekt angelegt werden und können eine Beschreibung, verantwortliche Personen, Kategorien und Abhängigkeiten enthalten. Zudem können einem Issue sogenannte Tasks hinzugefügt werden, welche die einzelnen Aufgaben des Issues beschreiben. Wenn ein offenes Issue erfolgreich bearbeitet wurde oder das Issue nicht mehr relevant ist, kann der Zustand des Issues von *open* in *closed* geändert werden.

GitLab bietet weiterhin die Funktion, Issue-Boards zu erstellen, in denen Issues gruppiert angezeigt werden können. Ein Issue-Board kann als Kanban-Board für die Kanban-Methode genutzt werden [Git24d]. Ein weiteres Feature von GitLab ist das Verwalten von Milestones. Diese fassen Issues zusammen, welche für den nächsten Release abgearbeitet sein sollen. Für einen Überblick über Anforderungen an die Software können Requirements verwendet werden. Diese beinhalten eine Beschreibung der Anforderungen und können, wenn möglich, in der CI-Pipeline automatisiert auf Erfüllung getestet werden [Git24c; Git24e].

2.4 ROS

ROS [Qui+09] ist ein Open-Source-Framework, welches für die Entwicklung von Software in der Robotik eingesetzt wird. ROS existiert in zwei Versionen: ROS1 und ROS2. ROS1 wurde im ersten Dreivierteljahr der Projektgruppe verwendet, wird jedoch ab Mai 2025 nicht weiter unterstützt. Im weiteren Verlauf der Projektgruppe wurde daher ROS2 verwendet. Folgend werden relevante Informationen von ROS im Allgemeinen beschrieben. Bei Abweichungen zwischen ROS1 und ROS2 wird explizit darauf hingewiesen. Die wichtigste Ausführungseinheit von ROS sind Nodes, welche als eigenständige Anwendung ausgeführt werden und üblicherweise jeweils eine spezielle Funktion erfüllen. Dieser modulare Aufbau erlaubt eine separate Entwicklung von Funktionalität in kleineren Komponenten. Nodes können unter anderem mittels C++ oder Python geschrieben werden.

Damit Nodes miteinander interagieren können, verwendet ROS eine Architektur, in der die Nodes über sogenannte Topics miteinander kommunizieren können. Ein Topic wird durch

einen Namen identifiziert. Nodes können auf Topics Nachrichten veröffentlichen (Publisher) oder empfangen, indem sie das Topic abonnieren (Subscriber). Bei ROS1 existiert dazu eine zentrale Komponente, Roscore, welche die Subscriber und Publisher der einzelnen Topics verwaltet und den an der Kommunikation beteiligten Nodes angibt, welchen Nodes sie die jeweiligen Nachrichten schicken sollen. Für ROS2 wird hingegen dazu eine Implementierung des Data Distribution Service (DDS) Standards verwendet, welche auch ohne zentrale Komponente die Vermittlung der Nodes unterstützt und damit einen Single Point of Failure vermeidet [Woo14]. Die Kommunikation selbst geschieht in beiden Fällen direkt zwischen den Nodes. So wird durch diese Peer-to-Peer-Architektur eine m:n-Kommunikation implementiert. Eine solche Kommunikation über ein ROS-Topic ist beispielhaft mit $m = 3$ Publishern und $n = 2$ Subscribern in Abbildung 2.6 dargestellt. Ein Vorteil der topic-basierten Kommunikation ist nach [TV13, S. 35], dass die Nodes voneinander entkoppelt sind. So können einzelne Nodes bearbeitet werden, ohne dass sich die Änderungen auf andere Nodes auswirken, auch wenn diese über Topics miteinander interagieren. Dies führt dazu, dass Nodes entfernt und neue Nodes hinzugefügt werden können, ohne dass der Code anderer Nodes verändert werden muss.

Eine Node im ROS-System kann durch den `rosrun`-Befehl (bzw. `ros2 run` Befehl mit ROS2) gestartet werden. Da jedoch in einem größeren System meist mehrere Nodes gestartet werden müssen, bietet ROS mittels Launchdateien eine XML- oder (in ROS2) Python-basierte Konfigurationsdatei an, mit welcher Nodes zusammengefasst gestartet werden können. Launchdateien bieten zudem noch weitere Funktionalität wie bspw. das konditionelle Starten von Nodes in Abhängigkeit von Umgebungsvariablen oder das Setzen von Parametern für Nodes. Launchdateien können ebenfalls in andere Launchdateien eingebunden werden, sodass diese in kleinere einzelne Dateien gegliedert werden können.

Neben dem Framework der Nodes und der Topic-basierten Kommunikation bietet ROS weitere, auf dem ROS-Ökosystem aufbauende Tools an. Dazu zählt unter anderem RQt, ein Tool zum Visualisieren, welches unter anderem die verteilt ausgeführten ROS-Nodes darstellen kann, sowie RViz, womit in einer 3D-Umgebung ROS-Nachrichten und Sensordaten gekoppelt an die Position des Roboters visualisiert werden können.

Eine weitere wichtige Funktionalität sind Rosbags. Der Nachrichtenverlauf aller Nodes kann in einer Datei, einem sogenannten Rosbag, aufgezeichnet werden. Dies erlaubt die spätere Wiederholung und Analyse des Nachrichtenverlaufs. Die Suche nach Systemfehlern ist ohne Rosbags schwierig, da ROS-Nodes nicht deterministisch sind und die Verwendung eines Debuggers aufgrund der vielen laufenden Prozesse bzw. Nodes nicht praktikabel ist. Die nichtdeterministische Natur führt auch dazu, dass Fehler durch erneutes Ausführen nicht immer reproduziert werden können. Durch Rosbags lassen sich fehlerhafte Durchläufe erneut abspielen und analysieren. Auch für die Evaluierung einzelner Simulationsdurchläufe sind Rosbags hilfreich.

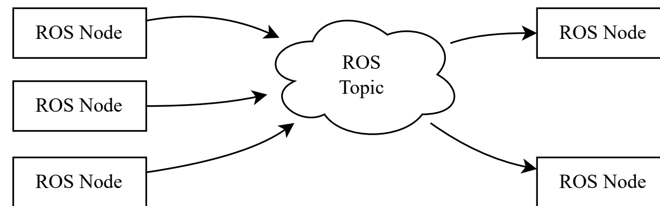


Abbildung 2.6: Abstrahierte $m : n$ topicbasierte Kommunikation über ein ROS-Topic mit $m = 3$ Publishern und $n = 2$ Subscribern.

2.5 JARVIC

JARVIC ist die Fahrsoftware für das autonome Fahren der Rennwagen von GET racing und wurde bereits auf Wettbewerben wie der Formula Student Germany (FSG) 2024 oder der Formula Student East 2024 eingesetzt. JARVIC verwendet ROS und findet sich aktuell in der Migration von ROS 1 Noetic zu ROS 2 Humble. Mehr Informationen zu dieser Migration folgen in Abschnitt 5.5. Die Nodes von JARVIC sind entsprechend ihrer Funktionalität in die folgenden Komponenten gruppiert:

Perception Besteht aus Nodes, welche die Positionen der Leitkegel bestimmen, indem sie Daten von den Kameras und dem Lidar verarbeiten. Der Lidar ist ein Sensor, der mithilfe von Laserstrahlen die Umgebung des Rennwagens abtastet und die Entfernung zu Objekten in der Nähe des Rennwagens bestimmen kann. GET verwendet seit der Saison 2024 kein Lidar mehr, sodass aktuell nur Kamerabilder verarbeitet werden. In der Saison 2024 wurde ein Sterokamera-Setup mit zwei Kameras verwendet. Ab Saison 2025 wird voraussichtlich ein Monokamera-Setup verwendet.

Estimation Umfasst Nodes, welche durch Auswertung von Sensoren wie zum Beispiel der IMU (Inertial Measurement Unit; ein Sensor, der die Richtung bestimmen kann, in die sich das Auto aktuell bewegt) die aktuelle Position des Autos schätzen und diese in einer dynamisch erstellten Map nachhalten. Zentraler Bestandteil dieser Komponente ist aktuell eine Implementierung eines SLAM-Algorithmus [Mon+02].

Planning Anhand der erstellten Map und der Position des Autos erstellen Nodes dieser Komponente einen Plan, wie das Auto fahren soll. Dazu gehören zum Beispiel Nodes zur Erkennung der Streckenbegrenzung, zum Zählen der bereits absolvierten Runden und der Erstellung der Trajektorie.

Control Nodes dieser Komponente übersetzen die in der vorherigen Planning-Komponente generierte Trajektorie in Steuerbefehle (Lenk-, Beschleunigungs- und Verzögerungsbefehle).

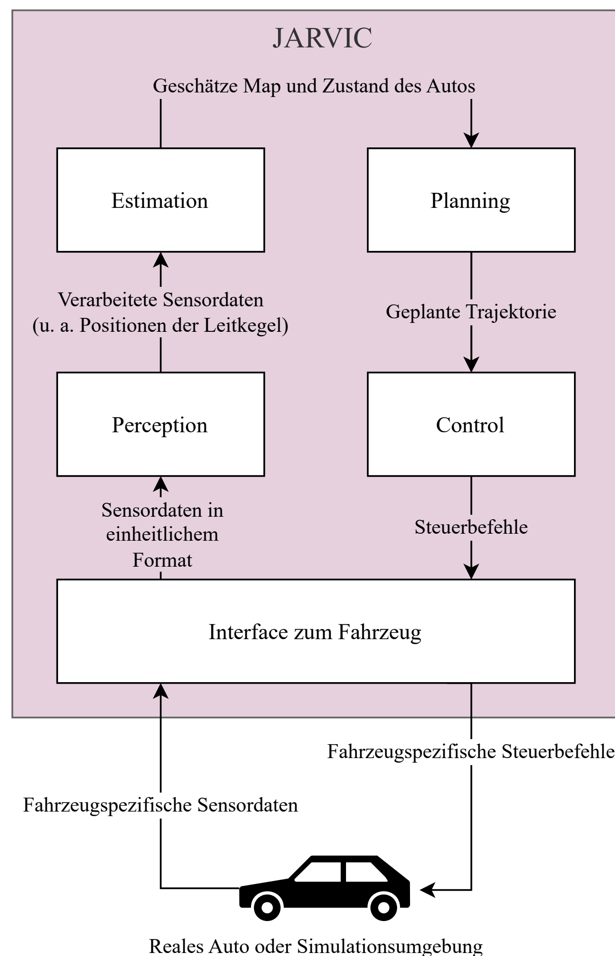


Abbildung 2.7: Vereinfacht dargestelltes Zusammenspiel der verschiedenen Komponenten von JARVIC (ausgenommen *Common*).

Interface zum Fahrzeug Diese Komponente enthält Adapter zu verschiedenen Umgebungen (Komponenten des physischen Fahrzeugs oder Simulatoren), um Steuerbefehle umzusetzen und Sensordaten vom Fahrzeug zu erhalten.

Common Verschiedene Nodes, die zusätzliche Dienste zur Verfügung stellen, bspw. zum Monitoring. Hierzu zählt u. a. die TopicObserver Node, welche feststellt, wenn bestimmte Nodes ausfallen und einen Notstopp einleitet.

Das Zusammenspiel der verschiedenen Komponenten von JARVIC (ausgenommen Common) ist in Abbildung 2.7 vereinfacht als Kreislauf dargestellt. Teilweise existieren in den einzelnen Komponenten mehrere austauschbare Nodes, welche dieselbe Funktionalität durch verschiedene Algorithmen implementieren. Beispielsweise gibt es verschiedene Nodes für die Erkennung der Streckenbegrenzung. Auf eine genauere Beschreibung der einzelnen Nodes

und deren Zusammenspiel innerhalb der Komponenten wird an dieser Stelle verzichtet. Dies wird an entsprechenden Stelle, sofern nötig, separat beschrieben.

Zum Starten von JARVIC existieren für verschiedene Missionen (Skidpad, Acceleration usw.) je eine Launchdatei. Jede Mission ist für die gleichnamige Disziplin der dynamischen Events der Wettbewerbe gedacht. Die Auswahl einer Mission und weitere Konfigurationsoptionen mittels Umgebungsvariablen entscheiden, welche Komponenten bzw. Nodes des JARVIC-Stacks gestartet werden.

Kapitel 3

Konzept

Ziel dieses Kapitels ist es, einen Überblick über die im Projekt entwickelten Komponenten zur Erhöhung der Resilienz von JARVIC zu geben. Es zeigt, wie die Arbeiten der Projektgruppe zusammenhängen und weshalb die jeweiligen Entwicklungen notwendig und sinnvoll sind. Zur Veranschaulichung wurden zwei Abbildungen erstellt. Abbildung 3.1 stellt den Aufbau der Software dar. Sie gibt einen groben Überblick darüber, wie die verschiedenen Komponenten miteinander interagieren und an welchen Stellen spezifische Verbindungen bestehen. In Abbildung 3.2 wird der Gesamtprozess dargestellt. Auf der linken Seite befindet sich der Pfad der Unittests, während sich der rechte Teil in die Bereiche Fehleranalyse, Simulation, Fehlerinjektion sowie Fehlererkennung und -behandlung aufteilt. Mithilfe der Simulation wird überprüft, ob die Fehlerbehandlung erfolgreich war. Fehler werden gezielt injiziert, und das Verhalten des Fahrzeugs wird anschließend analysiert.

Die Fehleranalyse bildete dabei die Grundlage für die weitere Arbeit. Bevor Fehler injiziert werden konnten, mussten diese zunächst identifiziert und definiert werden. Dazu wurde, wie Kapitel 6 beschrieben, eine strukturierte Analyse in enger Abstimmung mit Expertenteams durchgeführt. Sie war notwendig, um potenzielle Fehler überhaupt erst zu erfassen und eine Einschätzung ihrer Relevanz zu ermöglichen. Zu Beginn der Projektarbeit wurde daher eine umfassende Fehlerliste erstellt, die als Orientierung diente und half, den Arbeitsumfang zu strukturieren. Dabei wurde auch berücksichtigt, welche Fehler besonders kritisch sind oder in der Vergangenheit bereits aufgetreten sind. Auf Basis dieser Analyse wurden die relevanten Fehler für die Injektion ausgewählt und die weiteren Projektschritte geplant.

Aus der Fehleranalyse gingen verschiedene Fehlerszenarien hervor, die entweder einzelne Fehler, mehrere gleichzeitig oder verkettete Fehler enthalten können. Diese Szenarien werden mithilfe einer domänenspezifischen Sprache (DSL) einheitlich beschrieben, die von der Projektgruppe eigens entwickelt wurde. Die Sprache, genannt Flex und ermöglicht es, Fehlerszenarien in einer strukturierten und standardisierten Form darzustellen. Dadurch können sie maschinell besser verarbeitet und systematisch ausgewertet werden. Die einheitliche Beschreibung führt zur Generierung einer Output-Datei, die die definierten Fehler an die jeweils zuständigen Komponenten übergibt. Details zur Sprache Flex sowie Beispiele für

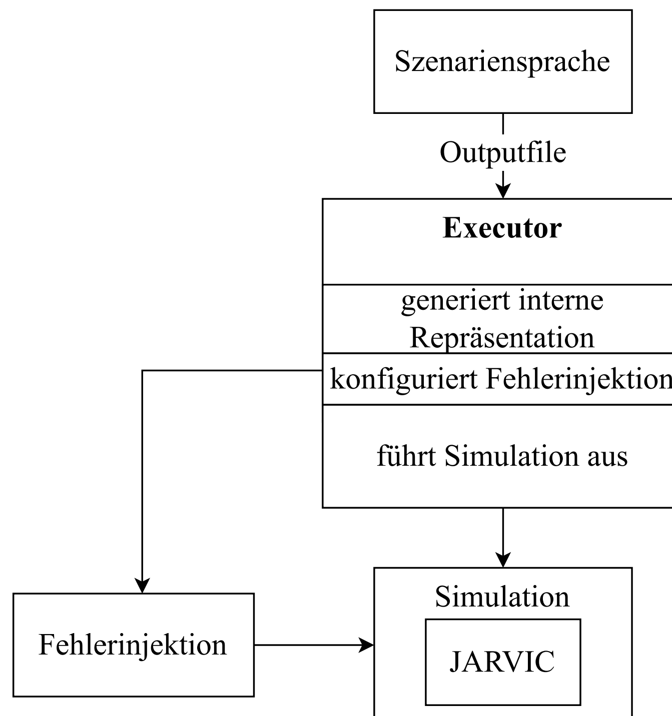


Abbildung 3.1: Der Aufbau der Softwarearchitektur der Projektgruppe.

ihre Anwendung werden in Kapitel 8 erläutert. Für die Fehlerinjektion wurde ein eigens entwickeltes Framework eingesetzt. Der Ablauf der Injektion ist in Abbildung 3.1 dargestellt. Dabei stellte sich die Frage, wie die in der Fehlerliste identifizierten Fehler sinnvoll in die Simulation überführt werden können. Je nach Art des Fehlers erfolgt die Injektion entweder über den Map Modifier oder direkt über JARVIC. Der Map Modifier verändert die Karte, um bestimmte Fehlerfälle abzubilden. So können etwa Leitkegel verschoben werden, um falsch erkannte oder in einer vorherigen Runde durch das Fahrzeug verschobene Leitkegel zu simulieren. Alternativ werden durch die gezielte Manipulation von Nachrichten über JARVIC Komponentenausfälle nachgestellt. Detaillierte Informationen zu den einzelnen Komponenten der Fehlerinjektion finden sich in Kapitel 7.

Die zuvor beschriebenen Werkzeuge dienen dazu, den sogenannten Executor zu realisieren. Dieser verarbeitet die in der DSL definierten Fehlerszenarien, indem er die entsprechenden Fehlerinjektions-Komponenten aktiviert und eine Simulation startet. Als zentrale Komponente koordiniert der Executor den gesamten Ablauf der Fehlersimulation. Er wurde so gestaltet, dass er besonders benutzerfreundlich ist und einen einfachen Einstieg in die Nutzung der entwickelten Komponenten ermöglicht. Dadurch wird auch die Weiterverwendbarkeit der Arbeitsergebnisse durch Get Racing wesentlich erleichtert. Der Executor ist modular aufgebaut und lässt sich problemlos um neue Szenarien, zusätzliche Fehlertypen oder weitere Simulatoren erweitern. Das macht ihn vielseitig einsetzbar und zukunftssicher. Die Architek-

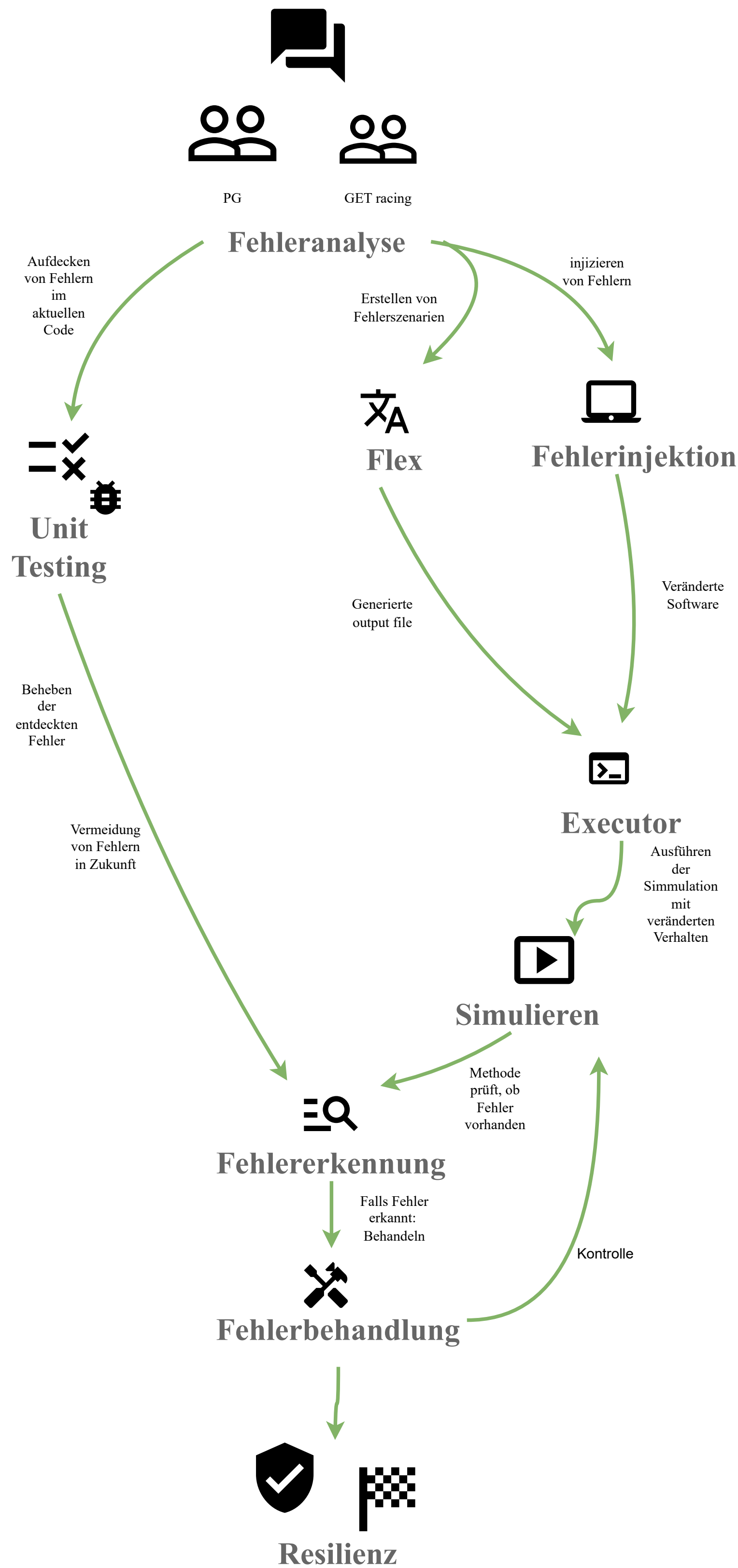


Abbildung 3.2: Eine Visualisierung des Arbeitsablaufs der Projektgruppe. Hier bilden grüne Pfade bereits umgesetzte Teilarbeit, gestrichelte noch unvollständige oder nicht vorhandene.[1]

tur des Executors wird in Kapitel 9 beschrieben, weitere Details zur Simulation finden sich in Abschnitt 5.1.

Die Simulation ermöglicht es, das Verhalten des Fahrzeugs unter dem Einfluss injizierter Fehler gezielt zu beobachten. So kann analysiert werden, welche konkreten Auswirkungen die Fehler auf das Fahrverhalten haben und ob daraus Handlungsbedarf entsteht.

Auf dieser Grundlage lassen sich Methoden entwickeln, mit denen Fehler automatisch erkannt werden können – sowohl in Bezug auf den Zeitpunkt ihres Auftretens als auch auf ihre genaue Art. Aufbauend auf dieser Fehlererkennung mussten anschließend Verfahren implementiert werden, die bei erkannter Störung automatisch entsprechende Gegenmaßnahmen einleiten.

Ziel der Fehlerbehandlung ist es, erkannte Fehler entweder vollständig zu beheben oder deren Auswirkungen auf das System zu minimieren. Dabei kann die Fehlerbehebung sowohl präventiv – durch das frühzeitige Erkennen potenzieller Probleme – als auch reaktiv während der Fahrt erfolgen, um kritische Zustände zu vermeiden. Diese Mechanismen tragen wesentlich zur Erhöhung der Resilienz des Gesamtsystems bei. Nach Umsetzung der Fehlererkennung und -behandlung erfolgt eine erneute Simulation, um die Wirksamkeit der entwickelten Maßnahmen zu überprüfen.

Auf dem linken Pfad, wie in Abbildung 3.2 dargestellt, wird eine ergänzende Strategie verfolgt. Hier stehen Unittests im Mittelpunkt, die bestehende Fehler im Code sichtbar machen und zukünftig dazu beitragen sollen, Fehler bereits vor ihrer Entstehung zu vermeiden. Durch Unittests kann die Funktionsfähigkeit einzelner Softwarekomponenten gezielt überprüft werden. Im Verlauf der Projektarbeit konnten mit ihrer Hilfe bereits vorhandene Fehler in der Software identifiziert und behoben werden. Auch perspektivisch bieten Unittests einen großen Mehrwert, da neu entwickelter Code frühzeitig auf Fehler geprüft werden kann. So wird verhindert, dass sich Fehler unbemerkt einschleichen, lange im System bestehen bleiben und später an anderer Stelle schwer nachvollziehbare Probleme verursachen. Die Erstellung und Integration der Unittests wird in Abschnitt 10.2 näher erläutert. Darüber hinaus ermöglicht die entwickelte CI-Pipeline ein automatisiertes Testen nach jeder Änderung. Durch dieses kontinuierliche Testverfahren können fehlerhafte Implementierungen sowie Fortschritte schnell und zuverlässig erkannt werden.

Durch die Aufteilung in verschiedene Themenbereiche konnten Fortschritte in mehreren Bereichen parallel erzielt werden. Die Mitglieder:innen der Projektgruppe konnten sich entsprechend ihrer Schwerpunkte aufteilen und unabhängig an unterschiedlichen Komponenten arbeiten. So entstand frühzeitig eine funktionsfähige End-to-End-Grundstruktur, die im weiteren Verlauf kontinuierlich erweitert und verbessert wurde. Probleme, die im Zusammenspiel der Komponenten auftraten, konnten durch enge Abstimmung im Team erkannt und gezielt behoben werden. Diese parallele Arbeitsweise spiegelt sich in den zwei Pfaden von Abbildung 3.2 wider. Alle beschriebenen Schritte und Entwicklungen tragen gemeinsam zur Erhöhung der Resilienz des Gesamtsystems bei. Jede einzelne Komponente unterstützt dieses

Ziel im Zusammenspiel mit den anderen. Das genauere Vorgehen sowie die Herausforderungen, die im Projektverlauf auftraten, werden in den folgenden Kapiteln im Detail beschrieben. Auf diese Weise wurde systematisch daran gearbeitet, die Resilienz von JARVIC nachhaltig zu stärken.

[1] https://fonts.google.com/icons

Kapitel 4

Vorgehensweise

In diesem Kapitel wird beschrieben, wie sich die Projektgruppe organisiert hat. Zunächst werden kurz die Rahmenbedingungen der Projektgruppe beschrieben, danach werden die verwendeten Organisationskonzepte erläutert und evaluiert.

4.1 Voraussetzungen und Rahmenbedingungen

Die Projektgruppe bestand aus neun Personen mit unterschiedlichem Vorwissen und Hintergründen. Keiner der Teilnehmer:innen hatte zuvor Kontakt mit JARVIC oder Rennauto-Hardware gehabt. Den Mitgliedern standen ein Server und ein Raum von GET Racing zum gemeinsamen Arbeiten zur Verfügung.

Die Projektgruppe war sich von Anfang an einig, eine überwiegend agile Vorgehensweise zu verwenden. Agile Vorgehensmodelle in der Softwareentwicklung zeichnen sich dadurch aus, dass Software in iterativen Zyklen um kleine, funktionierende Inkremente erweitert wird. Durch das Reduzieren langfristiger Planung auf ein Mindestmaß können die Anwender:innen auf kurzfristige Änderungen der Projektbedingungen schnell reagieren.

Die Arbeit der Projektgruppe war von der Arbeit von GET Racing abhängig und die Mitglieder belegten neben der Projektgruppe auch noch weitere Module. Daher waren etwa durch Prüfungsphasen die Kapazitäten der Mitglieder und die Umstände der Projektgruppe konstanten Wandel unterworfen, was die Flexibilität eines agilen Vorgehens enorm wertvoll machte.

Während der gesamten Dauer der Projektgruppe fanden wöchentliche Meetings (sogenannte Weeklys) statt, um den Fortschritt nachzuverfolgen und die Arbeit der kommenden Woche zu planen. Die Weeklys dienten als zentrales Kommunikations- und Organisationsmittel der Gruppe und waren äußerst nützlich.

Die Idee für diese Meetings wurde vom agilen Vorgehensmodell Scrum [SS25] übernommen. In Scrum gliedert sich die Entwicklungsarbeit in uniforme Zeitabschnitte, sogenannte Sprints.

Zu Beginn jedes Sprints findet eine Planungsbesprechung für den Sprint (das sogenannte Sprint Planning) und zum Ende jedes Sprints finden zwei Besprechungen zu Analyse und Bewertung des Sprints statt (Sprint Review und Sprint Retrospective).

Als Treffen zur Besprechung der letzten Woche und Planung der nächsten Woche setzen die Weeklys grob Sprint Review, Retrospective und Planning in einem um. In den Weeklys der Projektgruppe wurde die Arbeit der vergangenen Woche besprochen und die nächste Woche geplant, insofern wurde das Konzept von Scrum grob übernommen.

Tägliche Besprechungen (Dailys) aus Scrum wurden ebenfalls in Betracht gezogen, jedoch nicht angewandt, da der erhöhte Zeitaufwand den Vorteil einer engeren Abstimmung nicht aufwog. Der wöchentliche Rhythmus bot die ideale Balance zwischen regelmäßiger Fortschrittsplanung und flexibler Zeiteinteilung für die einzelnen Mitglieder.

4.2 Rollen und Kleingruppen

Es gab über den gesamten Zeitraum der Projektgruppe zwei feste Rollen, daneben haben sich mit der Zeit einige Teammitglieder auf bestimmte Themenbereiche spezialisiert. Als feste Rollen gab es ein Berichtsteam von zwei Personen und die Gruppenleiterin.

Das Berichtsteam protokollierte laufend den Fortschritt der Gruppe. Die Gruppenleiterin behielt die Übersicht über den Gesamtzustand des Projekts und stellte sicher, dass die Entwicklungsarbeit der Gruppe auf die Ziele fokussiert blieb. Neben den spezifischen Aufgaben dieser Rollen leisteten die Gruppenleiterin und das Berichtsteam auch Entwicklungsarbeit wie der Rest des Teams.

Um für die Entwicklung an mehreren Aufgaben gleichzeitig arbeiten zu können, hat die Projektgruppe Kleingruppen für bestimmte Themenbereiche gebildet. Für die Einteilung wurden zwei Ansätze getestet.

Im ersten Ansatz wurden feste Kleingruppen gebildet, die über einige Wochen an einer spezifischen Aufgabe zusammen arbeiten sollten. Jedes Teammitglied wurde genau einer Kleingruppe zugeteilt. Dieser Ansatz war nicht erfolgreich. Die Kleingruppen waren häufig sehr ungleich ausgelastet.

Eine flexible Umverteilung der Mitglieder war nur schlecht möglich, da bei einem Gruppenwechsel viel Zeitaufwand für die Einarbeitung in das andere Thema nötig war. Es kam zu Silo-Effekten: Es gab einige Expert:innen für bestimmte Themenbereiche, die untereinander sehr wenig kommuniziert haben. Durch den großen Einarbeitungs-Aufwand waren Krankheitsfälle der Expert:innen ein großes Problem für den Fortschritt der Gruppe.

Der zweite Ansatz war flexibler, mit lockerer organisierten Kleingruppen. Insbesondere gab es Überschneidungen unter den Kleingruppen, da nun die meisten Gruppenmitglieder in

mehrere Aufgabenbereiche eingearbeitet waren. Zudem wurden feste Zeiten vereinbart, in denen das Team zusammen in einem Raum gearbeitet hat, um engere Kommunikation und Zusammenarbeit zu ermöglichen.

Durch die höhere Flexibilität und engere Kommunikation gab es weder Leerlauf noch Überlastung innerhalb der Kleingruppen, da die Gruppengröße der Aufgabe angepasst werden konnte. Mit diesem Ansatz konnten sowohl lang- als auch kurzfristige Aufgaben zeiteffizient bearbeitet werden und problematische Silo-Effekte wurden vermieden.

4.3 Zeitliche Planung

Die kurzfristig anstehenden Aufgaben wurden mithilfe des Weekly-Protokolls und eines Issue Boards im Repository der Projektgruppe organisiert (mehr dazu in Kapitel 5). Das Issue Board war dabei in sechs Kategorien eingeteilt: „Open“, „Ready“, „In Progress“, „Blocked“, „Closed This Week“und „Closed“. Alle Aufgaben, die unter der Woche fertiggestellt wurden, wurden zunächst in 'Closed This Week' eingeordnet.

Die Issues in dieser Spalte wurden im Weekly besprochen und dann in „Closed“verschoben, womit das Berichtsteam die Arbeit der Gruppe protokolliert hat. Zudem erhielt damit die ganze Gruppe einen Überblick über den Gesamtfortschritt und konnte so besser das weitere Vorgehen besprechen.

Zur langfristigen Planung hat die Projektgruppe insgesamt drei Hilfsmittel verwendet.

Das erste Hilfsmittel war eine Fehlerliste, die zu Beginn der Projektgruppe erstellt wurde. Dafür wurde JARVIC auf Fehlerquellen analysiert und diese Fehler nach Priorität geordnet (mehr dazu in Kapitel 6). Diese Liste war als Sammlung der Arbeitsziele vom Product Backlog von Scrum inspiriert, wurde aber anders als das Product Backlog nicht zur kurzfristigen Planung verwendet.

Wenn die Projektgruppe eine Resilienzmaßnahme für einen Fehler fertiggestellt hatte, konnte der nächste zu bearbeitende Fehler der Liste entnommen werden. Dadurch, dass sich der nächste Fehler häufig durch neue Features der Fehlerinjektions-Infrastruktur ergeben hat (mehr dazu in Kapitel 7), wurde die Fehlerliste nur manchmal verwendet. Insgesamt war die Fehlerliste mäßig nützlich für die Arbeit der Projektgruppe.

Das zweite Hilfsmittel zur langfristigen Organisation war ein Zeitplan mit Meilensteinen. Aufgrund der festen zeitlichen Begrenzung der Projektgruppe wurde dieses nicht-agile Hilfsmittel verwendet, um Richtwerte für den Gesamtfortschritt zu setzen.

Ein Meilenstein bestand aus einem Zeitpunkt, kombiniert mit einem Prozentanteil bearbeiteter Fehler von der Fehlerliste. Die Meilensteine waren nicht nützlich für die Arbeit der Projektgruppe. Dies lag daran, dass sich einige Male Rahmenbedingungen kurzfristig geändert

haben und manche Vorhaben deutlich zeitaufwändiger waren als erwartet. So waren zum Beispiel der Zeitaufwand der Migration von JARVIC auf ROS2 nicht präzise planbar, da diese von Faktoren außerhalb der Kontrolle der Projektgruppe abhängig war. Aufgrund dieser Probleme hat die Projektgruppe früh aufgehört, sich an den Meilensteinen zu orientieren.

Das dritte Hilfsmittel zur längerfristigen Planung waren monatliche Besprechungen, die sogenannten Monthlys. Wie oben beschrieben, wurden in den Weeklys vor allem die technischen Details der Aufgaben für die nächste Woche besprochen. Ohne ein Planungsinstrument, das über die nächste Woche hinausging, bestand die Gefahr, die langfristige Perspektive aus den Augen zu verlieren. Dieses Problem wurde erfolgreich durch die Einführung der Monthlys gelöst.

In den Monthlys wurden die Ziele für den kommenden Monat festgelegt und in einer Liste dokumentiert. Diese Liste wurde anschließend in das Protokoll jedes Weeklys innerhalb des Monats übernommen. Am Ende jedes Weeklys wurde sie gemeinsam durchgegangen, um den Fortschritt der gesetzten Ziele zu überprüfen.

Diese Methode ermöglichte zusammen mit der flexiblen Kleingruppenstruktur, kurzfristig mehr Mitglieder einzubinden, falls sich eine Aufgabe aufwändiger war als erwartet. Der gewählte Zeitraum von einem Monat war ideal: Er war lang genug, um längerfristige Planungen zu ermöglichen, aber auch kurz genug, um flexibel auf veränderte Rahmenbedingungen zu reagieren – etwa die Migration von JARVIC auf ROS2.

Zudem konnten unregelmäßig anfallende Aufgaben, wie das Aufsetzen des Simulators PacSim, gezielt eingeplant und bearbeitet werden. Insgesamt trugen die Monthlys maßgeblich dazu bei, dass die Gruppe den Überblick über ihre längerfristigen Aufgaben behielt, ohne kurzfristige Anpassungen zu behindern. Sie erwiesen sich als äußerst nützlich für die Strukturierung der Projektarbeit."

4.4 Zusammenfassung

Zusammenfassend war die größte Herausforderung an die Organisationsstruktur der Projektgruppe, das richtige Mittelmaß aus langfristiger Planung und kurzfristiger Flexibilität zu finden. Die Projektgruppe hat einen Ansatz mit recht starren Rollen und Plänen erprobt, der die Arbeit der Gruppe aber mehr eingeschränkt als gefördert hat. Daher wurde dieser Ansatz schnell verworfen.

Mit einem flexibleren Kleingruppen- und Planungsmodell konnte die Projektgruppe viel effizienter und effektiver arbeiten. Durch die losere Kleingruppenstruktur mit Mitglieder, die in mehrere Themenbereiche eingearbeitet waren, konnten für jede Aufgabe so viele Mitglieder eingeteilt werden, wie für ihre rechtzeitige Fertigstellung nötig.

Zudem war die Gruppe dadurch deutlich weniger auf die Arbeit einzelner Expert:innen angewiesen und somit weniger anfällig gegenüber Ausfällen durch Krankheit. Das regelmäßige Besprechen der Monthly-Ziele in den Weeklys verschaffte der Gruppe eine gute Übersicht über ihren Fortschritt und ihre Planung. Gleichzeitig ermöglichte es ihr, flexibel auf veränderte Rahmenbedingungen zu reagieren. Durch Anwendung dieser flexibleren Konzepte reduzierte sich der Organisationsaufwand deutlich, während die investierte Planung die Zusammenarbeit der Gruppe spürbar verbesserte.

Kapitel 5

Developer Experience

In diesem Kapitel werden die Konzepte und Umsetzungen der Projektgruppe bezüglich der Developer Experience vorgestellt, die über GitLab und die Grundlagen von ROS und JARVIC hinausgehen. Zunächst wird der Ansatz der Simulation und die verschiedenen im Entwicklungsworkflow dafür eingesetzten Tools vorgestellt (vgl. ??). Dann wird die Entwicklungsumgebung und der Entwicklungsworkflow für die Arbeit an JARVIC präsentiert (vgl. Abschnitt 5.3). Abschließend werden weitere Services beschrieben, die für die Umsetzung der Projektgruppe wichtig sind (vgl. Abschnitt 5.4).

5.1 Simulatoren

Um das Vertrauen in die Fahrsoftware zu stärken, ist umfassendes Testen essenziell, um möglichst viele Fehler frühzeitig zu erkennen. Idealerweise erfolgt dies direkt am Fahrzeug, um eine realistische Testumgebung zu gewährleisten. Allerdings müssen dabei verschiedene Kosten und Herausforderungen sorgfältig abgewogen werden: Das Testen am realen Fahrzeug ist zeitaufwendig (Aufbau, Transport etc.), ressourcenintensiv (Streckenaufbau, Abnutzung etc.) und mit Risiken verbunden, da Unfälle oder Schäden am Fahrzeug auftreten können.

Dadurch kann diese Art des Testens nur bedingt eingesetzt werden und wird in der Praxis üblicherweise für mehrere gesammelte Anpassungen auf einmal genutzt und nicht zur Prüfung kontinuierlicher, kleinerer Verbesserungen. Gerade aufgrund der Komplexität des Systems ist es jedoch auch schon bei kleineren Änderungen sinnvoll, intensiv zu testen, um Fehler schnell zu identifizieren und Verbesserungen evaluieren zu können. Ein möglicher Ansatz dafür ist der Einsatz von Simulation, welche im Laufe der Projektgruppe im Fokus stand. Hierbei wurde verschiedene Software zur Umsetzung von Simulation betrachtet, welche im Folgenden vorgestellt und aus Perspektive der Projektgruppe bewertet wird.

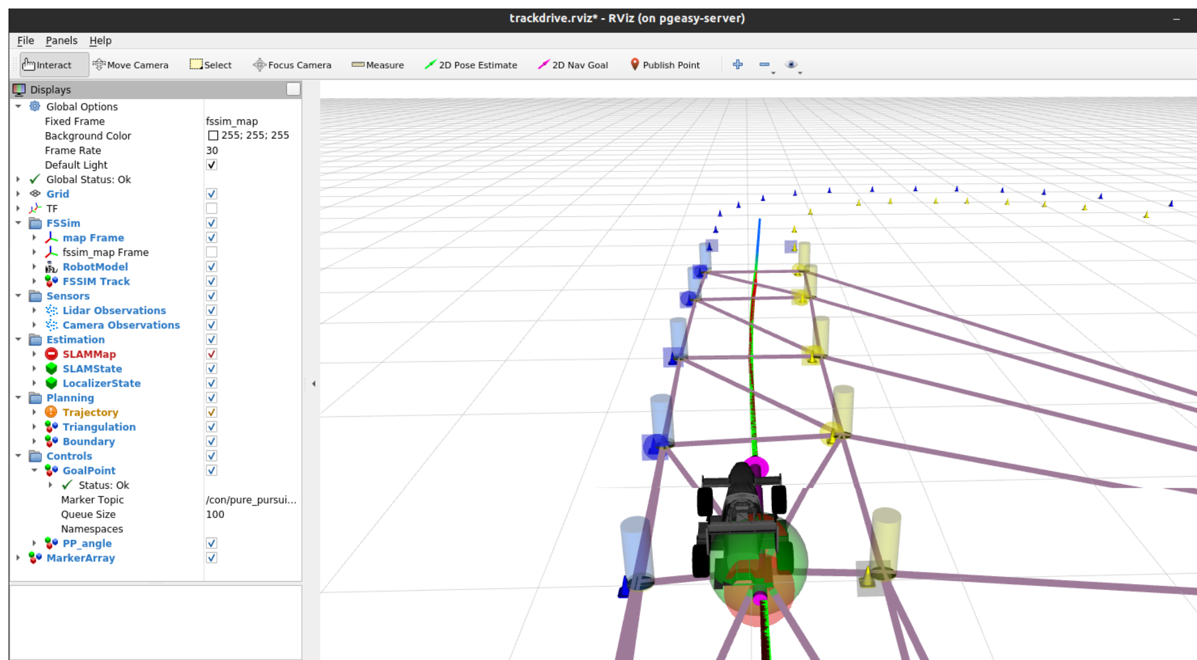


Abbildung 5.1: FSSIM während der Ausführung einer Trackdrive-Mission.

5.1.1 FSSIM

Der Formula Student Simulator (FSSIM) ist ein auf ROS und der Simulationsumgebung Gazebo basierender Simulator, der speziell für die Formula Student vom Akademischen Motorsportverein Zürich entwickelt und veröffentlicht wurde. Ein Screenshot von RViz mit laufendem FSSIM und der Fahrsoftware JARVIC ist in Abbildung 5.1 dargestellt. Als Beispiel wird darin die Trackdrive-Mission der Formula-Student-Driverless-Disziplin ausgeführt. Der Simulator kann durch eine Konfigurationsdatei angepasst werden. So können bspw. unterschiedliche Sensorkonfigurationen verwendet oder verschiedene Strecken geladen werden. FSSIM bietet zudem zur Automatisierung der Simulation direkt ein Skript an, mit welchem automatisch nach dem Start der Simulation die Fahrsoftware gestartet werden kann und ein Rosbag der Simulation aufgezeichnet wird. FSSIM simuliert die Bewegungen des Autos, indem von ROS ausgehend Steuerungsnachrichten an den Simulator gesendet werden (Lenkwinkel und Beschleunigung/Verzögerung). Umgekehrt sendet FSSIM Daten an die Fahrsoftware, z. B. die Positionen der für die Fahrbahnbegrenzung verwendeten Leitkegel. Letztere werden dabei direkt an die Fahrsoftware gegeben (ggf. mit stochastischem Rauschen etc.) und überspringen somit die Perception. FSSIM simuliert also keine realen Sensordaten, sodass die Nodes der Perception bei der Simulation nicht verwendet und entsprechend auch nicht getestet werden. Da die Perception jedoch einen wichtigen Teil der Fahrsoftware ausmacht, wurde sich im Rahmen der Projektgruppe mit ergänzenden Simulationsumgebungen beschäftigt, welche zusätzlich rohe Sensordaten simulieren können. JARVIC wurde durch GET racing bereits vor

Beginn der Projektgruppe an den FSSIM angebunden und stellt bereits seit Längerem einen wichtigen Teil der Qualitätssicherung des Teams dar.

5.1.2 PacSim

Im Zuge der Projektgruppe wurde durch die ROS2 Migration die zugrundeliegende Architektur von JARVIC umfassend geändert (vgl. Abschnitt 5.5). Eine der daraus folgenden Konsequenzen war der Wechsel von FSSIM als Simulator zum Planning and Controls Simulator (PacSim). Dies wurde notwendig, da der zuvor verwendete FSSIM nicht mit ROS2 kompatibel ist und PacSim stattdessen den gleichen Simulationsbereich für ROS2 abdeckt.

Hier übernahm die Projektgruppe eine führende Rolle in der Umstellung. Nicht nur die reine Migration auf ROS2 der relevanten Nodes wurde durchgeführt, sondern auch die Anpassung an die geänderte Schnittstelle des PacSim. Zudem wurde auch die automatische Simulation in der CI umgestellt, sodass diese stattdessen den PacSim als Simulator unterstützt. In der Verwendung durch die Nutzer ändert sich beim PacSim wenig, da dieser in der Endnutzerhandhabung sehr dem zuvor verwendeten FSSIM ähnelt und z. B. dieselbe Visualisierungssoftwares unterstützt.

5.1.3 CARLA

Eine Möglichkeit, um auch die Perception in die Tests miteinzubeziehen, stellt der Open-Source-Simulator CARLA [Dos+17] dar, welcher in Abbildung 5.2 zu sehen ist. Dieser wurde nicht spezifisch für die Formula Student entwickelt und ist somit für das Testen allgemeiner autonomer Fahrsoftware einsetzbar. Dabei wird mithilfe der Unreal Engine eine realistischere 3D-Umgebung simuliert, in welcher das Fahrzeug mit simulierten Sensoren, wie Kameras und Lidar, die Umgebung wahrnimmt. Dies würde das volle Testen der Perception ermöglichen. Auch weitere Einschränkungen, wie z. B. leichten Regen, können mit CARLA abgebildet werden. Für CARLA existiert eine ROS-Bridge, wodurch eine Anbindung an das ROS-Ökosystem möglich ist.

CARLA wurde gegenüber anderen Formula Student-Simulatoren präferiert, da aufgrund der wesentlich größeren Community voraussichtlich ein deutlich besserer und längerfristiger Support gewährleistet ist. So ist beispielsweise die letzte Änderung des in Unterabschnitt 5.1.1 beschriebenen FSSIM bereits 5 Jahre alt, sodass dieser mit neueren ROS-Versionen nicht mehr kompatibel ist. CARLA besitzt außerdem aufgrund der allgemeinen Anwendbarkeit im autonomen Bereich, im Gegensatz zu speziellen Simulatoren für die Formula Student, eine deutlich größere Verbreitung und würde den Mitglieder von GET racing voraussichtlich mehr langfristigen Nutzen auch nach Abschluss des Studiums bringen.

Abbildung 5.2: CARLAs grafische Oberfläche während der Simulation [Dom21].

Aufgrund verschiedener Probleme wurde CARLA jedoch nicht weiter betrachtet. So ist CARLA mit einer Anforderung von etwa 32 GB RAM und einer NVIDIA RTX 3070 GPU sehr ressourcenintensiv. Diese Ressourcen standen im Rahmen der Projektgruppe nicht zur Verfügung. Da die Integration von JARVIC in CARLA, insbesondere durch die Integration einer Fehlerinjektion, eine Anpassung an der Simulation selbst benötigt hätte, hätte diese selbst kompiliert werden müssen. Dies hätte ebenfalls sehr viel Komplexität und Aufwand bedeutet, da bspw. eine von selbstkompilierte modifizierten Version der Unreal Engine verwendet werden muss. Daher wurde die Integration von CARLA mit JARVIC nicht weiter fortgesetzt und Alternativen untersucht.

5.1.4 FSDS

Eine weitere Simulationsmöglichkeit, welche speziell für die Formula Student ausgelegt ist, ist der Formula Student Driverless Simulator (FSDS). FSDS basiert auf der Unreal Engine 4 mit der Fähigkeit, dreidimensionale Szenerien und damit auch rohe Kamerabilder als Sensoreingaben für die Fahrsoftware zu simulieren. Auch ein Lidar kann im FSDS simuliert und damit kann dessen Verarbeitung getestet werden. Zur Simulation wird eine angepasste Version des Microsoft AirSim genutzt, welcher jedoch nicht aktiv weiterentwickelt wird. Hierdurch können in der Zukunft Supportprobleme auftreten, was bei dessen Einsatz bedacht werden

Abbildung 5.3: Beispiel einer dreidimensional simulierten Umgebung im FSDS.

muss. Es ist auch möglich, das Wetter in der Simulationsumgebung zu kontrollieren. Somit können unterschiedliche Effekte auf die Perception evaluiert werden.

FSDS wurde auch durch GET racing in der Vergangenheit bereits als Option evaluiert und Teile der Anbindung von JARVIC an FSDS sind bereits vor Beginn der Projektgruppe entwickelt worden. Diese konvertieren eingehende Sensordaten in das entsprechende Format für die Perception-Nodes. Für die Steuerungsbefehle in die umgekehrte Richtung von JARVIC an FSDS wurde durch die Projektgruppe ein entsprechender ROS-Node als Interface geschrieben. Ein Screenshot von FSDS in der Formula Student Umgebung ist in Abbildung 5.3 dargestellt.

Nach der Migration auf ROS2 wurde durch die Projektgruppe ein signifikanter Teil der Anbindung von FSDS an das JARVIC Projekt umgesetzt. Hier war nicht nur die die Umstellung auf ROS2 ein zu lösendes Problem, sondern auch die für den FS225 neu gewählte Perception Architektur basierend auf einer einzelnen Mono-Kamera. Der FSDS ist somit inzwischen gänzlich einsatzbereit und wird aktiv vom GET racing Team zur Optimierung und Überarbeitung der eigenen Algorithmen inklusive der Perception eingesetzt.

Anschließend an die Projektgruppe wäre zudem der automatisierte Einsatz des FSDS innerhalb der CI denkbar, welcher aus zeitlichen Gründen nicht mehr umgesetzt werden konnte. Hier wäre insbesondere das vollautomatisierte Ausführen mehrerer durch FLEX (vgl. Kapitel 8) definierter Fehlerszenarien ein spannendes Einsatzgebiet der im Zuge der Projektgruppe entwickelten Systeme.

5.1.5 Simulationsdeterminismus

Ein Problem aller untersuchten Simulatoren ist, dass diese keine deterministische Simulation garantieren und in der Praxis recht unterschiedliche Simulationsläufe ergeben. Dies führt dazu, dass Änderungen an der Fahrsoftware nicht exakt miteinander verglichen werden können, da nicht nur die Änderung selbst Einfluss auf Unterschiede in der Simulation hat, sondern z. B. auch die aktuelle Rechenauslastung der ausführenden Maschine bzw. im Allgemeinen deren Hardwareausstattung. Ein vollständiger Determinismus in der Simulation ist z. B. wegen Multithreading und der Physiksimulation außerordentlich schwierig zu erzielen und inhärent durch die verwendete Software, z. B. ROS, behindert.

Dennoch wurden sich im Zuge der Projektgruppe Gedanken gemacht, wie Vergleichbarkeit realisiert werden könnte. Ein Aspekt ist die Verminderung von Umgebungseinflüssen. Hier wurde einige Arbeit investiert, um die Rechenauslastung der Server während der Simulationsausführung nicht durch weitere parallele Prozesse zu beeinflussen. Auch umfassendere Entwicklungen wurden diskutiert, unter anderem die Verringerung von Parallelisierung innerhalb von ROS durch den vermehrten Einsatz von strikt sequentiellen Abläufen.

5.2 Integrierte Simulation

Für eine möglichst einfache Verwendung wurde die Simulation in den bestehenden Entwicklungsworkflow integriert.

5.2.1 RBB

Der Rosbag Bazaar[1] ist ein vom Akademischen Motorsportverein Zürich (AMZ) entwickelter Webservice. Dieser erlaubt das automatisierte Analysieren von Rosbags. Dazu können Extraktionen aus Rosbags ausgeführt werden, um Metriken oder andere Daten zu ermitteln, welche beliebig weiterverarbeitet werden können. So können beispielsweise Graphen aus den Daten erstellt oder auch ein Video der Simulation automatisch generiert werden. In Abbildung 5.4 ist das Interface einer RBB-Instanz zu sehen. Neben einer Auflistung der Rosbags im linken Teil der Grafik befindet sich daneben die generierte Map mit dem Streckenverlauf der ausgewählten Simulation.

Eine Instanz des RBB wurde für den Eigengebrauch von GET racing schon vor Beginn der Projektgruppe aufgesetzt. Diese wird dafür genutzt, bei Testfahrten aufgenommene Rosbags zu analysieren. Auch für eine simulierte Umgebung kann RBB eingesetzt werden. Mit der Simulationsfunktion kann ein Docker-Image ausgeführt und anschließend ein daraus erstellter

[1] https://github.com/AMZ-racing/rbb_core

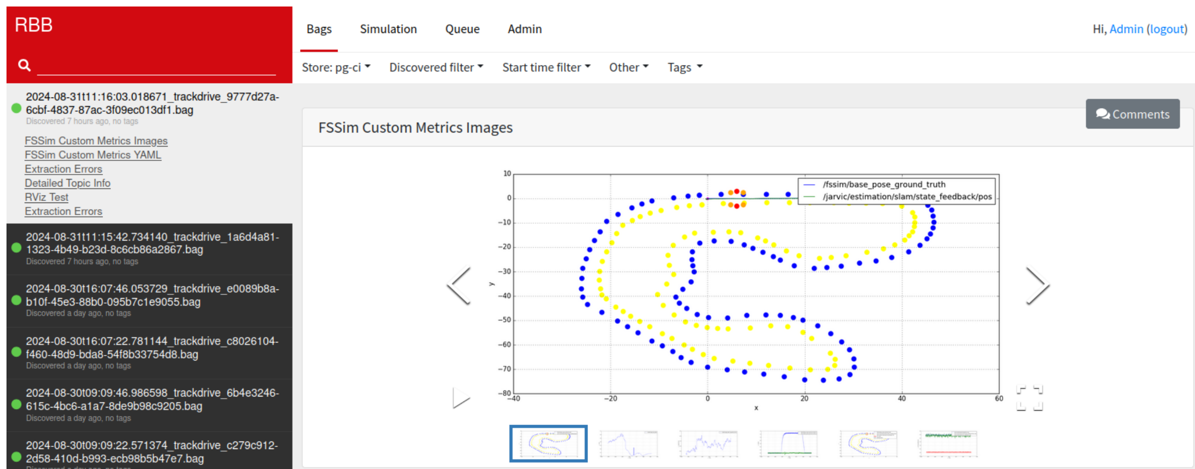


Abbildung 5.4: Visualisierung einer erkannten Strecke im RBB.

Rosbag analysiert werden. Dies kann beispielsweise in einer CI-Umgebung eingesetzt werden, um automatisch die Quellcodeänderungen durch Simulationen zu testen, so wie es bei GET racing der Fall ist. Dabei wird in der CI ein Image mit JARVIC und FSSIM gebaut, das daraufhin genutzt wird, um automatisch eine Simulation im RBB mit dem erstellten Image anzustoßen.

5.2.2 CI Integration

Im Rahmen der Projektgruppe wurde sich damit beschäftigt, die Simulation von JARVIC mit FSSIM sowie dem PacSim direkt in der CI ausführen zu können. Die resultierenden Rosbags sollten dann in den RBB hochgeladen und die Ergebnisse der RBB-Analyse daraufhin in der Pipeline verfügbar gemacht werden. Eine vorhergehende Projektgruppe hatte sich bereits mit einer ähnlichen Integration beschäftigt. Wegen der beschränkten Ressourcen seitens des Driverless Teams von GET racing wurden die resultierenden Ergebnisse jedoch über die Zeit nicht ausreichend gepflegt und dementsprechend funktionsunfähig. Dadurch konnte nur beschränkt durch diese nachfolgende Projektgruppe darauf aufgebaut werden.

Umgesetzt wurde diese Integration in die CI durch das Hinzufügen eines Simulationsjobs, welcher nach dem erfolgreichen Bauen von JARVIC ein entsprechendes Skript von FSSIM aufruft. Mit dem Skript führt FSSIM anhand einer Konfigurationsdatei verschiedene Simulationen aus und speichert die resultierenden Rosbags ab. Ein Beispiel dieser Konfigurationsdatei ist in Listing 5.1 dargestellt.

Die anschließende Kommunikation mit dem RBB ist in ein separates Skript ausgelagert. Dieses von der Projektgruppe erstellte Skript lädt die Rosbags in einen von der Projektgruppe betriebenen MinIO[2] Objektspeicher. Anschließend wird über die API des RBBs das Indexieren

[2] https://min.io/

```yaml
simulation_name: "Simulation"

robot_name: gotthard  # Name of the robot [string]
kill_after: 300      # After this time, repetition is stopped [s]
max_lap_count: 5

repetitions:
- {sensors_config_file: fssim_config/sensors/sensors_1.yaml, track_name:
   FSG.sdf, autonomous_stack: roslaunch common_meta
   generic_with_fssim_interface.launch launchfile:='$(find common_meta)/
   missions/trackdrive.launch' }
```

Listing 5.1: Ausschnitt einer Beispielkonfiguration für das automatische Ausführen von FSSIM.

in MinIO angestoßen, um neu hinzugefügte Rosbags zu finden und automatisch zu analysieren. Um die Ergebnisse der Analyse in GitLab über einen Report der Metriken verfügbar zu machen, prüft das Skript in regelmäßigen Abständen den Fortschritt der Analyse. Bei einer Merge Request wird ein Kommentar mit einem Link zum analysierten Rosbag hinzugefügt, damit jegliche Ergebnisse (Video etc.) einfach auffindbar sind. Der beschriebene Ablauf der Simulation, Analyse und Rückführung der Ergebnisse in die CI ist in Abbildung 5.5 zusammengefasst.

Eine Herausforderung bei der Ausführung der Simulation in der CI ist, dass wegen der hohen Hardwareanforderungen von JARVIC zur gleichen Zeit nur eine Simulation pro Host laufen sollte. Andernfalls ist durch die hohe Auslastung mit einer gegenseitigen Beeinflussung der Ergebnisse zu rechnen. Dies könnte dafür sorgen, dass Nachrichten über ROS verzögert zugestellt werden und dadurch die Simulation verfälschen. Auch die Ausführung anderer Prozesse mit hohem Ressourcenverbrauch sollte auf dem Host vermieden werden, wie beispielsweise das Bauen von JARVIC in der CI-Pipeline. Dies wird durch Ressource Groups in der Konfiguration der GitLab CI gelöst, welche eine Gruppe von Jobs angeben, die einander gegenseitig in der Ausführung ausschließen. In der erstellten Resource Group sind die Build-, Test- und Simulationsjobs hinzugefügt worden. Ein Nachteil dieser Lösung ist zurzeit, dass nur der Simulationsjob fest durch ein `tag` in der GitLab CI auf einem GitLab-Runner läuft. Die Build- und Testjobs laufen ggf. alternativ auf dem GitLab-Server. Daher sollten sich die Jobs nicht immer gegenseitig ausschließen. Eine rein sequentielle Ausführung kann eine Alternative sein, jedoch können unter anderem die Jobs zum Linting problemlos parallel laufen und somit Pipelinezeit sparen. Dies sollte bei Einsatz des Systems bewusst sein und entsprechend der Ansprüche des Anwenders konfiguriert werden.

In Folge der Migration hin zu ROS2 (vgl. Abschnitt 5.5) ist der RBB jedoch aktuell nicht mehr einsetzbar, da der RBB ROS2 Bags nicht unterstützt. Die durch die Projektgruppe entwickelten

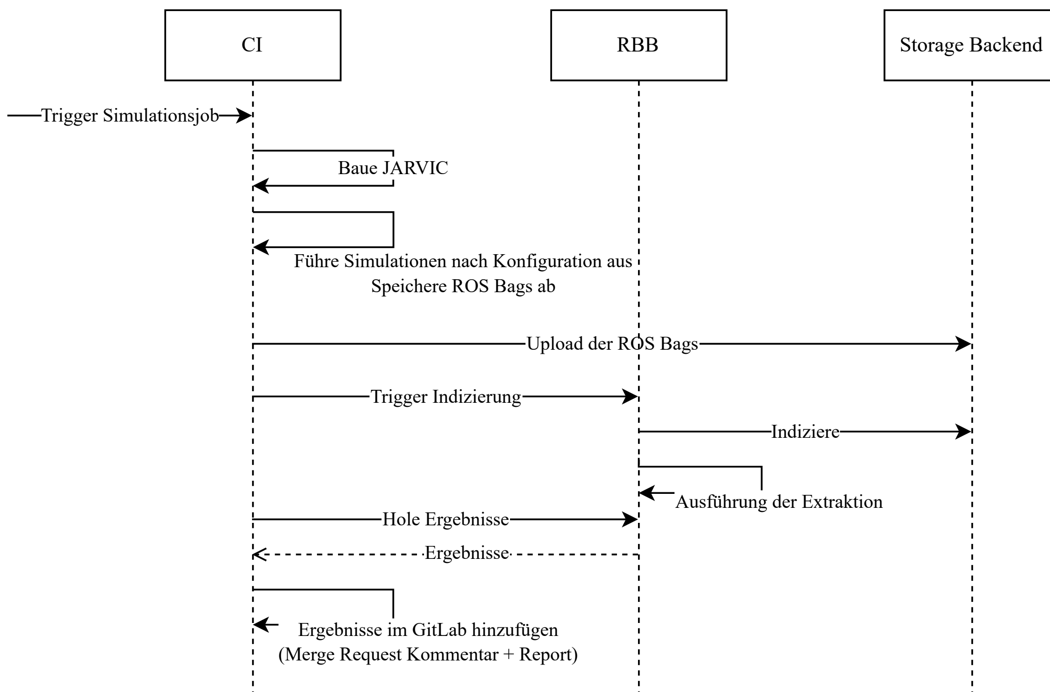


Abbildung 5.5: Ablauf der Simulation und Ergebnisauswertung in der CI.

Skripte wären jedoch, sobald ein Ersatz, für den RBB durch Dritte entwickelt wäre, auf dessen Schnittstelle hin anpassbar und somit wiederverwendbar.

5.3 Entwicklungsumgebung

Die Entwicklungsumgebung ist essenziell für die Arbeit innerhalb der Projektgruppe und es sollte eine möglichst störungsfreie Mitarbeit an der Software gegeben sein. Dazu wurde für die Arbeit innerhalb der Projektgruppe sowie für das Driverless Team der Entwicklungsworkflow von JARVIC vereinfacht.

Dabei sollten die Ziele sein, dass

- das Starten der Entwicklungsumgebung möglichst wenig Zeit in Anspruch nimmt. Dies ist essenziell, damit durch Reibungsverluste die Arbeitsaufnahme nicht beeinträchtigt und die betroffene Person demotiviert wird. So wenig produktive Zeit wie möglich sollte durch das Einstellen der Entwicklungsumgebung verloren werden.
- die Nutzung der Entwicklungsumgebung so einfach wie möglich ist sowohl für erfahrenere als auch weniger erfahrene Teilnehmer:innen der Projektgruppe. Einige Gruppenmitglieder besaßen bereits umfassende Erfahrung in der gemeinsamen Arbeit

an größeren Projekten, wohingegen manche im Zuge der Projektgruppe ihren ersten Kontakt damit erfuhren. Da jede Person so gut wie möglich eingesetzt werden soll, ist es eine Priorität, allen die volle Teilnahme zu ermöglichen und Hindernisse aus dem Weg zu räumen.

- der Entwicklungsworkflow vereinheitlicht wird. Trotz unterschiedlicher Hard- und Software sollte die Entwicklungsumgebung möglichst einheitlich bedienbar sein. Auch die Dokumentation zu Arbeitsabläufen ist in gleicher Weise anwendbar, sodass nur wenig spezifische Unterstützung für Einzelne aufgebracht werden muss.
- die resultierende Entwicklungsumgebung über den Zeitraum der Projektgruppe möglichst wenig bis gar nicht angepasst werden muss. Updates sollen im Hintergrund erfolgen, ohne dass es bei der Entwicklung bemerkt wird. Dies ist zum einen parallel zum Ziel der Vermeidung von Reibungsverlusten zu sehen, aber auch insbesondere relevant im Hinblick auf die Erweiter- und Anpassbarkeit der Umgebung an zukünftige Entwicklungen bei JARVIC. Das Hinzufügen von Dependencies z. B. sollte vor dem Anwender möglichst versteckt werden und nur zu minimaler manueller Interaktion führen.

Unter diesen Gesichtspunkten wurden die bisher von JARVIC bereitgestellten Möglichkeiten betrachtet:

Native Entwicklungsumgebung: Der intuitive Ansatz ist, JARVIC und seine Dependencies nativ auf den Geräten der Teilnehmer:innen aufzusetzen und damit zu entwickeln. Dies würde bei korrekter Installation dazu führen, dass alles funktioniert und Performanceverluste durch Virtualisierung vermieden werden. Der benötigte Wissensstand wird dadurch im Gegensatz zur Virtualisierung verkleinert. Es muss sich nur mit klassischen Paketmanagern auseinandergesetzt werden, nicht aber mit Containern oder anderen Virtualisierungssystemen.

Docker: Der andere durch JARVIC bereitgestellte Ansatz basiert auf Containerisierung mit Docker[3]. Dies wurde durch ein Shell-Skript `jdocker.sh` umgesetzt, welches im Hintergrund das Bauen und Starten eines Containers verwaltete. In diesem kann das JARVIC-System gebaut, getestet und ausgeführt werden. Zudem sind einige für die Arbeit mit dem System relevanten Programme darin mit installiert und eingerichtet.

Kurz nach dem Start der Projektgruppe wurde sich in der Diskussion gegen die Verwendung der nativen Entwicklungsumgebung entschieden. Die heterogene Ausstattung der Teilnehmer:innen bzgl. ihrer Hardware sowie Software stellt kurz- wie langfristig einen zu hohen Supportaufwand dar. Nativ wird JARVIC aufgrund der verwendeten Version von ROS (ROS Noetic) nur unter Ubuntu 20.04 unterstützt. In Anbetracht der Tatsache, dass kein Mitglied der Projektgruppe Ubuntu 20.04 verwendet, wäre entweder

[3] https://www.docker.com/

- der komplette Umstieg notwendig gewesen, was insbesondere im Hinblick auf die veraltete Version von Ubuntu als unerwünscht erachtet wurde, oder
- die Installation als Virtual Machine (VM) bzw. Dual-Boot-System notwendig. VMs erscheinen unzureichend, da bei diesen zum einen ein Performanceverlust befürchtet wird und zum anderen die Verwendung im Konflikt mit der Zielsetzung der möglichst reibungslosen Arbeit angesehen wird, da zum Beispiel für die Entwicklung das an die eigenen Bedürfnisse angepasste Hauptsystem verlassen werden muss. Allgemein gestaltet sich die Nutzung einer VM für ihre Verwender weniger komfortabel als die Nutzung eines nativen Systems.

Im Folgenden wird daher über die Anpassung der bestehenden Entwicklung im Container (Unterabschnitt 5.3.1) berichtet. Außerdem wurde ein Mittelweg gefunden, bei dem native Entwicklung auf einem Server ermöglicht wird, der in Unterabschnitt 5.3.2 vorgestellt wird.

5.3.1 Containerbasierte Entwicklung

Zu Beginn der Projektgruppe wurde ein auf *podman*[4] basierendes System verwendet. Hierbei wurde *docker-compose* genutzt, um alle für die Entwicklung notwendigen Abhängigkeiten zu starten und zu verwalten.

Dieses System benötigte jedoch leider weiterhin diverse kleinere Anpassungen abhängig von dem genutzten Host-System. Im Zuge der ROS2 Migration wurde sich entsprechend entschieden, den vorherigen Ansatz grundsätzlich zu ändern und zu vereinfachen. Anstatt dass native und containerbasierte Systeme unterschiedlich behandelt werden, sollten beide nun zentral ihre Abhängigkeiten definieren. Entsprechend wurde das zuvor bereits vorhandene `update_dependencies.sh` Skript umfassend umstrukturiert und um weitere Abhängigkeiten wie z. B. ROS2 erweitert, welches vorher durch den Nutzer selbst manuell installiert werden musste. Dabei war das Ziel, ein Skript zu erhalten, dass durch Ausführen jederzeit die aktuell genutzte Umgebung auf den aktuellsten Stand bzgl. der für die Arbeit mit JARVIC notwendigen Software bringen kann. Alle Abhängigkeiten, auch wenn sie nur für einen Teil der Nutzer:innen relevant sein sollten (z. B. für bestimmte Simulationsframeworks) sollten darin spezifiziert und installiert werden.

Dies führte dazu, dass anstatt von *podman* oder *docker* direkt, eine Zwischenschicht verwendet werden konnte, welche die genutzten Container direkter mit dem Host-System verbindet, z. B. durch integriertes X Forwarding. Hierbei wurde sich für *distrobox*[5] entschieden, welches die zuvor notwendigerweise selbst entwickelten Integrationen bereits mit sich bringt.

[4] https://podman.io/
[5] https://distrobox.it/

Das Aufsetzen einer neuen Entwicklungsumgebung wurde dadurch auf 3 Schritte reduziert:

1. *distrobox* installieren
2. Mit *distrobox* einen neuen Container für Ubuntu 22.04 erstellen
3. Innerhalb dieser das `update_dependencies.sh` Skript ausführen

Zudem können neu notwendige Dependencies dem `update_dependencies.sh` Skript hinzugefügt werden und jedes Teammitglied kann durch erneutes Ausführen des Skriptes diese installieren.

5.3.2 Serverbasierte Entwicklung

Als Ergänzung zum containerbasierten Ansatz wurde die mit dedizierten Grafikkarten zur Verfügung gestellte Hardware genutzt, um eine via Internet erreichbare Entwicklungsumgebung zur Verfügung zu stellen. Die Idee war dabei, den Server nativ für JARVIC einzurichten. Dadurch wurde die hinzugefügte technische Komplexität der Containerisierung umgangen und direkt auf die Grafikkarte zugegriffen. Alle Teilnehmer:innen der Projektgruppe haben einen eigenen Zugang mit eigener JARVIC-Instanz erhalten. Dabei wird auf dieselbe zugrundeliegende Installation zugegriffen, um nur einmaligen Wartungsaufwand zu haben. Dieser wird somit durch einen einzigen Verantwortlichen aufgefangen.

5.4 Weitere Unterstützung der Developer Experience

Auch außerhalb der direkten Entwicklungsumgebung traten verschiedene Probleme auf, die es zur Verbesserung der Developer Experience zu lösen gilt. In diesem Abschnitt soll eine kurze Übersicht über die im Zuge der Projektgruppe bereitgestellten Services gegeben werden und welche Zwecke sie erfüllen.

Dabei gilt für alle Services, dass sie nur innerhalb des eigenen Tailscale-VPNs der Projektgruppe über verschiedene Subdomains unter `pgeasy.de` erreichbar sind. Für Tailscale wurde sich entschieden, damit die Services nicht unnötig frei im Internet zugänglich gemacht werden müssen. Der Zugriff für Projektgruppenteilnehmer:innen wird dennoch durch die schnelle Installation und den geringen Wartungsaufwands einfach gehalten.

VaultWarden ist die serverseitige Open-Source Implementation des Passwortmanagers BitWarden[6]. Der Service wird innerhalb der Projektgruppe als Datenbank für alle Zugänge verwendet. Hierdurch besteht unter den Teilnehmer:innen keine Abhängigkeit und alle können auf alles problemlos zugreifen.

[6]https://bitwarden.com

Rosbag Bazaar (RBB) ist, wie in Unterabschnitt 5.2.1 ausgeführt, wichtig für die unabhängige Analyse von Fahrzeugdaten. In absehbarer Zeit wird voraussichtlich auf den vom GET racing betriebenen RBB umgestiegen.

MinIO ist ein Objektspeicher, der über eine S3 Schnittstelle als Datenspeicher genutzt werden kann. Hauptsächlicher Einsatzzweck ist aktuell der Betrieb des RBB. Auch eine darüber hinausgehende Nutzung ist denkbar.

GitLab Runner werden innerhalb der CI genutzt, um z. B. die Simulation (vgl. ??) auszuführen. Für die Projektgruppe wurden eigene Runner aufgesetzt, um die Laufzeit zu beschleunigen und Limitationen der durch GitLab zur Verfügung gestellten Shared Runner zu vermeiden.

RBB Runner sind - analog zu GitLab Runnern - Worker, welche innerhalb des RBB anfallende Rechenarbeiten, z. B. die Extraction auf Rosbags ausführen können. Hier wurde ein Runner ohne und einer mit GPU Anbindung aufgesetzt, um die Anforderungen des RBB bestmöglich zu erfüllen.

5.5 Migration auf ROS2

Während der Projektgruppe wurde die Notwendigkeit der Migration von ROS1 auf ROS2 innerhalb des Driverless Teams bewusst. Der Hauptgrund war die allgemeine Verlagerung des ROS-Ökosystems auf die neue Version, wodurch Dependencies für die alte Version nicht mehr weiter unterstützt würden. Da das erste Dreivierteljahr der Projektgruppe JARVIC noch auf ROS1 basierte, wurde zuerst für dieses System entwickelt. Gegen Ende 2024 begann das GET Racing Driverless Team mit der Umstellung auf ROS2, die zahlreiche Änderungen mit sich brachte und direkten Einfluss auf die Arbeit der Projektgruppe hatte.

Damit die von der Gruppe entwickelten Funktionen auch langfristig vom Driverless Team genutzt werden könnten, wurde sich dazu entschieden, diese ebenfalls auf ROS2 zu migrieren. Dies brachte einige Vorteile mit sich durch die modernere Architektur von ROS2, z. B. die Entfernung des ROS Cores und die konsistentere API. Jedoch musste durch die Umstellung ein signifikanter Aufwand in die Migration gesteckt werden.

5.6 Fazit

Insgesamt konnte so eine Developer-Experience für die Projektgruppe sichergestellt werden, welche die produktive Mitarbeit aller Teilnehmer:innen unterstützte. Durch den Fokus auf Anfängerfreundlichkeit können die resultierenden Anpassungen und neuen Funktionen auch dem GET racing Driverless Team zugutekommen, um neue Mitglieder schneller am Projekt

mitarbeiten lassen zu können. Die entkoppelte Infrastruktur half der Projektgruppe dabei, schneller iterieren zu können, ohne mit dem parallellaufenden Workflow von GET racing zu interferieren.

Kapitel 6

Fehleranalyse

Fehler werden nach [RAB19] in Kombination mit Störungen und Ausfällen definiert. Störungen (engl. *Fault*) treten demnach spontan auf und können dazu führen, dass Komponenten in einem System in einen Zustand übergehen, in welchem die Komponente ihre Spezifikation nicht mehr erfüllt, wodurch dann ein Fehler (engl. *Error*) vorliegt. Wenn dies dazu führt, dass das System seine Funktion nicht mehr erfüllen kann, wird von einem Ausfall (engl. *Failure*) gesprochen. Der Zusammenhang dieser Begriffe ist in Abbildung 6.1 schematisch dargestellt.

Zum Generieren einer Liste von möglichen Fehlern bietet sich eine Hazard and Operability Study nach [Kle18] an. Das Ziel dieser ist es, alle möglichen Fehler zu benennen. Dabei werden systematisch Leitwörter mit Komponenten kombiniert. So könnten beispielsweise als Komponente die Erkennung von Leitkegeln mit den Wörtern „zu sensibel“ und „nicht sensibel genug“ kombiniert werden. Dies stellt die Ereignisse dar, dass Leitkegel erkannt werden, wo es keine gibt, oder dass Leitkegel übersehen werden. Die Kombinationen, welche tatsächlich mögliche Ereignisse darstellen, werden einer Liste hinzugefügt.

Bei der Failure Mode and Effects-Analyse nach [Boz+10] werden mögliche Fehler aufgelistet und Kenngrößen mit Werten zwischen 1 und 10 geschätzt. Das können Größen, wie das von diesen Fehlern ausgehende Risiko, die Wahrscheinlichkeit des Auftretens sowie die Wahrscheinlichkeit, dass der Fehler entdeckt wird, sein. Diese Werte werden jeweils multipliziert, um eine Wertung für jeden Fehler zu bekommen, nach welcher diese sich ordnen lassen. Je größer das jeweilige Produkt ist, desto größere Auswirkungen hat ein Fehler und desto wichtiger sollte seine Bearbeitung sein.

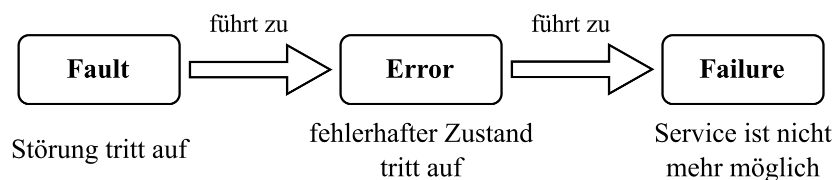


Abbildung 6.1: Der Zusammenhang zwischen *Fault*, *Error* und *Failure* [RAB19].

Der Zusammenhang von Störungen und Ausfällen lässt sich unter anderem durch Fault Trees darstellen. Fault Trees nach [RS15] verwenden dabei logische Operatoren, um das Zusammenwirken von Störungen und Fehler in einer Baumstruktur zu strukturieren. So kann nachvollzogen werden, in welcher Kombination Störungen und Fehler zum Ausfall des Systems führen.

6.1 Fehleranalyse der PG

Um Fehler gezielt injizieren und anschließend behandeln zu können, muss das System zunächst auf potenzielle Fehler analysiert und diese klar definiert werden. Deshalb wurde zu Beginn des Projekts eine umfassende Fehlerliste erstellt. Dazu wurden Fehlerbäume, die Hazard and Operability Study (HAZOP) und eine Failure Mode and Effects-Analyse (FMEA) betrachtet. Im weiteren Verlauf des Abschnittes wird auf die Vor- und Nachteile und die Verwendung der oben aufgeführten Methoden in der Projektgruppe eingegangen.

Fehlerbäume geben an, welche Fehler andere Fehler verursachen und können so die Erkenntnis liefern, dass bestimmte Fehler nicht behandelt werden müssen, wenn ein Fehler, der diese hervorruft bereits behandelt wird. Somit könnte der Aufbau von Resilienz und die Behandlung der Fehler auf einen Bruchteil der Fehler fokussiert werden und die Arbeit der PG ließe sich zeiteffizienter gestalten. Beim Versuch Fehlerbäume zu erstellen, wurde jedoch deutlich, dass diese nicht für JARVIC geeignet sind, da die einzelnen Fehler jeweils ohne Interaktion mit anderen Fehlern direkt zum Ausfall des Systems führen, was das Nutzen des Vorteiles eines Fehlerbaumes, die Darstellung des Zusammenwirkens von Fehlern, verhindert. Somit wurden Fehlerbäume verworfen.

Mithilfe der HAZOP lässt sich eine erschöpfende Liste von möglichen Fehlern erstellen, wozu jedoch eine Kenntnis von den möglicherweise betroffenen Komponenten und sinnvollen Leitwörtern, deren Kombinationen die potentiellen Fehler bilden, vorausgesetzt ist. Die resultierende Liste würde Kombinationen enthalten, die keine sinnvollen Fehler darstellen. Diese Fehler müssten aus der Liste entfernt werden, bevor eine sinnvolle Liste übrig bliebe. Diese Fehler könnten ebenfalls nur mit ausreichendem Wissen von Jarvic erkannt und entfernt werden. Zu dem Zeitpunkt, an welchem eine Liste erstellt wurde, war der für das Erstellen und Bereinigen der Liste nötige Kenntnisstand noch nicht erreicht, weshalb eine HAZOP als Methode verworfen wurde.

Der Fokus wurde auf die Identifikation realistischer Fehlerquellen gerichtet. Es wurden dabei verschiedene Ansätze verfolgt, um zu spezifizieren, welche Fehler realistisch sind: Die Analyse von Rosbags und alten Videos von fehlgeschlagenen Fahrten bot wertvolle Einblicke in konkrete Fehlerszenarien. Eine systematische Betrachtung der Komponenten von JARVIC (u. a. anhand von Strukturdiagrammen für Messages, Nodes und Kommunikationsstrukturen) war dazu angedacht, die Fehlerabhängigkeiten innerhalb des Systems besser zu verstehen.

Die erstellte Fehlerliste wurde mit GET racing besprochen und ergänzt. Gemeinsam mit GET wurden die Prioritäten für die Fehler festgelegt. Das Ziel war, dass die Fehler in der Reihenfolge von hoher zu niedriger Priorität bearbeitet werden können. Es wurde auf der Liste eine Abwandlung der Failure Mode and Effects Analyse nach [Boz+10] durchgeführt. Die Prioritäten basierten hauptsächlich auf den potenziellen Auswirkungen auf das Gesamtsystem und der Häufigkeit des Auftretens. Diese wurden anders als bei FMEA nicht multipliziert, sondern lediglich addiert, da dies intuitiver erschien. Außerdem sollten die Kategorien nicht unterschiedliche gewichtet werden. Dafür wäre eine Multiplikation notwendig gewesen.

6.2 Resultat der Analyse

Bei der Auswahl der Prioritäten stellten sich zwei Kategorien heraus: Fehler, bei denen die Fahrt fortgesetzt werden kann (sogenannte wiederherstellbare Fehler) sowie Fehler, bei denen die bestmögliche Behandlung der Notstopp des Fahrzeugs ist (katastrophale Fehler). Die Fehlerliste wurde in wiederherstellbare und katastrophale Fehler mithilfe von Einschätzungen durch GET racing eingeteilt. Dadurch kann die Entwicklung der Fehlerbehandlung auf ein Vorgehen spezifiziert werden. Bei beiden Kategorien ist es sehr relevant, dass frühzeitig erkannt wird, dass ein Fehler vorliegt. Nur bei den katastrophalen Fehlerfällen kann daraufhin unmittelbar eine Bremsung eingeleitet werden. Im Falle von wiederherstellbaren Fehlern muss das Auto angepasst auf den Fehlerfall fahren und eine Möglichkeit finden auf diesen Fehler zu reagieren.

Zusätzlich wurden die Fehler danach eingeteilt, ob und wie sie injiziert werden können. Da zum Injizieren von Fehlern in der Perception die Kamera simuliert werden musste, ließen sich diese Fehler nur in FSDS injizieren und konnten erst im späteren Verlauf der PG bearbeitet werden. Fehler, durch welche Leitkegel an falschen Positionen oder nicht erkannt werden, können auch in anderen Simulatoren durch das Bearbeiten der Karte injiziert werden.

In Tabelle 6.1 wird ein Teil der identifizierten Fehler abgebildet. Jeder Fehler hat einen Prioritätswert, zusammengesetzt aus vier Werten. Diese Werte sind: Behandelbarkeit, Relevanz, Auftrittswahrscheinlichkeit und Injizierbarkeit. In den restlichen Spalten wird angegeben, ob sich die jeweiligen Fehler mit den jeweiligen Methoden injizieren lassen. Die Tabelle ist in drei Abschnitte geteilt. Während der erste und zweite Teil die einfach und mittelschwer behandelbaren Fehler auflistet, zeigt der letzte Teil die Fehler auf, bei denen ein Notstopp nötig ist. Die einzelnen Abschnitte sind nach Priorität sortiert.

Tabelle 6.1: Die wichtigsten Fehler gruppiert nach Schwierigkeitsgrad der Injektion. Injizierbar beschreibt dabei, ob die Fehler mit dem aktuellen Technikstand injiziert werden können.

Fehler	**Injizierbar**	**Injektion über Track Modifier**	**Injizierbar nur mit FSDS**	**Priorität**
Einfach behandelbar				
Leitkegelfarbe falsch erkannt	Ja	Ja	Ja	275
Verfälschung des Kamerabilds mit Dauer <300ms	Nein	Nein	Ja	250
Gemessener Lenkwinkel ≠ angesteuerter Lenkwinkel	Ja	Nein	Nein	250
Ausfall einer Kamera mit Dauer <300ms	Nein	Nein	Ja	230
Verlangsamung durch Berechnungsburst	Nein	Nein	Nein	210
Nichterkennen einiger Leitkegel	Ja	Ja	Nein	205
Temporärer Ausfall des Lenkwinkelsensors (Funktioniert zu Beginn des Rennens)	Ja	Ja	Nein	195
Erkennen von nicht existenten Leitkegeln (Korrekte Farbe für die Seite)	Ja	Ja	Nein	160
Pixelausfälle im oberen Kamerabild	Nein	Nein	Ja	160
Fehler in der Synchronisation von Nodes	Nein	Nein	Nein	160
Leitkegel physisch auf die Mitte der Fahrbahn geschleudert	Ja	Ja	Nein	145
Leicht verdrehte Kamera relativ zum Auto	Nein	Nein	Ja	135
Mittelschwer behandelbar				
Störung in Lidar Punktewolke	Nein	Nein	Ja	230
Temporärer Ausfall der Odometrie	Nein	Ja	Nein	225
Pixelausfälle im unteren Kamerabild	Nein	Nein	Ja	210
Absturz von Teilen des ROS-Netzwerks <1s	Nein	Nein	Nein	205
Trajektorie ändert sich stark zwischen zwei Bildern	Nein	Nein	Nein	190

Fehler	Injizierbar	Injektion über Track Modifier	Injizierbar nur mit FSDS	Priorität
Erkennen von nicht existenten Leitkegeln (Falsche Farbe für die Seite)	Ja	Ja	Nein	155
Rauschen in der IMU	Nein	Nein	Nein	120
Rauschen des Lenkwinkelsensors	Nein	Nein	Nein	120
Rauschen in der Odometrie	Nein	Nein	Nein	65
Notstopp nötig				
Auto mechanisch lenkunfähig	Nein	Nein	Nein	260
Ausfall einer Kamera mit Dauer >300ms	Nein	Nein	Ja	255
Rauschen der Kamera mit Dauer >300ms	Nein	Nein	Ja	250
Permanenter Ausfall des Lenkwinkelsensors	Ja	Ja	Nein	230
Ausfall des Lidars	Nein	Nein	Ja	205
Temporärer Ausfall des Lenkwinkelsensors (Funktioniert nicht zu Beginn des Rennens)	Ja	Ja	Nein	190
Ausfall beider Kameras	Nein	Nein	Ja	180
Absturz von Teilen des ROS-Netzwerks >1s	Nein	Nein	Nein	175
Leitkegel längerfristig nicht erkannt	Nein	Nein	Ja	160

Durch die in der Liste gegebene Priorisierung sollte eine Reihenfolge erarbeitet sein, in welcher die Fehler im Rahmen der PG bearbeitet werden. Nach dem Bearbeiten von Fehlern bot es sich jedoch an, erst vergleichbare Fehler zu bearbeiten, da Werkzeuge, die zur Injektion oder Absicherung eines Fehlers entwickelt wurden, ohne größeren Aufwand auf vergleichbare Fehler angepasst werden konnten. So ergab sich die Reihenfolge, in welcher die Fehler bearbeitet wurden, aus den zum jeweiligen Zeitpunkt zur Verfügung stehenden Werkzeugen und die durch die Liste gegebene Priorisierung wurde verworfen.

Kapitel 7

Fehlerinjektion

Fehlerinjektion ist eine Methode zum Testen von Resilienzmaßnahmen durch vorsätzliches Einbringen von Fehlern in das System [Sal+22]. Dadurch lassen sich die Auswirkungen der Fehler beobachten und die Qualität der Resilienzmaßnahmen überprüfen. Allgemeine Grundlagen zum Themenbereich der Fehlerinjektion werden im Folgenden zuerst in Abschnitt 7.1 vorgestellt. Anhand der Fehleranalyse in Abschnitt 6.1 wurden verschiedene Arten von möglichen Fehlern identifiziert. Anfängliches Ziel war es daher, verschiedene Methoden zu finden, um diese Fehler auch in JARVIC injizierbar zu machen. Im Rahmen der Projektgruppe wurden dabei folgende Möglichkeiten betrachtet, deren Prinzipien und Umsetzungen in den darauf folgenden Abschnitten vorgestellt werden:

Injektion auf Nachrichtenebene in ROS Dies ist die Hauptmethode, welche in der Projektgruppe umgesetzt wurde. Diese wird in Abschnitt 7.2 vorgestellt.

Injektion von Fehlern der Map Hierfür wurde ein Track Modifier erstellt. Dieser wird in Abschnitt 7.3 vorgestellt.

Injektion von Fehler vor der Perception Bestimmt Fehler, wie beispielsweise realistische Regentropfen auf dem Kamerabild, müssen bereits in der grafischen Simulation injiziert werden. Hierfür ist eine Schnittstelle zum Simulator notwendig. Für FSDS wurde eine Änderung des Wetters implementiert, welche in Abschnitt 7.4 vorgestellt wird.

7.1 Grundlagen der Fehlerinjektion

Ziel der Fehlerinjektion ist es, das Verhalten eines Systems unter auftretenden Fehlern zu untersuchen. Dafür muss eine Fehlerinjektion entwickelt werden, welche Fehler in ein System injizieren kann. Eine konzeptuelle Beispielarchitektur für eine solche Fehlerinjektion nach [NCM16] ist in Abbildung 7.1 dargestellt. Dort verteilt eine Steuerungseinheit (Controller) Befehle an die Komponenten, welche die Fehlerinjektion durchführen. Eine Komponente erzeugt Eingaben für das zu testende System (Load Generator). Der Monitor sammelt alle

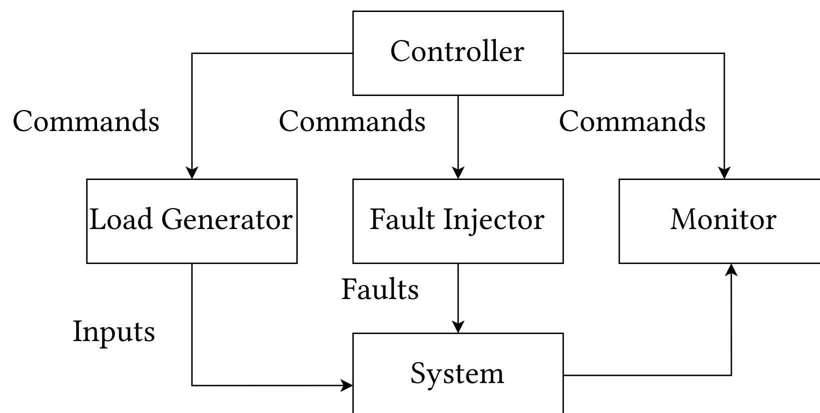


Abbildung 7.1: Beispielhafte Architektur der Fehlerinjektion. Fehlerinjektion braucht für Ausführung unterschiedliche Komponenten, die mit dem System interagieren (Load Generator, Fault Injector) und Daten für die Analyse sammeln (Monitor). Der ganze Prozess wird vom Controller gesteuert (nach [NCM16]).

relevanten Daten für die Analyse des Systemverhaltens. Der Fehlerinjektor (Fault Injector) injiziert den Fehler in das System. Das System muss dabei nicht in der Realität ausgeführt. Die Fehlerinjektion ist auch mit Simulation oder Emulation des Systems möglich, wie zum Beispiel bei JARVIC mit FSSIM.

Fehler können auf unterschiedlichen Ebenen des Systems auftreten. Für die Entwicklung der Fehlerinjektion sollte den Entwickler:innen daher klar sein, welche Art von Fehlern ins System gebracht werden soll (z. B. Timeout, Paketverlust), wann (z. B. bei jedem Speicherzugriff, zufällig) und wo (z. B. in den Speicher, Quellcode) [NCM16]. Je nach Art sind ggf. unterschiedliche Techniken für die Fehlerinjektion notwendig.

7.1.1 Techniken der Fehlerinjektion

Bei den Techniken wird unter anderem zwischen der hardwarebasierten und softwarebasierten Fehlerinjektion unterschieden.

Hardwarebasierte Fehlerinjektion arbeitet direkt mit Schaltungen und kann z. B. Überspannung oder Bit-Flips injizieren. Einige wesentliche Vorteile dieses Ansatzes sind, dass es im System keine unerreichbaren Stellen gibt, die analysiert werden sollen und auch die Zeitauflösung ist hoch, da man direkt mit der Hardware arbeitet [HTI97; Sal+22]. Hardwarebasierte Fehlerinjektion war früher relevanter, wird aber heute durch Softwarefehlerinjektion ersetzt, da heutige Software viel komplexer ist [NCM16]. Da sich die Fehler aus der Fehleranalyse der PG hauptsächlich auf Softwarefehler beziehen, wird dieser Ansatz für die Fehlerinjektion für JARVIC hier nicht verwendet.

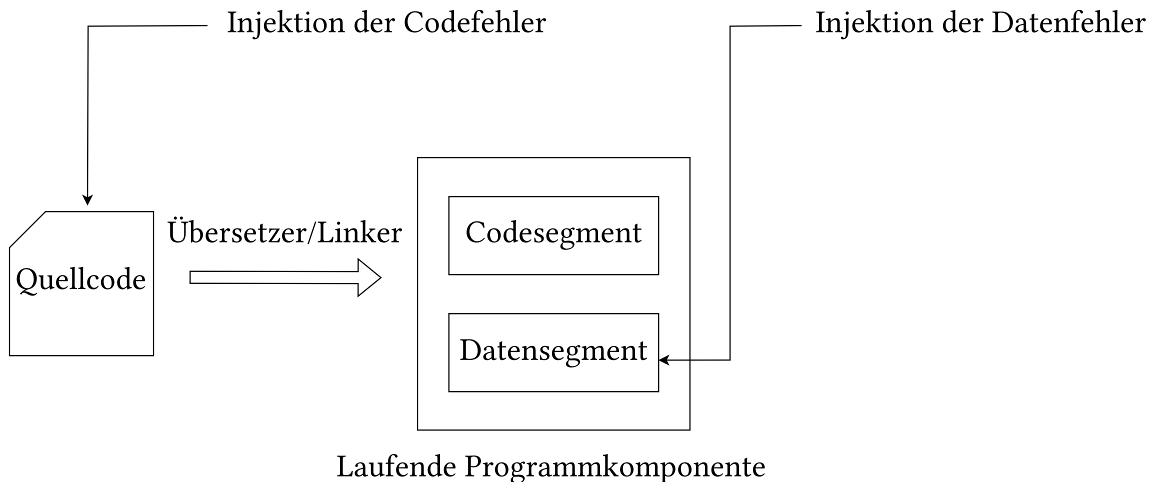
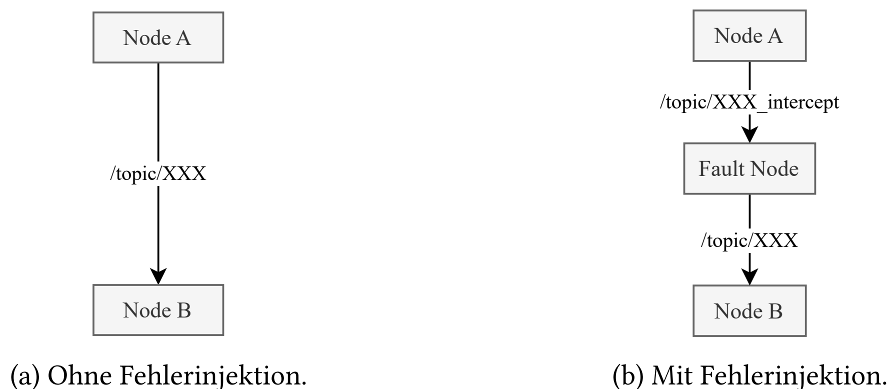


Abbildung 7.2: Softwarebasierte Injektion stützt sich auf Code- und Datenänderungen, wobei Datenänderungen auch zur Laufzeit möglich sind (nach [NCM16]).

Bei der softwarebasierten Fehlerinjektion werden die Fehler direkt in die Software injiziert. Softwarefehler werden insbesondere bei großen Softwaresystemen untersucht [NCM16]. Allgemeine Ansätze der Softwarefehlerinjektion sind in Abbildung 7.2 dargestellt. Faults können schon während der Übersetzungsphase eingebettet werden. Codefehler stellen ein mögliches Beispiel dar. Softwarefehlerinjektion wird weiter auf Compile-time und Runtime-Fehlerinjektion aufgeteilt. Während Compile-time Fehlerinjektion durch Codeänderungen repräsentiert wird, wird Runtime-Fehlerinjektion z. B. durch Exceptions, Timer usw. implementiert, um die Kontrolle an den Fault Injector an der richtigen Stelle zu übergeben [HTI97]. Somit können Datenfehler erzeugt werden, die zu dem richtigen Zeitpunkt oder unter bestimmten Bedingungen injiziert werden können [NCM16; Sal+22]. Mit diesem Ansatz können Betriebssysteme und Software getestet werden, wo Hardwarefaults nicht injizierbar sind. Der Ansatz ist auch auf neue Software erweiterbar, was für Hardwarefaults nicht der Fall ist [HTI97].

7.2 Fehlerinjektion auf Basis von ROS-Nachrichten

In diesem Abschnitt wird ein möglicher Ansatz für die softwarebasierte Fehlerinjektion in JARVIC auf Node-Ebene mittels ROS-Nachrichten vorgestellt. Die Grundidee ist, dass ein zusätzlicher ROS-Node auf dem Kommunikationspfad zwischen zwei oder mehreren Node integriert wird, der die Nachrichten abfängt, sie auf bestimmter Weise bearbeitet und weiterleitet. Deswegen kann der Ansatz auch als Man-In-The-Middle-Node bezeichnet werden. Durch diese Architektur kann somit vorgetäuscht werden, dass der Absender einer Nachricht,



(a) Ohne Fehlerinjektion. (b) Mit Fehlerinjektion.

Abbildung 7.3: Architektur des Ansatzes für Fehlerinjektion auf Basis von ROS-Nachrichten.

aus beliebigen Gründen, fehlerhafte Daten versendet hat. Hierdurch wird eine softwarebasierte Runtime-Fehlerinjektion auf Basis von Datenfehlern durchgeführt (vgl. Abbildung 7.2).

Anhand des Beispiels in Abbildung 7.3 wird die Architektur im Folgenden kurz erläutert. Ohne Fehlerinjektion würde Node A mit Node B über das Topic `/topic/XXX` kommunizieren (vgl. Abbildung 7.3a). Dabei ist Node A ein Publisher und Node B ein Subscriber. Für die Fehlerinjektion wird zwischen beiden Nodes ein Fault-Node installiert (vgl. Abbildung 7.3b). Dieser soll die von Node A gesendeten Nachrichten ändern. Damit die von Node A versendeten Nachrichten abgefangen und manipuliert werden können und diese nicht direkt von Node B empfangen werden, muss entweder Node A ein anderes Topic für die Veröffentlichung der Nachricht verwenden oder Node B ein anderes Topic abonnieren. In diesem Fall verwendet Node A ein Topic mit dem Postfix `_intercept`. Der Inhalt der Nachrichten kann somit verändert werden, ohne dass die eigentlichen Nodes dies bemerken. Dies hat den Vorteil, dass die Fehlerinjektion ohne Änderungen des Quellcodes von JARVIC auskommt und somit die Entwicklung von den JARVIC-Hauptkomponenten unabhängig von der Fehlerinjektion bleibt. Es kann eine beliebige Anzahl von Node A oder Node B geben, solange diese dasselbe Topic zur Kommunikation nutzen.

Die vollständige Architektur dieser Art der Fehlerinjektion ist in Abbildung 7.4 dargestellt. Zusätzlich zu den vorgestellten Fault-Nodes existiert noch eine zentrale Steuerung der Fault-Nodes mittels sogenannten Control-Node. Verglichen mit der Beispielarchitektur aus Abbildung 7.1 übernimmt dieser Control-Node die Rolle des Controllers. Die einzelnen Fault-Nodes stellen den Fault Injector dar, welche Fehler in das System (hier JARVIC) injizieren. Ein expliziter Load Generator existiert in dieser Architektur nicht, da für JARVIC die Eingaben bereits automatisch aus der Simulation stammen. Der Monitor entspricht allen Komponenten, welche die Simulationsdaten auswerten, wie z. B. eine RBB-Auswertung in der CI.

Im Folgenden werden zuerst die Überlegungen und Implementierungsdetails zu den Fault-Nodes vorgestellt. Anschließend wird die Implementierung und Konfiguration des Control-Nodes beschrieben. Darauf folgend werden verschiedene Methoden zum Starten des Systems

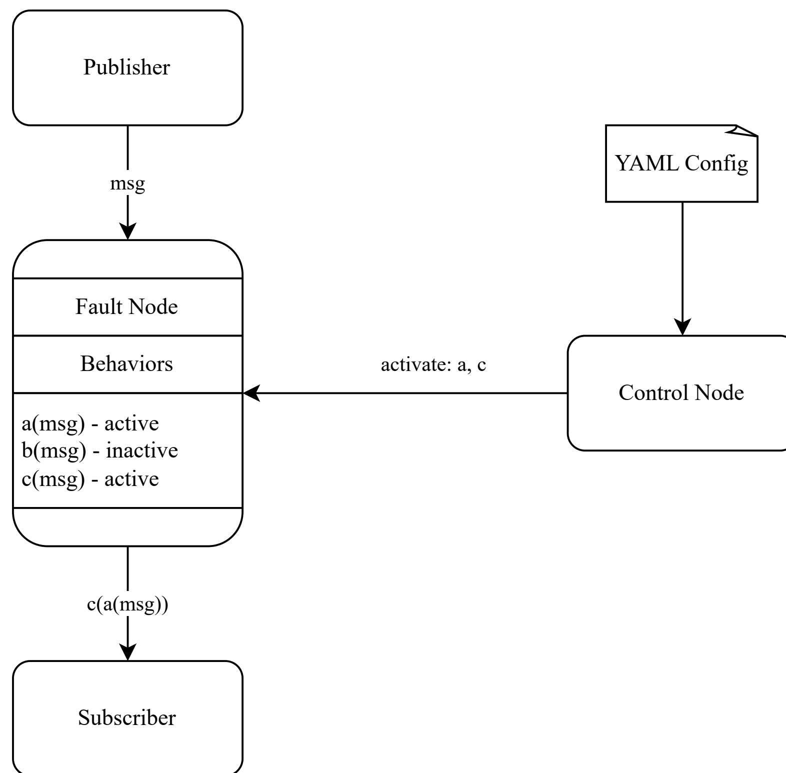


Abbildung 7.4: Gesamtarchitektur der Fehlerinjektion mit Fault-Node, Control-Node und dem Beispiel einer modifizierten Nachricht mit zwei Behaviors.

mit aktivierter Fehlerinjektion vorgestellt und weitere Aspekte diskutiert, die es dabei zu beachten gibt. Abschließend werden die bereits mit dieser Methode implementierten Fehler vorgestellt.

7.2.1 Implementierung von Fault-Nodes

Es wurde ein Framework in C++ entwickelt, mit welchem Fault-Nodes möglichst einfach implementiert werden können. Das Framework soll dabei die Implementierungsdetails, wie genau das Abfangen der Nachricht und weitere für ROS notwendige Schritte ablaufen, zusammenfassen. Somit wird Code-Duplikation von Code, der für alle Fault-Nodes gleich ist, vermieden und die Usability vereinfacht. Ein Beispiel für ansonsten doppelt benötigten Code ist das Erstellen von ROS-Subscribern und Publishern.

Als Eingabe benötigt das Framework zum einen den Namen des Topics, auf welchem Nachrichten abgefangen werden soll. Der Nutzer kann zudem sogenannte Behaviors hinzufügen. Ein Behavior besteht aus einem Namen sowie aus einer Funktion, welche die eingehenden Nachrichten erhält und die (ggf. veränderten) Nachrichten zurückgibt. In der Funktion kann

ein beliebiges Verhalten implementiert werden, welches die Nachricht verändert oder bspw. auch unverändert zurückgibt. Das Framework implementiert damit eine Art des Strategie-Entwurfsmusters. Für viele Behaviors kann es sinnvoll sein, Parameter zu verwenden. So könnte beispielsweise ein Behavior, welches Rauschen auf einen Wert hinzufügt, Parameter zur verwendeten Zufallsverteilung benutzen. Um die Behaviors möglichst wiederverwendbar zu halten, sollte jedoch nicht für jede verschiedene Parameterkombination ein eigenes Behavior implementiert werden müssen. Deshalb wird der Funktion des Behaviors zudem noch eine Key-Value Map übergeben, welche behaviorspezifische Parameter beinhalten kann.

Das Framework verwaltet eine Liste an aktuell aktiven Fehlern bzw. Behaviors. Die aktiven Behaviors werden der Reihe nach auf die eingehenden Nachrichten angewandt. Somit ist auch eine Verkettung von verschiedenen Fehlern auf nur eine Nachricht möglich. Beispielsweise könnte bei der Nachricht an den Lenkaktor zuerst ein fester Offsetwinkel addiert werden und zusätzlich ein Rauschen hinzugefügt werden. Da die Fehler nach einer fest sortierten Reihenfolge angewandt werden, können ebenfalls Fälle abgebildet werden, bei denen die Anwendung der Behaviors nicht kommutativ wäre. Durch diese Kombinierbarkeit von Behaviors können komplexere Fehler aus kleineren Behaviors zusammengestellt werden. Wie die Liste der aktuellen Behaviors gepflegt wird, wird in Unterabschnitt 7.2.2 beschrieben.

Hinsichtlich der Implementierung besteht das Framework aus einer Template-Klasse, welche die Logik für das Abfangen und Weiterleiten der Nachricht kapselt. Ein vereinfachtes UML-Diagramm der Implementierung ist in Abbildung 7.5 dargestellt. Der Templateparameter `MsgType` der Frameworkklasse gibt den Typen der ROS-Nachricht an, die abgefangen werden soll. Der Fault-Node als Nutzer kann der Instanziierung des Frameworks Behaviors durch die Funktion `setBehavior` hinzufügen. Das Behavior verwendet implementierungstechnisch eine `std::function` und kann so z. B. vom Nutzer durch ein Lambda implementiert werden. Damit auch das Verlorengehen einer Nachricht simuliert werden kann, muss die Funktion des Behaviors ein `std::optional<MsgType>` zurückgeben. `std::optional` ist ein Typ der C++-Standardbibliothek, welcher entweder einen Wert oder einen leeren Wert enthalten kann. Durch das Zurückgeben des leeren optionalen Werts im Behavior wird erreicht, dass die Nachricht nicht weitergeleitet wird.

Für die größtmögliche Flexibilität der möglichen Parameter, welche ein Behavior als Eingabe erhalten kann, ist die Key-Value Map als Map von Strings auf ein `std::variant` von ints, floats, Strings oder booleans implementiert. `std::variant` ist ein Typ der C++-Standardbibliothek, welcher einen Summentyp implementiert. Wie das Setzen der aktiven Behavior-Parameter abläuft, wird in Abschnitt Unterabschnitt 7.2.2 beschrieben.

Eine Beispielimplementierung eines Fault-Nodes ist in Listing 7.1 dargestellt. Nach der normalen ROS-Initialisierung in Zeile 3 wird das Objekt des Fehlerinjektionsframeworks in Zeile 5 erstellt. Als Templateparameter wird hier `ConeDetections` als Nachrichtentype von FSSIM übergeben. Dieser enthält eine Liste an (simuliert) erkannten Leitkegel, wobei jeder Leitkegel wiederum die Wahrscheinlichkeiten enthält zu einer bestimmten Farbklasse zu gehören. Ebenfalls wird das Topic übergeben, auf welchem die Nachrichten weitergeleitet

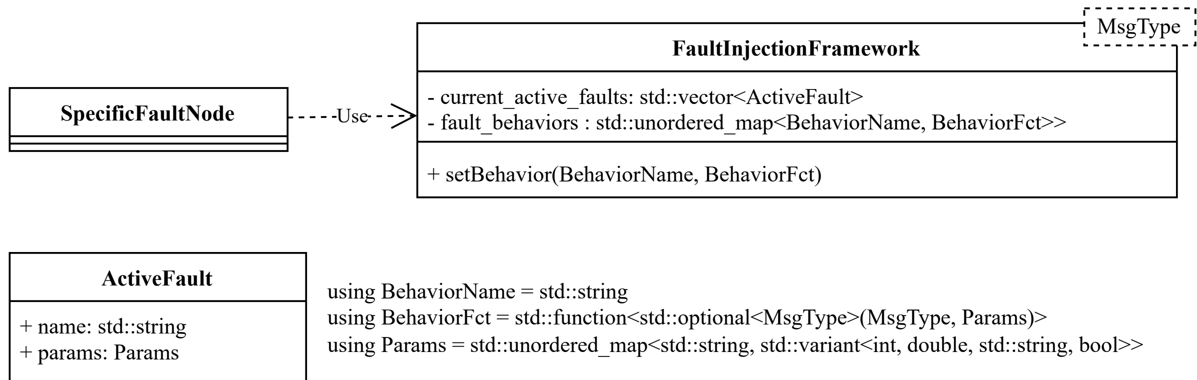


Abbildung 7.5: Vereinfachtes UML-Diagramm der Implementierung der Fault-Nodes.

werden sollen. Der letzte Parameter des Konstruktors ist für die Steuerung des Frameworks von außen notwendig und wird später genauer beschrieben. Anschließend wird in den Zeilen 7-21 das Behavior „changeColor“ hinzugefügt. Die Behaviorfunktion kann direkt als Lambda übergeben werden, welche hier alle Leitkegel der gelben Klasse hinzufügt. Anschließend kann ein normaler ROS-Loop ausgeführt werden. Alles andere, wie das Erstellen des Subscribers und Publishers und das Verarbeiten der Nachrichten, wird durch das Framework automatisch im Hintergrund erledigt.

Durch die Implementierung können Fehlverhalten schnell implementiert werden. Wenn ein Fehlverhalten beispielsweise einen Wert einer Nachricht auf einen fixen Wert setzen soll, ist dies einfach in der Funktion des Verhaltens möglich. Ein Nachteil der Implementierung ist jedoch, dass bei komplexeren Fehlverhalten, die z. B. vom Nachrichteninhalt vorhergehender Nachrichten abhängen, also einen globalen Zustand benötigen, die Implementierung komplexer ist. In diesem Fall muss jeder spezifische Fault-Node selbst den Zustand verwalten. Das Framework bietet für jedes hinzugefügte Verhalten noch die Hilfsfunktion an, die aufgerufen wird, wenn das Verhalten deaktiviert wird. Dies kann dazu verwendet werden, um den gespeicherten Zustand zurückzusetzen. Dennoch muss je nach gewünschtem Fehlverhalten der Zustand unterschiedlich gespeichert und behandelt werden.

7.2.2 Zentrale Steuerung durch Control-Node

Mit der vorgestellten Man-In-The-Middle-Node-Architektur können die Nachrichten abgefangen und bearbeitet werden. Die Faults sollen aber noch ins System injiziert und aktiviert werden. Dazu wird zusätzlich zu den Fault-Nodes ein spezialisierter ROS-Control-Node gestartet. Der Control-Node steuert zeitliches Verhalten von Faults und aktiviert sie zum richtigen Zeitpunkt. Dazu wird eine sogenannte Activation-Nachricht über ein ROS-Topic an den entsprechenden Fault-Node verschickt. Die Activation-Nachricht enthält die Liste von Behaviors, die im entsprechenden Fault-Node aktiviert werden sollen. Zur Vereinfachung geschieht die

```
1  int main(int argc, char **argv)
2  {
3      ros::NodeHandle nodeHandle("~");
4
5      FaultInjectionFrameworkHandle<ConeDetections> handle(nodeHandle, "/
          fssim/cone_detections/camera", activationTopic);
6
7      handle.setBehavior("changeColor", [](const ConeDetections &msg, const
           FaultBehaviorAdditionalParams &params)
8          {
9              using Cone = ConeWithCertainty;
10             // Return modified message to simulate a fault
11             // just for demonstration purposes
12             auto msg_copy = msg;
13             for(auto &cone : msg_copy.cones) {
14                 cone.classCertainty[CLASS_ID_YELLOW] = 1.0;
15                 cone.classCertainty[CLASS_ID_BLUE] = 0.0;
16                 cone.classCertainty[CLASS_ID_ORANGE_SMALL] = 0.0;
17                 cone.classCertainty[CLASS_ID_ORANGE_BIG] = 0.0;
18                 cone.classCertainty[CLASS_ID_UNKNOWN] = 0.0;
19             }
20             return msg_copy;
21         });
22
23     // Run default ROS loop
24 }
```

Listing 7.1: Beispielcode eines Fault-Nodes, um die Farben von erkannten Leitkegeln aus FSSIM zu modifizieren.

```
string key
int32[] int_value
float64[] double_value
string[] string_value
bool[] boolean_value
```

Listing 7.2: Elementstruktur der ROS-Activation-Nachricht für einen einzelnen zusätzlichen Parameter.

```
faults: [
  {
    type: "control",
    timing: ["once", {time_start: 15., time_end: 30.}], # in sec
    behavior: ["ENGINE_DAMAGE"],
    additional: {"partial_damage": 0.9},
  }
]
```

Listing 7.3: Beispiel einer Konfiguration für die Simulation eines Motorschadens.

Übertragung der aktiven Behaviors zustandslos, sodass beim Empfangen einer Activation-Nachricht im Fault-Node alle bislang aktiven Behaviors deaktiviert werden und nur die in der neuen Activation Nachricht gelisteten Behaviors aktiv geschaltet werden. Somit ist keine aufwändige und fehleranfällige Zustandslogik notwendig.

Die Activation-Nachricht enthält zudem je aktiven Behavior die Map an Parametern, die dem Behavior bei der Ausführung mitgegeben werden sollen. Die Parameter vom Fault-Node bestehen aus einer C++-Map mit einem `std::variant`. Die Activation-Nachricht ist jedoch eine ROS-Nachricht, welche nicht nativ, aufgrund von Sprachunabhängigkeit, C++-Standardbibliothekstypen unterstützt. Daher musste die Übersetzung in von ROS unterstützte Datentypen (Array, Integer, Float, ...) selbst implementiert werden. Die Map wurde dabei als ein Array implementiert, welches Elemente enthält, die jeweils ein Key und Value enthalten (vgl. Listing 7.2). Für das Variant der Values wurden jeweils Arrays der entsprechenden Typen verwendet. Diese Implementierung konnte sowohl für ROS1 als auch ROS2 verwendet werden.

Der Control-Node wird durch eine YAML-Datei konfiguriert. Diese enthält in JSON-Format die Konfiguration, nach welcher bestimmte Behaviors in verschiedenen Nodes aktiviert werden sollen. Ein Beispiel einer solchen Konfigurationsdatei ist in Listing 7.3 dargestellt, bei welcher ein 90-prozentiger Motorschaden zwischen 15 und 30 Sekunden nach Start der Simulation simuliert wird.

Im Allgemeinen wird durch den `type` der Name des Activation-Topic und somit auch implizit der Fault-Nodes bestimmt, welcher bei der Aktivierung des Faults angesprochen wird.

```
1 <remap from="/different_topic" to="/needed_topic"/>
```

Listing 7.4: Roslaunch remap-Tag.

`behavior` gibt den Namen des Behaviors an, welches mit den Parametern aus `additional` aktiviert werden soll. Für das Zeitverhalten gibt es drei Arten: Bei einem einmaligen Fault werden Start- und Endzeit benötigt; bei einem periodischen Fault wird neben der Startzeit auch die Dauer der Periode benötigt. Die Periode legt fest, dass der nächste Fehler mit ihrem Ende beginnt und das aktuelle Fehlerende zur Halbperiode bestimmt wird.

Der zentrale Steuerungs-Node fügt der Fehlerinjektion einen Single-Point-Of-Failure hinzu. Ebenfalls erzeugt der gewählte Ansatz Komplexität, da ein weiterer Nachrichtenaustausch zwischen Steuerungs-Node und Fault-Nodes notwendig ist. Ein anderer möglicher Ansatz dies zu vermeiden wäre, dass die Konfiguration der Fehler (vgl. Listing 7.3) durch jeden einzelnen Fault-Node eingelesen wird und nur die diesen Node betreffende Fehler automatisch aktiviert werden. Hierdurch wäre jedoch eine zeitliche Synchronisation aller Fault-Nodes notwendig gewesen, damit das Timing der Faults korrekt ist. Mit dem zentralen Steuerungs-Node besitzt dieser automatisch das globale Timing. Der Steuerungs-Node wird anfänglich mit einem simulatorspezifischen Signal synchronisiert, damit die Fehler entsprechend der Zeit nach Beginn der Simulation und nicht nach Start des Steuerungs-Nodes triggern. Ebenfalls erlaubt das nachrichtenbasierte Interface der Fault-Nodes zur Aktivierung auch manuell Faults im laufenden System zu triggern (bspw. durch Senden einer Aktivierungsnachricht über die Kommandozeile). Dies kann zum Testen der implementierten Fehlverhalten hilfreich sein. Aus diesen Gründen wurde sich für den zentralen Ansatz mit Steuerungs-Node entschieden.

7.2.3 Starten des Systems mit Fehlerinjektion

Integraler Bestandteil der hier vorgestellten Fehlerinjektion ist, dass ein Node auf einem Topic Nachrichten veröffentlicht, welche vom Man-in-the-Middle Node empfangen werden. In Abbildung 7.3b wurde dies einfach umgesetzt, indem der Publisher bei aktiver Fehlerinjektion auf einem anderen Topic publisht, als wenn das System ohne Fehlerinjektion läuft. Da aber eines der Hauptziele dieser Fehlerinjektionsmethode ist, dass keine Quellcodeänderungen an JARVIC notwendig sind, sollte der Publisher diese Änderung im Idealfall gar nicht bemerken. Hierzu kann sogenanntes Remapping verwendet werden.

Die hierarische Struktur von JARVIC-Launchdateien kann auch für Fehlerinjektion genutzt werden. Roslaunch verfügt über einen Tag, das ROS-Nachrichten umleiten kann, nämlich `<remap>` (vgl. Listing 7.4). Dieser Tag kann direkt in den Node-Tag eingesetzt werden. Somit werden alle Nachrichten, die dieser Node auf den Topic publisht, der im remap-Tag unter `from` steht, auf den Topic unter `to` gepublisht.

Die remap-Tags müssen den Launchdateien hinzugefügt werden, sollten jedoch auch nur ausgeführt werden, wenn JARVIC mit der Fehlerinjektion ausgeführt werden soll. Da Launchdateien nur am Programmstart gelesen werden, müssen die Tags vor dem Programmstart in den Launchdateien enthalten sein. Da die Fehlerinjektion mit separaten Fault-Nodes und einem Control-Node durchgeführt wird, müssen die Launchdateien für diese Nodes auch inkludiert werden, damit diese beim Start von JARVIC mit gestartet werden. Zwei mögliche Ansätze, um diese Ziele zu erreichen wurden untersucht:

- Alle notwendigen Tags werden vorab eingefügt und je nach Bedingung eingeschaltet (z. B. durch Umgebungsvariablen).
- Die Launchdateien werden durch ein Skript dynamisch direkt vor dem Ausführen geparsed, bis der passende Node-Tag gefunden wird. Dann wird ein remap-Tag zusammen mit einem include-Tag für den entsprechenden Fault-Node hinzugefügt. Nach dem Ende der Ausführung können die injizierten Tags gelöscht werden. JARVIC bleibt unverändert.

Beide Ansätze haben Vor- und Nachteile. Die erste Methode ist wesentlich einfacher zu implementieren: In diesem Fall sollen die Tags direkt in JARVIC-Launchdateien eingefügt werden. Zudem wird nur durch die Ausführung der Fehlerinjektion der Code im Repository nicht verändert, sodass eine in den Launchdateien aktivierte Fehlerinjektion nicht versehentlich commitet werden kann. Allerdings müssen dafür die Launchdateien von JARVIC bearbeitet werden, sodass JARVIC nicht mehr komplett unabhängig von der Fehlerinjektion ist. Daneben werden bei einer sehr einfachen Implementierung immer alle Fault-Nodes gestartet, auch wenn in einem Fehlerszenario diese vielleicht überhaupt nicht benötigt werden. Im zweiten Fall ist die Situation umgekehrt: Es können nur die Fault-Nodes gestartet werden, welche auch tatsächlich für ein Szenario nötig sind. Zudem sind keine dauerhaften Änderungen in den JARVIC Launchdateien notwendig. Allerdings ist das Parsen der Launchdateien und das Handling von Launchdateien, welche von vorherigen Durchläufen ggf. schon eine teilweise aktivierte Fehlerinjektion beinhalten, komplexer. Beide Ansätze wurden im Rahmen der Projektgruppenarbeit implementiert und ausprobiert. Aktuell wird hauptsächlich der erste Ansatz verwendet, wobei durch eine globale Umgebungsvariable die komplette Fehlerinjektion aktiviert werden kann. Zudem kann die Konfigurationsdatei für den Control-Node ebenfalls durch eine Umgebungsvariable gesetzt werden.

Für die Deaktivierung eines aktuell injizierten Faults wird auf die von ROS zur Verfügung gestellte Klasse des Timers zurückgegriffen. Für einen reibungslosen Ablauf werden Faults, die zum Start der Simulation aktiviert werden sollen, bereits beim Start des Programms zur Fehlerinjektion aktiviert. Dies dient dazu, um Fehler zu simulieren, die nicht erst während der Fahrt auftreten. Beispielsweise könnte ein Kabel lose sein, sodass die Kommunikation zu einem Gerät unterbrochen ist. Das System, das schon vor Start des Rennens Sensordaten bekommt, soll also insgesamt keine Informationen erhalten, die durch den Fault verhindert werden. Die Dauer eines solchen Fault startet erst mit dem Start des Rennens, sodass ein kurzer Fault nicht schon zu Ende sein kann, wenn der Rennwagen losfährt.

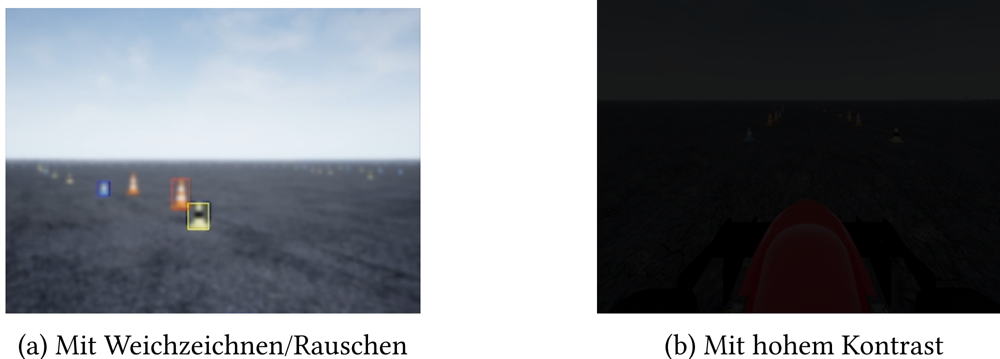
7.2.4 Implementierte Faults

Mehrere Faults wurden mithilfe der nachrichtenbasierten Methode implementiert. Diese werden im Folgenden erläutert.

Lenkung Bei der Lenkung können zwei Arten von Faults simuliert werden. Zum einen kann die Lenkung komplett blockiert werden, sodass das Lenken nicht mehr möglich ist. Hierzu wird der erste mitgeteilte Lenkwinkel nach dem Aktivieren des Fault gespeichert. Über die gesamte Dauer wird mit diesem Wert die von JARVIC berechneten Winkel überschrieben. Daneben kann als Fault noch eine Nullpunktabweichung der Lenkung simuliert werden, sodass bei aktivierten Fault das Rennauto, wenn es geradeaus fahren soll, nach links oder rechts fährt. Die Abweichung kann über Parameter angepasst werden. Nach Absprache mit GET racing kommt es auch vor, dass die Abweichung mit der Zeit abnimmt, da bei tatsächlichem maximalem Ausschlag der Lenkung die Abweichung um den Wert verringert wird, der die Differenz zwischen dem tatsächlichen und berechneten Winkel ist. Bei kurzen Lenkungs-Faults kann sich das Fahrzeug ohne Behandlung meist noch wieder zurück auf die Strecke manövrieren oder verlässt diese gar nicht erst. Längere, unrealistischere Faults, bei denen das Fahrzeug weit von der Strecke abkommt, können dazu führen, dass das Fahrzeug nicht oder nicht an der richtigen Stelle wieder zurück auf die Strecke findet. Bei beiden Faults handelt es sich um Faults, welche Zugriff auf die vorhergehende Nachricht benötigen und somit einen Zustand zwischenspeichern (vgl. Unterabschnitt 7.2.1).

Motor Um einen Motorschaden simulieren zu können, wird der berechnete Wert der Beschleunigung verändert, sodass eine Bremsung volles Potenzial hat, aber eine Erhöhung der Geschwindigkeit erschwert wird. Die Beschleunigung wird im Intervall $[-1, 1]$ angegeben, wobei negative Werte bedeuten, dass der Wagen bremst. Der Motorschaden wird in Prozent angegeben. Die injizierte Beschleunigung ergibt sich aus dem von JARVIC berechneten Soll-Beschleunigung, die um den Prozentwert des Schadens verringert wird. Je nach Intensität des Faults bewegt sich das Fahrzeug langsamer oder gar nicht mehr.

SLAM-Ausfall Die SLAM-Komponente, die wesentlicher Bestandteil der Verarbeitung der Sensordaten ist, hat aufgrund dessen eine wichtige Funktionalität. Ihr Ausfall führt dazu, dass die Umgebungsmap nicht mehr aktualisiert wird, und die Richtung des Fahrens nicht mehr bestimmt werden kann. Den Ausfall von dieser Komponente wird dadurch simuliert, dass alle von diesem Node gesendeten Nachrichten verworfen werden. Ohne Behandlung bewegt sich das Fahrzeug weiter entlang der geplanten Trajektorie. Diese kann jedoch auch in der Erweiterung außerhalb des Tracks führen und wird aufgrund fehlender neuer Daten des SLAM nicht weiter aktualisiert.



(a) Mit Weichzeichnen/Rauschen (b) Mit hohem Kontrast

Abbildung 7.6: Fehlerinjektion auf Perception mit FSDS.

YOLO JARVIC verwendet in der Perception YOLO, um auf den Kamerabildern die Leitkegel entsprechend ihrer Farbe zu klassifizieren. Hierfür wurde ein Fault implementiert, welcher den Ausfall der YOLO-Komponente simulieren kann. Dabei können entweder alle Ausgaben der Komponente verworfen werden, oder nur diejenigen vom linken oder rechten Bild (bei Stereo-Kamera-Setup). Dieser Fault ist nur in Simulatoren anwendbar, die auch die Perception simulieren (hier FSDS).

Kamerabilder Noch tiefer im Perception-Stack setzt der implementierte Fault an, welcher direkt Veränderung auf den eingehenden Kamerabildern durchführt. Hierdurch können Fehler bei der Bildaufnahme, wie bspw. durch eine fehlerhafte Kamera oder Fehler in der Konfiguration (zu lange Belichtungszeit, ...), simuliert werden. Aktuell unterstützt dieser implementierte Fault das Verwerfen der Kamerabilder, um den Ausfall der Kamera simulieren zu können. Ebenfalls kann ein Verrauschen mittels Weichzeichnen der Bilder angewandt werden (vgl. Abbildung 7.6a) sowie der Kontrast und die Helligkeit der Bilder angepasst werden (vgl. Abbildung 7.6b). Dazu wird die OpenCV-Bibliothek verwendet. Je nach Intensität des Faults werden weniger oder keine Leitkegel mehr erkannt. Dieser Fault ist ebenfalls nur in Simulatoren anwendbar, die auch die Perception simulieren (hier FSDS).

7.2.5 Generische Fehlerinjektion

Eine überlastete Node kann durch Verzögern oder Löschen von Nachrichten simuliert werden. In dem bisher beschriebenen Ansatz bedeutet dies, dass für jeden einzelnen Nachrichtentyp diese Funktionalität vor Verwendung entwickelt werden muss. Zur Compile-Zeit wird die Typisierung der Topics der Fehlerinjektion benötigt. Stattdessen kann über eine globale Node Nachrichten unabhängig vom Nachrichtentyp Fehler injiziert werden. Doppelte Implementierung der gleichen Funktionalität wird durch diese Node eingespart. Der Erhalt von Nachrichten auf Topics kann über einen Befehl in ROS2 verzögert werden. Jedoch ist das

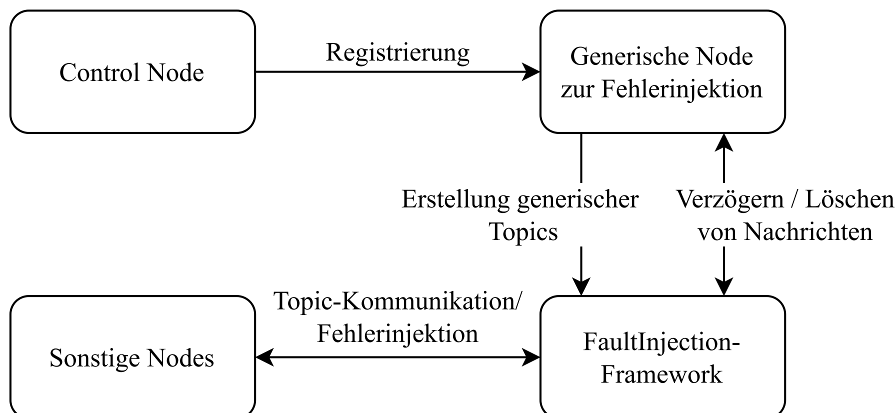


Abbildung 7.7: Struktur der generischen Fehlerinjektion.

manuelle Ausführen eines solchen Befehls zum Simulieren eines Fehlers zeitlich nicht präzise und nicht wiederholbar.

Das Ziel bei der Erstellung der Node zur generischen Fehlerinjektion ist es, dass so wenig Wissen wie möglich über Nachrichtentypen benötigt wird. Dazu gehören zum Beispiel die in einer Nachricht enthaltenen Attribute. Die Struktur der Nodes für die generische Fehlerinjektion ist in Abbildung 7.7 dargestellt. Nach dem Lesen der zu injizierenden Fehler wird an sie durch die Control-Node erst zur Laufzeit der Name und der Nachrichtentyp der ausgewählten Topics zur Registrierung übergeben. Für jedes erhaltene Topic werden die benötigten Subscriber und Publisher für die Fehlerinjektion erstellt. Mit der Verwendung der ROS2-API für C++ werden der Name und der Nachrichtentyp für die Erstellung von Publisher und Subscriber spätestens zur Laufzeit benötigt. Der Ablauf zur Manipulation der Nachrichten ist danach ähnlich zu dem Normalfall, der in Unterabschnitt 7.2.1 beschrieben ist. Die Kommunikation zu den anderen gestarteten Nodes ist daher gleich. Als Arten der Manipulation stehen die Verzögerung und das Löschen von Nachrichten zur Verfügung. In ROS Noetic ist die API für generische Publisher und Subscriber so unterschiedlich, dass diese Features nur in ROS2 implementiert wurden.

Diese Art der Implementierung hat im Gegensatz zum eingangs erwähntem CLI-Substitut den Nachteil, dass die Umleitung der zu injizierenden Topics aktiviert werden muss. Beispielsweise können Umgebungsvariablen dafür genutzt werden. Sonst funktioniert die Weiterleitung nur, wenn ein Fault für das Topic injiziert wird. Andererseits kann für den Zeitraum der Simulationen für die Fehlerinjektion das entsprechende Topic in der Fehlerliste mit einem Startwert so aufgeführt werden, dass letztendlich der Fault nicht aktiviert wird. Somit wird diese Einschränkung umgangen.

Das Verzögern und Löschen sind als eigenständiger Fault implementiert, wobei das Verzögern einen Parameter über die Dauer der Verzögerung benötigt. Das Verzögern ist über Multithreading umgesetzt. Am Ende eines solchen Faults werden verzögerte, aber noch nicht gesendete

Nachrichten gelöscht. Dies verhindert, dass zu dem Endpunkt nicht alle diese Nachrichten auf einmal an den oder die entsprechenden Subscriber gesendet werden. Weitere Faults, die unabhängig von dem Nachrichtentyp sind, lassen sich, wie bei den anderen Nodes zur Fehlerinjektion hinzufügen.

7.2.6 ROSMonitoring zur Fehlerinjektion

Neben der Implementierung des PG-eigenen Fehlerinjektionsframeworks, wurde sich anfänglich auch mit ROSMonitoring beschäftigt [Fer+20]. ROSMonitoring ist ein Tool zur Laufzeitüberwachung von Nachrichten anhand eines Formalismus und ist unter einer Open-Source-Lizenz verfügbar[1]. Um die Nachrichten zur Überwachung abzufangen, bietet ROSMonitoring ein Tool zur Instrumentierung an, welches anhand einer YAML-Konfigurationsdatei ROS-Python-Nodes als Monitore erstellt und die Launchdateien entsprechend anpasst. Die Monitore können dabei so erstellt werden, dass sie nebenher auf ein Topic hören und bspw. die Nachrichten in eine Log-Datei schreiben, oder die Kommunikation, wie in Abbildung 7.3b dargestellt, unterbrechen. In letzterem Fall sind die Monitore dabei mit der Idee der Fault-Nodes vergleichbar. Sie erhalten die abgefangene Nachricht und können entscheiden, ob diese auf das eigentliche Topic weitergeleitet werden soll. Die Logik, ob eine Nachricht weitergeleitet werden soll, ist bei ROSMonitoring jedoch in einer separaten Komponente, dem Orakel, implementiert. Dieses erhält die eingehenden ROS-Nachrichten per Websocket und gibt eine Antwort an den Monitor zurück. Aufgrund mehrerer Aspekte wurde ROSMonitoring jedoch nicht weiter für die Fehlerinjektion betrachtet. Zum einen kann der extra Hop über das Netzwerk zum Orakel eine weitere Verzögerung hinzufügen, welche ggf. bei einer hohen Kommunikationsfrequenz Auswirkung auf das System haben kann. Des Weiteren können Monitore bei ROSMonitoring bislang die Nachricht beim Weiterleiten nicht editieren. Eine einfache allgemeine Erweiterung, mit der eine beliebige Nachrichtenmanipulation über die Konfigurationsdatei in Python-Code integrierbar ist, schien nicht einfach möglich. Daher wurde sich auf das selbst entwickelte Fehlerinjektionsframework fokussiert.

7.3 Fehlerinjektion in der Simulation durch Veränderung der Map

FSSIM und PacSim simulieren nicht direkt die Perception, sondern laden eine Map, welche die vorhandenen Leitkegel abbildet. Trotzdem ist ein Ziel der Projektgruppe, bereits mit diesen Simulatoren Fehler wie nicht oder falsch erkannten Leitkegel zu simulieren. Auch soll das Verhalten von JARVIC bei nicht ideal platzierten Leitkegeln untersuchbar gemacht werden. Beides kann durch eine Veränderung der Maps der Simulationsumgebung umgesetzt

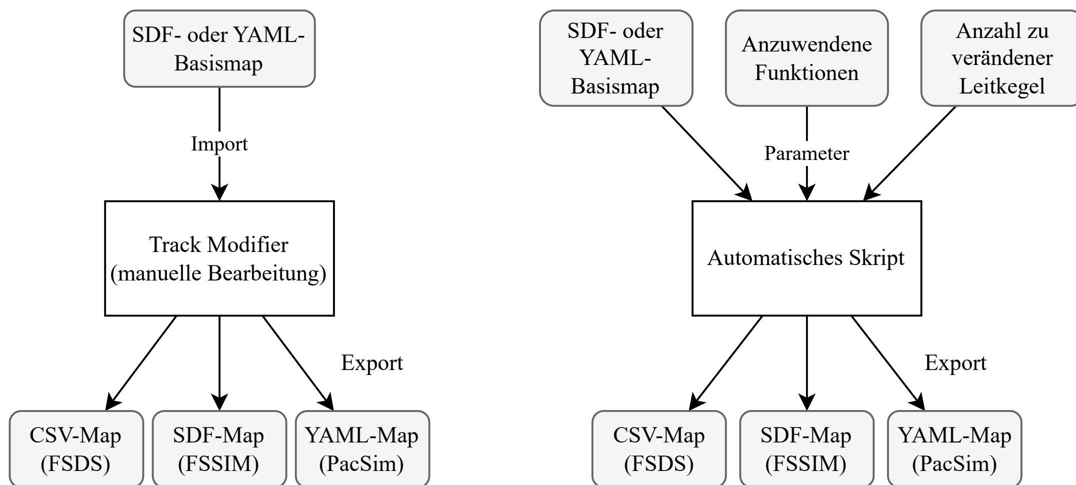
[1] https://github.com/autonomy-and-verification-uol/ROSMonitoring

werden, da diese unter anderem die Positionen der einzelnen simulierten Leitkegel enthalten. Für die Modifikation der Karten wurde daher der *Track Modifier* entwickelt. Ein Ziel des Track Modifiers ist es, die Map für unterschiedliche Simulatoren exportieren zu können. Damit kann erreicht werden, dass die exakt gleiche Map in Simulatoren mit unterschiedlichen Untersuchungsschwerpunkten verwendet werden kann. Aktuell ist der Export im SDF-Format für FSSIM, als CSV für FSDS und als YAML für PacSim möglich. SDF ist ein XML-Dateiformat, welches FSSIM für die Repräsentation der Maps verwendet [2]. Eine Erweiterung für andere Dateiformate ist einfach möglich, sofern diese eine entsprechende Struktur für die Formula-Student-typischen Leitkegel besitzen. Auch der Import ist in Zukunft leicht erweiterbar, falls die Notwendigkeit besteht, andere Formate, als SDF oder YAML zu importieren. Zur Modifikation der Maps wurden zwei Möglichkeiten entwickelt: Sie können entweder manuell über die Benutzeroberfläche des Track Modifiers oder automatisiert durch ein Skript geändert werden.

Bei dem Skript handelt es sich um eine Klasse in Python, die verschiedene Funktionen zur Veränderung einer von den Nutzer:innen übergebenen Basiskarte zur Verfügung stellt. Die Ein- und Ausgaben dieser Modifikationsart sind in Abbildung 7.8b dargestellt. Instanzen der Klasse erhalten eine Basiskarte im SDF- oder YAML-Format sowie die Anzahl der Leitkegel, welche bei Aufruf der Funktionen verändert werden. Anschließend können einzelnen Funktionen aufgerufen werden, welche jeweils die Basiskarte verändern und die modifizierte Map unter einem angegebenen Dateinamen abspeichern. Die Informationen zur Basiskarte, gewünschte Funktion und Anzahl der Kegel, sowie der zu speichernde Ort für die neue Map werden über das Terminal angegeben. Damit ist es möglich, schnell viele modifizierte Maps zu erstellen. Diese leicht voneinander unterschiedlichen Maps können zum Testen hinsichtlich unter minimal unterschiedlichen Bedingungen verwendet werden, damit die dadurch auftretenden Fehler anschließend behandelt werden können. Aktuell können folgenden Funktionen angewandt werden:

- Entfernen von Leitkegeln an zufälligen Stellen aus der Basiskarte.
- Duplizieren von zufälligen Leitkegeln, um durch die Perception dupliziert erkannte Leitkegel zu simulieren (vgl. Abschnitt 6.1).
- Zufälliges leichtes Verschieben von Leitkegeln, als seien diese schlecht positioniert.
- Zufälliges Ändern des Typs des Leitkegels, um durch die Perception falsch erkannte Leitkegel zu simulieren (Position und Typ).
- Hinzufügen von Leitkegel in der Mitte der Karte, um auf die Strecke geschleuderte Leitkegel zu simulieren.
- Entfernen einer Reihe an Hütchen.

[2] http://sdformat.org/



(a) Manuelle Modifikation mittels Track Modifier. (b) Automatische Modifikation mittels Skript.

Abbildung 7.8: Datenfluss bei den beiden entwickelten Ansätzen zur Mapmodifikation der Simulationen.

Die Benutzeroberfläche für den interaktiven Modus des Track Modifier wurde ebenfalls mit Python entwickelt und kann daher auf verschiedenen Plattformen wie Linux oder Windows ausgeführt werden. Als UI-Framework wurde tkinter sowie Matplotlib zur Darstellung der Map verwendet. Die Ein- und Ausgaben des Track Modifier im interaktiven Modus sind in Abbildung 7.8b abgebildet. Ein Screenshot vom Track Modifier ist in Abbildung 7.9 dargestellt. Die Leitkegel können dabei interaktiv über die angezeigte Karte per Drag and Drop verschoben werden. Ebenfalls können weitere Operationen aus der UI ausgeführt werden, wie das Hinzufügen oder Entfernen von Leitkegeln.

7.4 Fehlerinjektion in der Simulation durch Veränderung des Wetters

FSDS unterstüzt einige Wettereffekte wie Regen, Schnee, Nebel, Staub usw. Es ist auch möglich, die Intensität eines Wettereffekts zu steuern (z. B starker/schwacher Regen). Da FSDS und JARVIC über ROS-Nachrichten kommunizieren, ist es auch möglich, Wettersteuerung durch den oben erwähnten Control-Node zu kontrollieren. Dafür wurde ein neuer Nachrichtentyp erstellt. Eine Wetternachricht besteht aus einer Liste der Wettereffekte, die eingehalten werden sollen, und einer Liste der entsprechenden Intensitäten. Der Simulatorcode wurde um eine passende Callback-Funktion ergänzt, um die Wetternachrichten zu empfangen, auszupacken und das Wetter zu ändern.

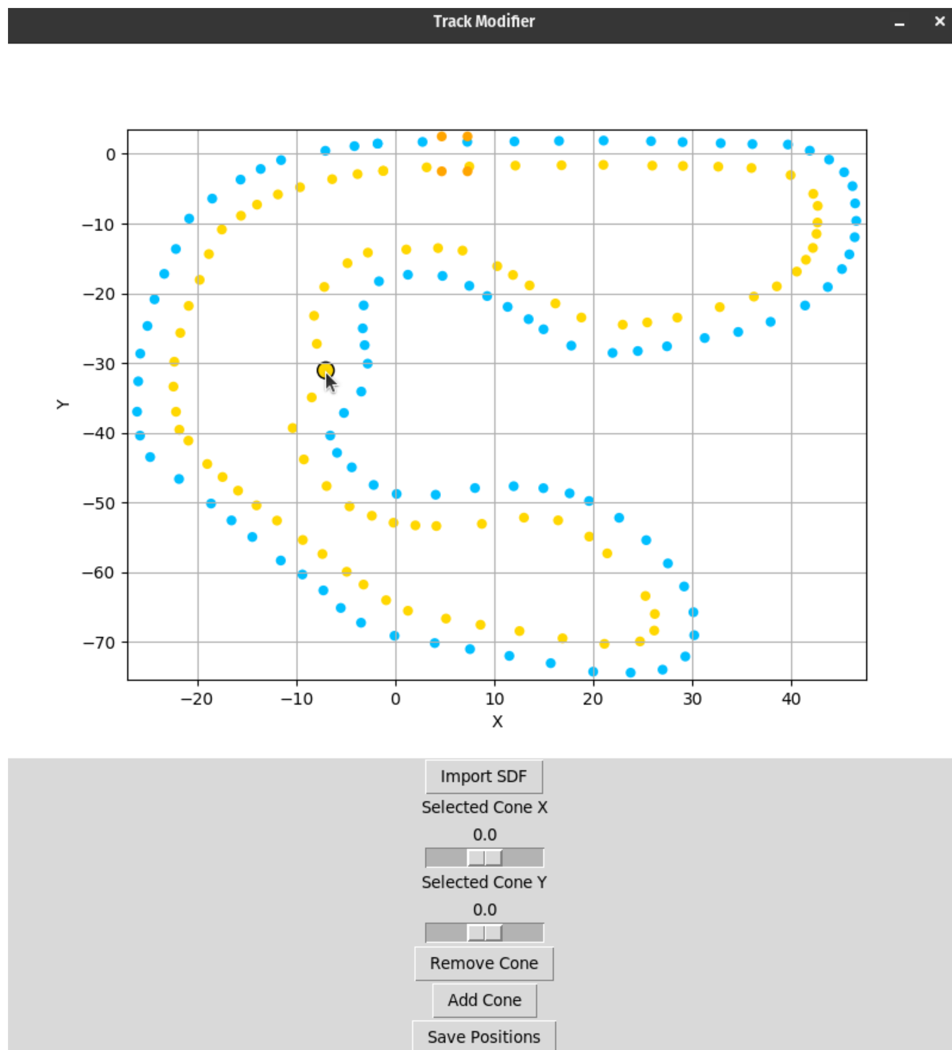


Abbildung 7.9: Screenshot des Track Modifiers.

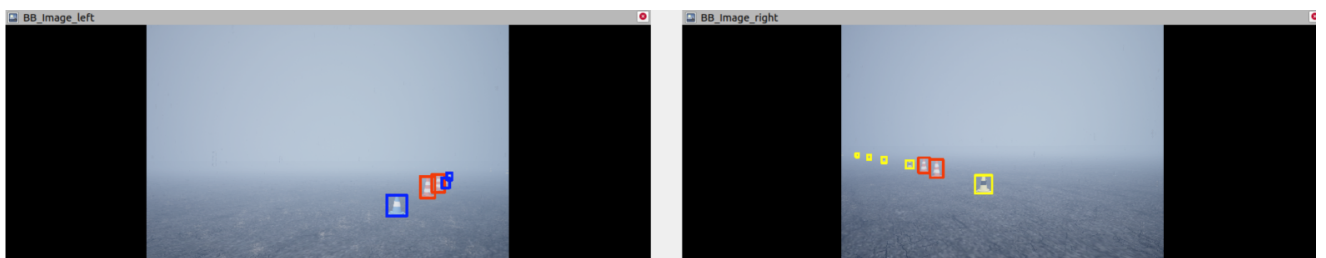


Abbildung 7.10: Nebel und Regen verkürzen die Reichweite der Perception, nähere Leitkegel werden aber richtig erkannt.

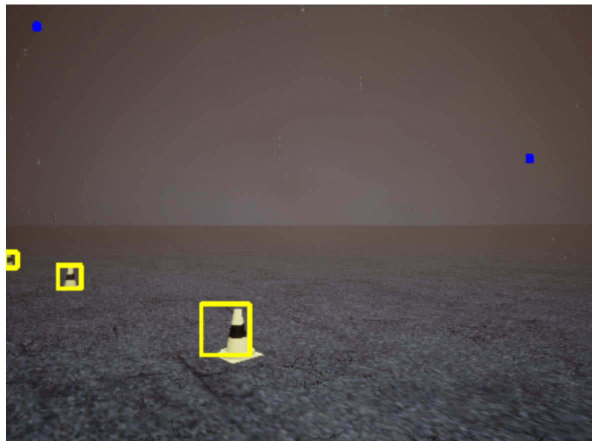


Abbildung 7.11: Effekte von Staub und Regen auf die Perception. Wassertropfen werden manchmal unter diesen Bedingungen als blaue Hütchen erkannt.

Der Einfluss von Wettereffekten auf die Perception kann in den Abbildungen 7.10 und 7.11 beobachtet werden. Wettereffekte reduzieren möglicherweise die Reichweite, aber die Erkennung von naheliegenden Objekten bleibt korrekt. Daher hat es kaum Einfluss auf Stabilität von JARVIC. Interessanterweise werden Wassertropfen bei Kombination von Staub und Regen selten als blaue Hütchen erkannt. Diese Erkennung verschwindet aber in weniger als eine Sekunde und hat daher auch keinen Effekt auf JARVIC.

7.5 Integration der Fehlerinjektion in die CI-Pipeline

Um später bei der Implementierung von Resilienzmaßnahmen auch eine automatisierte CI-basierte Validierung zu ermöglichen, wurde die Fehlerinjektion in die CI integriert. Zum aktuellen Stand ist die Fehlerinjektion nur für ROS1 mit FSSIM in die CI-Pipeline integriert, da bedingt durch die nicht abgeschlossene ROS2-Migration mit ROS2 und PacSim noch keine relevanten Ergebnisse zu erzielen wären. Bislang ist nur für die nachrichtenbasierte Fehlerinjektion (vgl. Abschnitt 7.2) geschehen, indem diese mit einer fest einprogrammierten Konfiguration ausgeführt wird. Diese Fehlerinjektion lässt sich durch das Setzen einer Umgebungsvariable vor dem Start von JARVIC aktivieren. Ebenfalls wird der Pfad zur Konfigurationsdatei des Control-Nodes durch eine Umgebungsvariable spezifiziert. Die FSSIM-Konfigurationsdatei, welche für die Konfiguration der CI-Simulationsausführungen verwendet wird (vgl. Abschnitt CI Integration), ermöglichte allerdings nicht das Setzen von Umgebungsvariablen für einzelne Simulationsdurchläufe. Daher wurde die FSSIM-Konfigurationsdatei so modifiziert, dass je Simulationsdurchlauf mit dem Keyword `environment` zusätzlich noch Umgebungsvariablen für den Befehl zum Starten der Fahrsoftware gesetzt werden können. Somit kann durch Hinzufügen weitere Simulationsausführungen in der FSSIM-Konfigurationsdatei relativ einfach JARVIC mit aktivierter, als auch ohne aktivierter Fehlerinjektion in der

```
1  repetitions:
2  - {
3      sensors_config_file: fssim_config/sensors/sensors_1.yaml,
4      track_name: FSG.sdf,
5      environment: {
6          LAUNCH_FAULT_INJECTION: "1",
7          CONTROL_COMMAND_MASTER_CONFIG_PATH: "config.yml"
8      },
9      autonomous_stack: roslaunch common_meta generic_with_fssim_interface.
         launch launchfile:='$(find common_meta)/missions/trackdrive.
         launch'
10 }
```

Listing 7.5: Ausschnitt aus einer FSSIM-Konfiguration, um einen Durchlauf mit aktivierter Fehlerinjektion auszuführen.

gleichen Konfiguration ausgeführt werden. Eine FSSIM-Beispielkonfiguration mit aktivierter Fehlerinjektion ist in Listing 7.5 dargestellt.

In der letzten ROS1 Version der CI-Pipeline von JARVIC ist implementiert, dass je ein Durchlauf ohne und ein Durchlauf mit Fehlerinjektion ausgeführt wird. Für den Durchlauf mit aktivierter Fehlerinjektion wird zurzeit eine statische Konfiguration des Fault Control-Nodes verwendet. Diese injizierte einen Motorschaden 15 Sekunden nach Simulationsstart. Die statische Konfiguration wird aktuell nur testweise verwendet, um den generellen Ablauf der Fehlerinjektion und deren CI-basierte Ausführung zu testen und weiterzuentwickeln. Perspektivisch sollen mehrere sinnvolle Fehlerszenarien integriert werden, welche über die Szenariensprache definiert werden. Ebenfalls soll die Fehlerinjektion über Veränderungen an der Map sowie ggf. zukünftige weiter entstehende Fehlerinjektionsmethoden für FSDS in die CI integriert werden. Zur Vereinheitlichung und Vereinfachung der Ausführung soll zudem der Executor (vgl. Kapitel 9) auch für die CI-Ausführung verwendet werden. Die bislang verwendete direkte Integration der Fehlerinjektion ohne den Executor in die CI wurde nur verwendet, um die Infrastruktur und den Entwicklungsworkflow zu testen und aufzubauen.

Als ein mögliches Problem bei der Ausführung von vielen Fehlerszenarien in der CI wird noch die CI-Laufzeit angesehen. Idealerweise sollte pro Runner aufgrund der hohen Hardwareanforderungen nur eine Simulation parallel ausgeführt werden, damit sich die Simulationen nicht gegenseitig beeinflussen. Gleichzeitig soll die CI ein schnelles Feedback liefern. Da jedoch bislang für die CI immer nur ein Fehlerszenario getestet wurde, wurde sich mit dieser Problematik nicht weiter beschäftigt.

Kapitel 8

Domänenspezifische Sprache

In dem folgenden Kapitel wird auf die Entwicklung der domänenspezifischen Sprache FLEX eingegangen. Dabei wird zuerst beschrieben, weshalb die Entwicklung einer eigenen Sprache sinnvoll erschien. Danach wird auf die dazugehörige Grammatik eingegangen, gefolgt von einer Betrachtung der Konzepte der Sprachentwicklung und den zugrunde liegenden Syntaxbäumen. Abschließend wird auf die Implementierung und Codegenerierung eingegangen.

8.1 Grundkonzept FLEX

FLEX ergab sich als Notwendigkeit beim Injizieren von Fehlern. Nachdem ein Fehler bestimmt, analysiert und injiziert werden konnte, entstand der Bedarf nach einer nutzerfreundlichen Möglichkeit diesen eingeben zu können. Da es keine Sprache für diesen Anwendungsfall gab, musste jeder Fehler einzeln definiert werden. Es waren keinerlei Strukturen festgelegt, um Fehler zu beschreiben. Dabei sind diese Fehlerszenarien notwendig, um das Verhalten des Fahrzeuges zu testen und zu einem späteren Zeitpunkt resilient fahren zu können. Das Ziel war dementsprechend mit FLEX zu ermöglichen, Fehler zu beschreiben und diese darstellen zu können. Besonders relevant sind dabei auch Kombinationen aus Fehlern.

Die Ziele für die Sprachen waren Nutzerfreundlichkeit, Geschwindigkeit, Wartbarkeit und Erweiterbarkeit. Eine selbst entwickelte Sprache bietet dabei Flexibilität. Es besteht die Möglichkeit eigene Konstrukte einzuarbeiten, welche relevant für die Verkettung von Fehlern sind. Durch eigene Anpassungen wird zudem die Nutzerfreundlichkeit und die Anpassbarkeit im Falle von Änderungen gewährleistet werden.

Konzepte der Sprachentwicklung Ein grundlegendes Konzept in der Sprachentwicklung ist die Definition einer formalen Grammatik. Eine Grammatik beschreibt die Struktur einer Sprache durch Regeln. Diese legen fest, wie gültige Programme oder Sätze gebildet werden können. Sprachen wie FLEX basieren auf kontextfreien Grammatiken (CFGs). Eine kontextfreie Grammatik besteht aus einer Menge von Nichtterminalen, Terminalsymbolen, einem

Startsymbol und einer Reihe von Regeln, bei denen immer genau ein Nichtterminal auf der linken Seite steht. Dadurch können komplexe Strukturen rekursiv beschrieben werden, ohne dass der umgebende Kontext eine Rolle spielt. Die Produktionen bestehen aus Nichtterminalen und Terminalsymbolen und legen fest, welche Sequenzen erlaubt sind, sodass eine eindeutige und systematische Zerlegung von Eingabetexten möglich ist.

Ein weiteres zentrales Konzept sind die Syntaxbäume: Ein konkreter Syntaxbaum (CST) stellt den vollständigen Parse-Baum eines Programms dar, inklusive aller syntaktischen Elemente wie Schlüsselwörter, Interpunktionszeichen und Operatoren. Der CST dient als Zwischenrepräsentation, die alle Details der grammatikalischen Struktur abbildet. Der abstrakte Syntaxbaum (Abstract Syntax Tree, AST) enthält eine reduzierte und abstrahierte Darstellung der syntaktischen Struktur eines Programms. Dabei werden unnötige syntaktische Details eliminiert und sich auf die wesentliche semantische Struktur konzentriert.

Langium und seine Verwendung in FLEX Langium wurde als Entwicklungswerkzeug für FLEX eingesetzt. Damit wurde eine leistungsfähige und erweiterbare domänenspezifische Sprache erstellt. Langium basiert auf TypeScript und ermöglicht eine Definition von Sprachen durch deklarative Grammatikregeln. Diese Regeln definieren die Struktur von FLEX. Dadurch wird gesteuert, wie Texte geparst und in abstrakte Syntaxbäume umgewandelt werden.

Durch den Einsatz von Langium konnte eine automatische Fehlerprüfung, Code-Validierung und eine Entwicklungsumgebung in VS Code implementiert werden. Langium bietet zudem Mechanismen zur Semantikvalidierung. Dadurch werden nur sinnvolle und gültige Eingaben akzeptiert. So werden beispielsweise Zeitspannen und Werte in FLEX überprüft, um fehlerhafte Definitionen frühzeitig zu erkennen.

Ein weiterer Vorteil von Langium ist die Möglichkeit, angepasste Codegeneratoren zu integrieren. In FLEX wird dies genutzt, um die definierten Fehlerszenarien in eine maschinell lesbare Form zu übersetzen. Dadurch können Testszenarien direkt aus den FLEX-Beschreibungen abgeleitet werden, ohne dass manuelle Anpassungen nötig sind.

Syntax & Semantik In FLEX werden strukturierte Sätze verwendet, um Fehlerszenarien verständlich zu beschreiben. Diese Sätze folgen einem klaren Muster, das sowohl die Eingabe vereinfacht als auch die maschinelle Verarbeitung erleichtert. Grundsätzlich lassen sich die Sätze in FLEX in zwei Haupttypen unterteilen: Konfigurationssätze: Diese legen die grundlegenden Rahmenbedingungen für eine Simulation fest. Dazu gehören Parameter wie der verwendete Simulator, die Umgebungskarte und die Startposition des Fahrzeugs. Ein Beispiel für den Simulator FSSIM, der Map „Test“, einer Startposition von [8,10] und einem Startwinkel von -1π ist in Listing 8.1 dargestellt.

```
1   Use simulator fssim with map test and startposition [8,10] and yaw -1
       pi degrees.
```

Listing 8.1: Konfiguration der Simulation mit FLEX

Fehlerszenario-Sätze: Diese beschreiben, welche Fehler injiziert werden sollen, unter welchen Bedingungen sie auftreten und wie sie sich auf das Fahrzeug auswirken. Die Struktur der Fehlerbeschreibung ist im Ordnerkonzept aufgebaut. Durch diesen strukturierten Ansatz wird die Semantik der Sprache intuitiv erfassbar. Gleichzeitig wird die maschinelle Verarbeitung erleichtert. Die Syntax ermöglicht eine direkte Übersetzung in ausführbare Testszenarien. Ein möglicher Fehler ist eine Störung in der Lenkung. Dieser gehört zur Kategorie „control“ und davon ausgehend in den Bereich „Lenkung“ und in den Unterbereich „Lenkwinkel“. Auf den ersten Blick ist erkennbar, ob es sich um einen Control- oder um einen Perception-Fehler handelt. Außerdem haben wir uns für die weiteren Verwendungszwecke dafür entschieden, dass es möglich ist Sekunden und Millisekunden zu übergeben. Im Anschluss werden die Sekunden ebenfalls in Millisekunden umgerechnet. Dies hat sich als praktikabel für die weitere Bearbeitung herausgestellt.

Grundsätzlich lassen sich die mit FLEX darstellbaren Fehler in drei Kategorien einteilen: Control, Perception und Weather. Bei den Control-Fehlern unterscheiden wir aktuell in Maschinen-bezogene und Steuerungs-bezogene Szenarien. Diese Fehler setzen sich aus der Fehlerbezeichnung, dem Verhalten, zusätzlichen Attributen und dem Zeitpunkt zusammen. In Listing 8.2 wird ein nach 15 Grad verschobener Lenkwinkelaktor im Zeitraum von fünf bis zwanzig Sekunden beschrieben.

```
1 Use simulator fsds with map test2.
2 Fault control/steering_actor/angle_deviation 15 degrees left in
    [5, 20] seconds.
```

Listing 8.2: Beispiel für einen Fehler im Lenkwinkelaktor mit FLEX

Die Perception-Fehler stellen Probleme mit Leitkegeln dar. Dazu gehört die fehlerhafte Erkennung oder Einordnung von Leitkegeln und ein nicht ordnungsgemäßer Aufbau der Leitkegel. Dieses Szenario kann zum Beispiel eintreten, wenn in der ersten Runde bereits Leitkegel von ihren Ausgangspositionen gestoßen wurden. Der Aufbau der Fehler ist nahezu identisch zu Control-spezifischen Fehlern. Perception-Fehler enthalten dabei keine Zeitangabe, da die Leitkegel bereits zu Beginn aus der Karte entfernt werden müssen. In Listing 8.3 wird ein Fehlen von acht Leitkegeln auf der Strecke injiziert.

```
1 Use simulator fsds with map test.
2 Fault perception/cones/missing_cones 8 cones.
```

Listing 8.3: Beispiel für eine fehlende Erkennung der Leitkegel

Bei den Weather-Fehlern handelt es sich um Fehler, bei denen Wetterbedingungen das Kamerabild verändern. Darunter fällt zum Beispiel Nebel oder Regen. Genau wie bei Perception-Fehlern wird hier kein Zeitpunkt angegeben, da das Wetter nur global beim Starten der

Simulation verändert wird. In Listing 8.4 wird starke Sonneneinstrahlung von der rechten Seite injiziert.

```
Use simulator fssim with map test3.
Fault perception/camera/lightning side right.
```

Listing 8.4: Beispiel für eine fehlende Erkennung der Leitkegel

FLEX verfügt über eine mit Docusaurus [1] erstellte Dokumentation, die kurz in die Fehlertypen und deren Syntax einführt. Darüber können alle Informationen gewonnen werden, die zur Erstellung von Szenarien notwendig sind. Neben den einzelnen Szenarien war bei FLEX von hoher Relevanz Verkettungen von Fehlern darzustellen. Bei diesen können unterschiedliche Parameter für die Fehler eingestellt werden und Fehlerdateien müssen nicht manuell erstellt werden.

Parameter werden nicht nur fest übergeben. Es sind drei Typen an Funktionalitäten implementiert.

Increment Der Wert des Fehlerparameters wird von null ausgehend, um einen bestimmten Wert erhöht.

Decrease Der Wert wird ausgehend von einem Startwert, über eine definierte Anzahl an Runden, um einen gewählten Wert reduziert.

Change Die verschiedenen Werte werden als Liste übergeben, um Werte manuell zu ändern.

Zusätzlich wird in „Runs“ und „Laps“ unterschieden. Dabei handelt es sich zum einen um Simulationsdurchläufe und zum anderen um Runden innerhalb einer Simulation. Für jeden Simulationsdurchlauf wird eine einzelne JSON-Datei erstellt, über welche die Simulation gestartet werden kann. Für jede Runde in einer Simulation wird ein Fehler-Ausschnitt in der JSON-Datei angelegt. Die Anzahl an „Runs“ und „Laps“ ist vom Nutzer wählbar. Bei der Decrease Funktionalität muss darauf geachtet werden, dass der Wert nicht aufgrund der Rundenanzahl unter null reduziert wird. Bei Verwendung von mehreren Features parallel wird das Kreuzprodukt der Fehlerwerte verwendet, um alle daraus entstehenden Szenarien abbilden zu können.

8.2 Implementierung

FLEX wurde mit Langium [2] implementiert. Die Implementierung ist in zwei Bereiche einzuteilen: die Grammatik mitsamt der Semantik und Syntax und die Codegenerierung. Die

[1] https://docusaurus.io
[2] https://langium.org

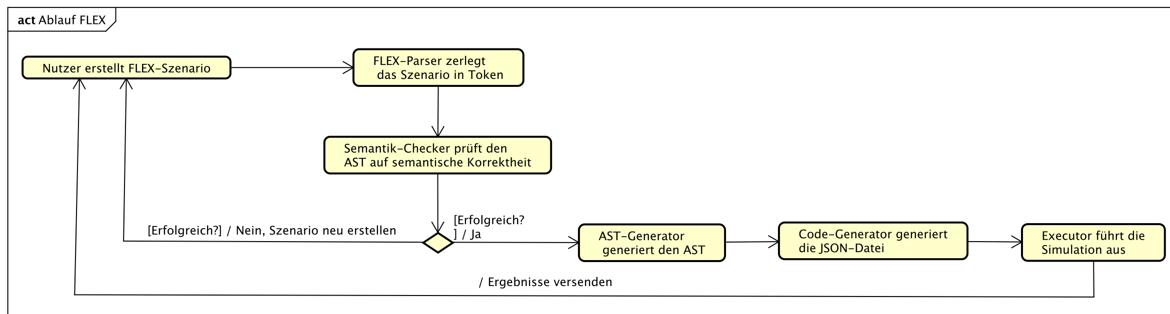


Abbildung 8.1: Ablauf vom FLEX-Szenario bis zum Executor als UML-Aktivitätsdiagramm.

```
Use simulator fsds with map test_map.
Fault control/engine_damage/torque_limit_factor 0.9 in [7,15] seconds.
```

Listing 8.5: Definiert einen Motorschaden durch den nur noch 90% der Motorleistung verfügbar ist. Der Fehler wird mit FSDS und der test_map in einem Zeitraum zwischen 7 bis 15 Sekunden simuliert.

notwendigen Schritte vom FLEX-Szenario bis zur Ausführung im Executor sind in Abbildung 8.1 dargestellt. Dabei wird auch das Zusammenspiel zwischen den einzelnen Komponenten deutlich. Nachdem der Nutzer das Szenario erstellt hat, wird die Eingabe vom FLEX-Parser in einzelne Token zerlegt. Diese werden vom Semantik-Checker auf semantische Korrektheit überprüft. Wenn ein Fehler vorliegt, muss das Szenario neu geschrieben werden. Ansonsten wird der AST generiert. Mithilfe dessen kann die JSON-Datei generiert werden und an den Executor weitergegeben werden. Dieser führt die Simulation aus und versendet die Ergebnisse an den Nutzer. Langium unterstützt bei der Entwicklung in den Bereichen der Syntax-Überprüfung, der Auto-Completion und dem Übergang zur Codegenerierung. Gemäß den Anforderungen von Langium wurde eine Grammatik erstellt. Diese definiert den Aufbau der Sprache und prüft, ob die Syntax korrekt verwendet wird. Die Eingabe wird in verschiedene Tokens zerlegt. Diese sind beispielsweise Schlüsselwörter, Operatoren und Parameter. In einem konkreten Syntaxbaum werden alle Token einer Eingabe dargestellt. Der konkrete Syntaxbaum für das Beispiel in Listing 8.5 ist in Abbildung 8.3 dargestellt. Der erste Knoten repräsentiert den Startpunkt der Grammatik, welcher in Abbildung 8.2 als Startknoten abgebildet ist. Davon ausgehend wird in das UseStatement und das FaultStatement unterschieden. Das UseStatement aus dem Beispiel bestand aus „Use simulator fsds with map test_map." und enthält keine weiteren Produktionen. Das FaultStatement wird ebenfalls bis auf die Token aufgetrennt. Es müssen dort auch weitere Produktionen, wie „ControlSpecification" aufgelöst werden. Dadurch wird deutlich, wie das Beispiel im Detail von der Grammatik definiert wird.

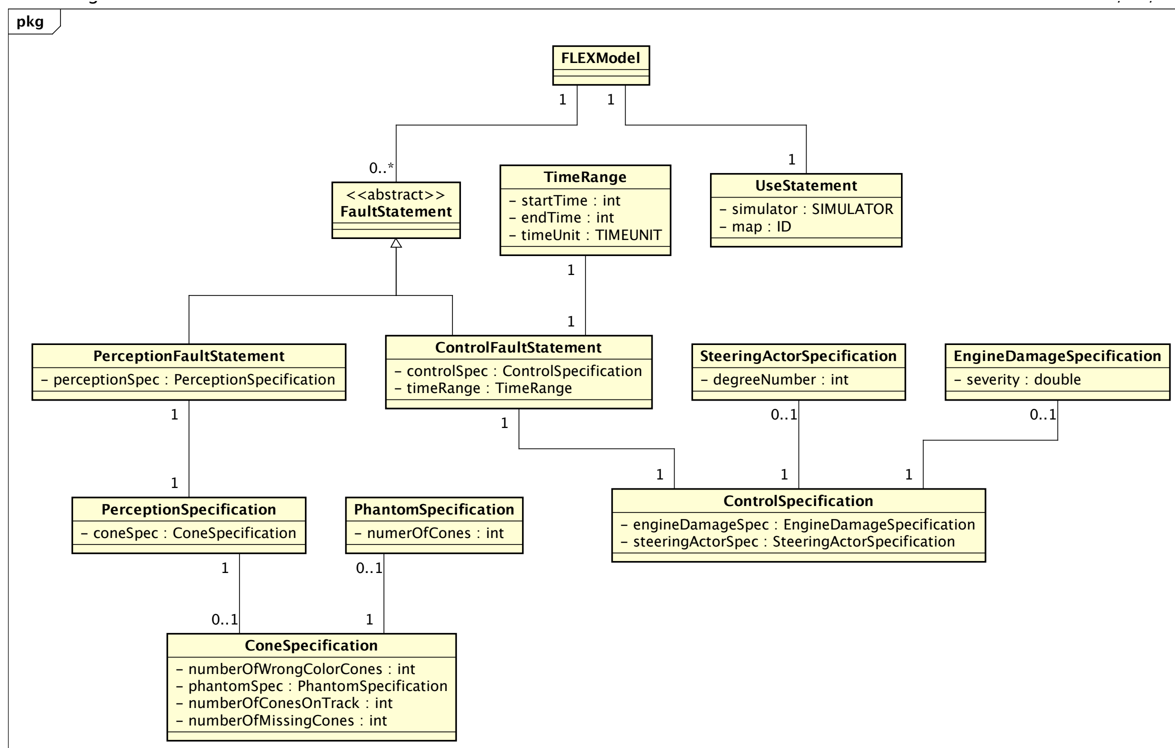


Abbildung 8.2: UML-Diagramm der FLEX Grammatik. Es werden die Definitionen in der Grammatik als Klassen in UML dargestellt.

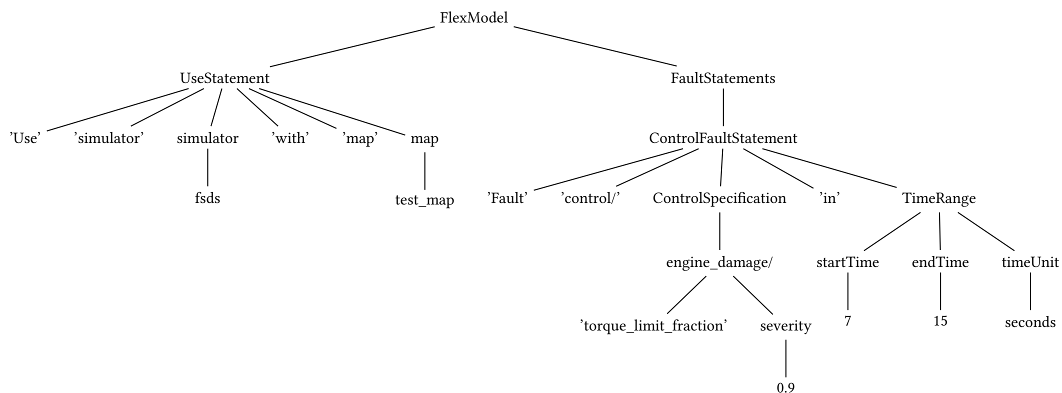


Abbildung 8.3: Konkreter Syntaxbaum für das Beispiel in Listing 8.5 basierend auf der FLEX-Grammatik abgebildet in Abbildung 8.2.

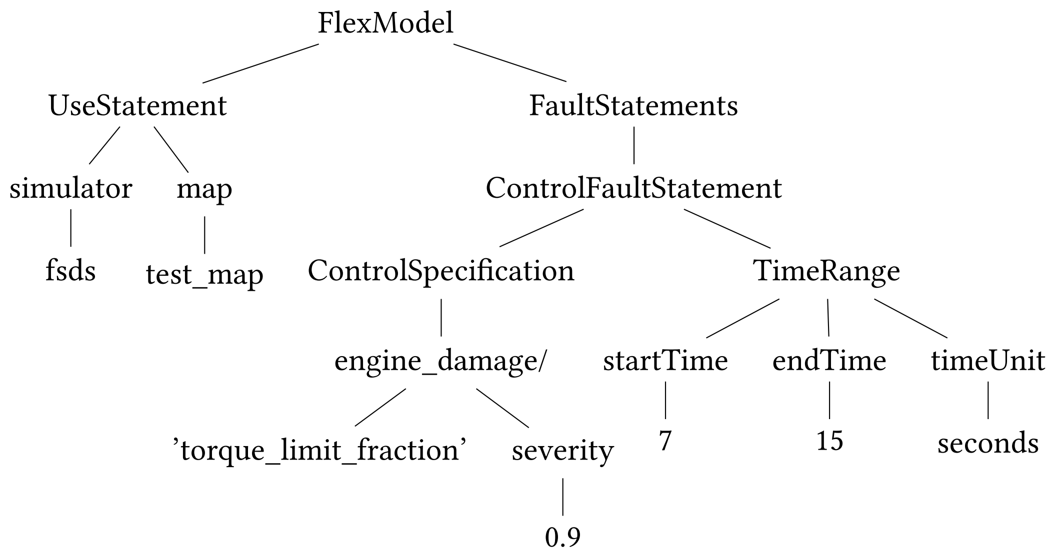


Abbildung 8.4: Abstrakter Syntaxbaum für das Beispiel in Listing 8.5 basierend auf der FLEX-Grammatik abgebildet in Abbildung 8.2.

Beim AST werden die für die maschinelle Verarbeitung notwendigen Elemente eingefügt. Konstrukte, die nur der Nutzbarkeit und Verständlichkeit dienen, werden entfernt. Der AST für ein FLEX-Konstrukt ist in Abbildung 8.4 abgebildet.

Nach der Erstellung des AST wird die semantische Analyse durchgeführt. Diese kann durch den Semantik-Check in Langium implementiert werden. In FLEX wird überprüft, ob die Zeitspannen und Fehlerparameter zulässige Werte haben. Dafür wird beispielsweise der Wert des Knotens TimeRange überprüft, um sicherzustellen, dass alle Fehlerszenarien korrekt sind, bevor Code generiert wird.

Mithilfe von Langium wird basierend auf dem AST Code generiert. Bei FLEX wird aus den strukturierten Sätzen JSON-Code erzeugt. Dieser wird für die weitere Verarbeitung genutzt. Aus dem Beispiel in Listing 8.5 wird der JSON-Code in Listing 8.6 generiert. Wie in Abbildung 8.4 abgebildet, beginnt der Baum mit dem FlexModel. Daraus wird ein UseStatement und ein Faultstatement erzeugt. Durch das UseStatement wird der Simulator und die Map definiert. Dargestellt sind diese Token in Abbildung 8.4 als Blätter des AST. Aus dem Knoten SIMULATOR mit dem Wert fsds, wird 'simulation_tool': 'fsds' generiert.

Auf diese Art und Weise wird der AST traversiert und Code generiert. Der Generator liest dafür zuerst das vollständige Fehlerszenario ein und teilt die Fehler in die verwendeten Konstrukte ein. Dabei bestimmt er beispielsweise, ob ein Lap-Increment Konstrukt verwendet wurde. Anhand dieser Informationen werden anschließend die benötigten JSON-Dateien oder JSON-Ausschnitte gebildet. Dadurch werden die Fehler unkompliziert für den Simulator

```
{
    "simulation_tool": "fsds",
    "map_name": "test_map",
    "faults": [
        {
            "type": "control",
            "timing": [
                "once",
                {
                    "time_start_ms": 7000,
                    "time_end_ms": 15000
                }
            ],
            "behavior": [
                "ENGINE_DAMAGE"
            ],
            "additional": {
                "partial_damage": 0.9
            }
        }
    ]
}
```

Listing 8.6: JSON-Code zum FLEX-Beispiel in Listing 8.5.

eingelesen. Für den Nutzer ist diese Darstellung unübersichtlich, weshalb die DSL entwickelt wurde.

Die DSL wird mithilfe einer VS-Code-Extension entwickelt. Diese unterstützt den Semantik-Check, den Syntax-Check, die Syntax-Vorschläge und die Code-Generierung. Langium ist darauf ausgerichtet, als Extension verwendet zu werden. Zum Testen wird das Extension-Development Host Fenster verwendet. Dies ermöglicht, die Extension zu testen, ohne diese zu deployen. Bei Finalisierung kann diese deployed und anschließend als Extension heruntergeladen werden. Dadurch kann die DSL getestet und später unkompliziert veröffentlicht werden.

8.3 Nutzer-Umfrage

Um die Nutzerfreundlichkeit zu untersuchen, wurde eine Nutzer-Umfrage erstellt. Diese zielt darauf ab, dass FLEX eigenständig aufgesetzt, die Dokumentation gestartet und anhand

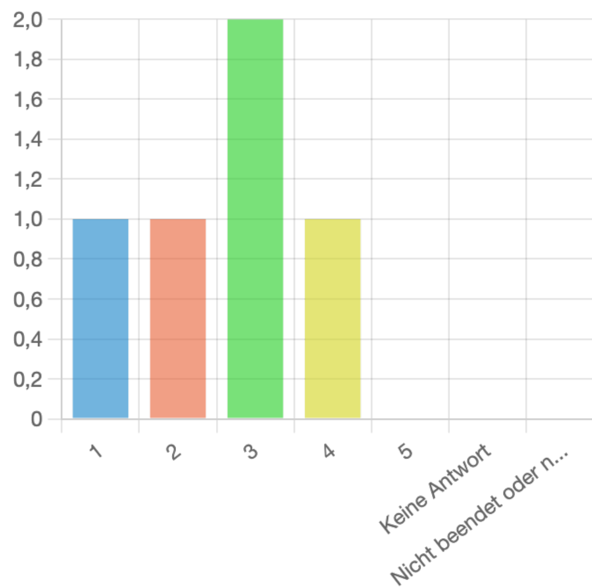


Abbildung 8.5: Wie einfach war die Installation von FLEX? (1 = sehr schwer, 5 = sehr einfach)

dieser sich in die Sprache eingearbeitet wird. Abschließend wird in einer kurzen Umfrage abgeprüft, ob dies unproblematisch funktioniert hat und wo noch Verbesserungsbedarf besteht. Dabei wurde die Schwierigkeit der Installation auf einer Skala von eins (sehr schwer) bis fünf (sehr einfach) abgefragt. Die Ergebnisse sind Abbildung 8.5 abgebildet. Das dabei herausgekommene arithmetische Mittel war 2.6. Die Schwierigkeiten entstanden durch technische Probleme, wie in Abbildung 8.6 abgebildet. Aus den Kommentaren wurde deutlich, dass dabei vor allem ein Problem mit der verwendeten NodeJS Version und Ubuntu 22.04 bestand. Diese waren inkompatibel und erforderten nicht beschriebene Installationsschritte. Nachdem die Umgebung erfolgreich aufgesetzt wurde, wurde die erste Ausgabe als durchschnittlich verständlich bewertet. Die Ergebnisse sind in Abbildung 8.7 abgebildet. Dabei ist die Skala von eins bis fünf angelegt, wobei die eins für „unverständlich“ und die fünf für „sehr verständlich“ steht. Die generelle Nutzung wurde als einfach eingestuft. Auf einer Skala von eins (sehr kompliziert) bis 5 (sehr einfach) war das arithmetische Mittel bei 3.75. Die Ergebnisse sind in Abbildung 8.8 dargestellt. Die Einarbeitungszeit lag durchschnittlich bei 30-60 Minuten, wobei dort auch viel Zeitverlust durch die technischen Probleme entstanden sein wird. Die bestehenden Codebeispiele und die Syntax wurden als leicht verständlich eingestuft. Die Syntax wurde größtenteils als logisch empfunden. In Abbildung 8.9 ist abgebildet, das nur eine Person Stellen in der Syntax als unlogisch beschrieben hat. Es gab keine Fehler, die nicht gelöst werden konnten. Die technischen Probleme beim Aufsetzen lösten allerdings Verwirrung aus. Bei der Auswertung der Features der DSL wurden nahezu alle Features gleichmäßig getestet und keine als nicht funktionierend beschrieben. Es gab die Anmerkung, dass es hilfreich wäre, mehr halluzinierte Leitkegel an verschiedenen Positionen einbauen zu können. Ansonsten

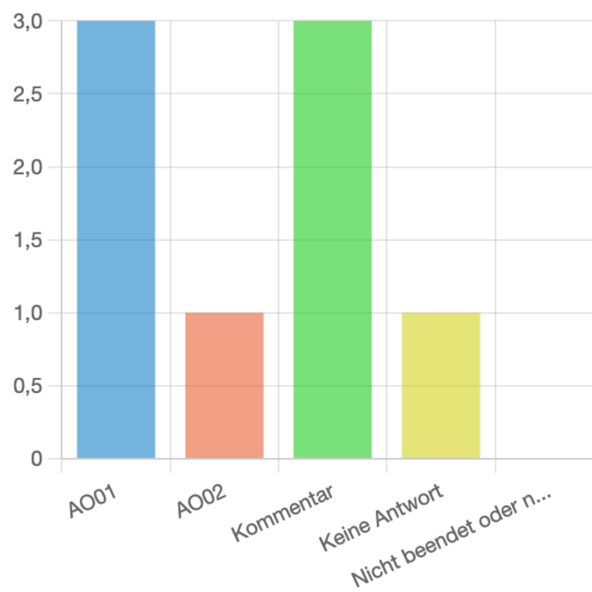


Abbildung 8.6: Gab es technische Probleme bei der Installation? Ja = Blau, Nein = Rot, Kommentar = Grün, Keine Antwort = Gelb

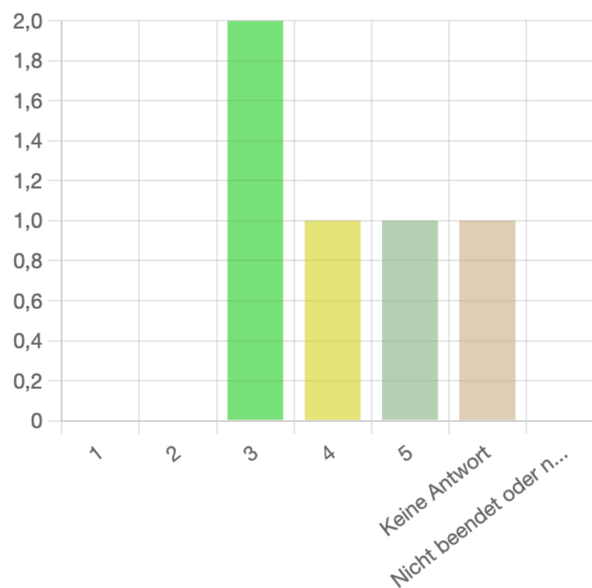


Abbildung 8.7: Wie verständlich war die erste Ausgabe? (1 = unverständlich, 5 = sehr verständlich)

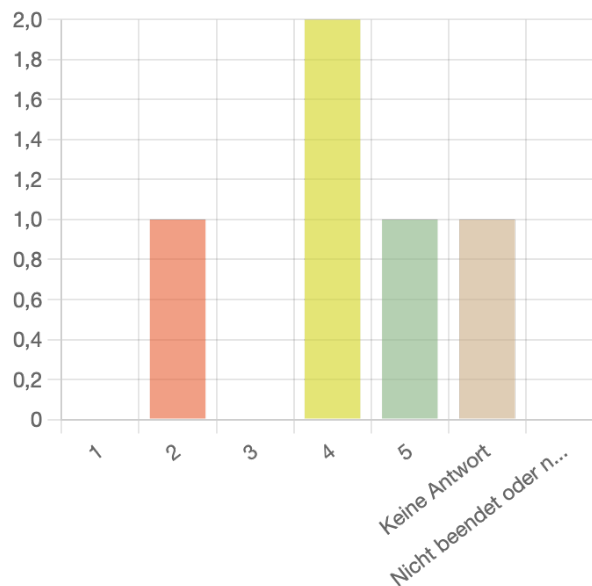


Abbildung 8.8: Wie intuitiv fandest du die Nutzung von FLEX? (1 = sehr kompliziert, 5 = sehr einfach)

wurden keine Features vermisst und es wurde auch keine Notwendigkeit zur Verbesserung der bestehenden gesehen.

Insgesamt war das Feedback dieser Nutzer-Umfrage positiv. Es wurde besonders deutlich, dass die Syntax als leicht verständlich wahr genommen wurde und das größte Problem rein technisch war. Die Konstrukte, um Fehlerverkettungen darzustellen, erscheinen ebenfalls vollständig. Als problematisch stellte sich die Installation aufgrund von Betriebssystem Konflikten heraus. Dies muss in Zukunft gelöst werden. Aus **??** wird deutlich, dass die meisten FLEX verwenden würden. Es wurde dabei, besonders herausgestellt, dass Testen und Validieren eine hohe Relevanz hat und das Verhalten von neu entwickelter Software gerade in der Konstruktionsphase damit gut getestet werden könnte. In der Konstruktionsphase existiert nicht durchgehend eine laufende Testplattform und es ist schwierig das Verhalten in Randfällen zu testen. Es wurde außerdem beschrieben, dass Simulationen meistens zu optimal sind. Diese Herausforderung kann durch FLEX bewältigt werden. Die Simulation kann schnell mit Fehlern angepasst werden. Die Abdeckung von Fehlerszenarien durch die implementierten Konstrukte erscheint vollständig. Es können viele Fälle simpel implementiert und später geeignet für den Executor generiert werden. Durch die Verwendung von Langium ist die Sprache gleichzeitig simpel anpassbar und kann auch in Zukunft weiter entwickelt werden.

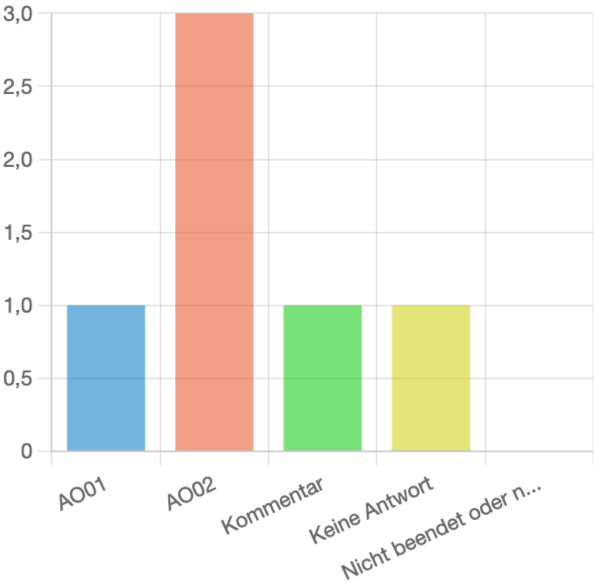


Abbildung 8.9: Gab es Stellen, an denen du die Syntax als unlogisch empfunden hast? Falls ja, wo?, Ja = Blau, Nein = Rot

Kapitel 9

Executor

Um die in der Szenariensprache FLEX beschriebenen Szenarien auszuführen, ist eine Übersetzung der Ausgabe des FLEX-Generators in die Konfigurationsdateien für die Simulatoren und die Fehlerinjektion, sowie das Starten der Simulation nötig. Dafür sind einige manuelle Schritte durch die Nutzer:innen erforderlich, welche fehleranfällig und aufwendig sein können. Falls die FLEX-Szenarienbeschreibung mehr als ein Szenario beschreibt (z. B. aufgrund von Loop-Konstrukten) müssten die Nutzer:innen die genannten Schritte mehrfach ausführen, was den (zeitlichen) Aufwand weiter steigert und die Akzeptanz senken könnte. Daher wurde der Executor entwickelt. Der Executor ist die zentrale Komponente, die alle Elemente nahtlos miteinander verbindet und die Abläufe von der FLEX Szenarienbeschreibung bis zur Ausführung der Simulation automatisiert.

Der komplette Workflow von einer FLEX-Szenarienbeschreibung bis zur Ausführung der Simulation ist in Abbildung 9.1 dargestellt. Zuerst müssen die Nutzer:innen eine FLEX-Szenarienbeschreibung mit z. B. der VS Code Erweiterung für FLEX erstellen. Anschließend wird der Executor aufgerufen, indem der Pfad einer FLEX-Datei übergeben wird. Der Executor ruft automatisch den FLEX-Generator für die übergebene Szenarienbeschreibung auf und erstellt aus den resultierenden Szenarien die Konfigurationsdateien für die Fehlerinjektion und Simulation. Mit diesen Dateien wird schließlich für jedes von FLEX generierte Szenario die Simulation gestartet. Durch den Executors kann somit der gesamte Prozess auf Knopfdruck ausgeführt werden, was den Workflow der Nutzer:innen erheblich vereinfacht und beschleunigt, sowie die Fehleranfälligkeit reduziert. Ebenfalls müssen die Nutzer:innen sich nicht mit den technischen Details der Simulation auseinandersetzen (wie diese bspw. gestartet und konfiguriert werden).

Der Executor ist in Python geschrieben. Die schematische Struktur und der Informationsfluss des Executors ist in Abbildung 9.2 dargestellt. Bei der Entwicklung des Executors wurden die einzelnen internen Komponenten, wie die Anbindungen an die Simulatoren, voneinander getrennt. Somit ist die Erweiterungen durch weitere Komponenten, wie andere Simulatoren oder neue Wege der Fehlerinjektion einfach möglich. Auch Anpassungen einzelner Komponenten können leicht vorgenommen werden. Aktuell sind sowohl Funktionen/Komponenten

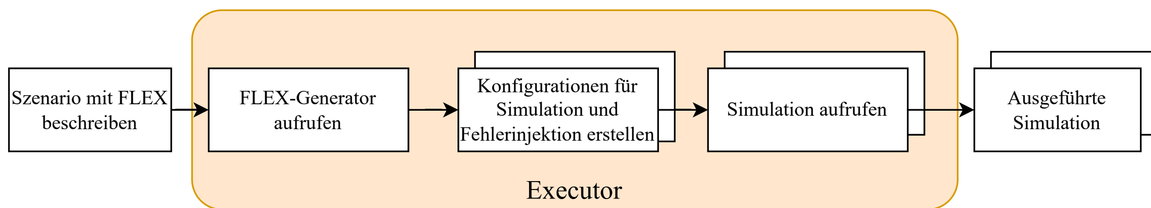


Abbildung 9.1: Workflow von der FLEX-Szenarienbeschreibung bis zur Ausführung der Simulation.

zum Starten von PacSim, FSDS als auch FSSIM im Executor verfügbar. Diese Simulatorkomponenten sind dafür zuständig, je nach Notwendigkeit, simulatorspezifische Konfigurationen zu erstellen und den Simulator auszuführen. Für das Ausführen kann die Komponente bspw. wie für PacSim benötigte Umgebungsvariablen setzen (bei PacSim u. a., um die Karte zu spezifizieren) oder Stoppbedingungen für die Simulation beinhalten (Timeout nach X Sekunden, ...). Falls mehrere Szenarienbeschreibungen aus einer FLEX-Szenarienbeschreibung generiert werden (z. B. durch for-loops), werden die Szenarien bzw. Simulatordurchläufe aktuell sequentiell abgearbeitet. Eine Parallelisierung der Durchläufe wurde nicht implementiert, da mehrere laufende Simulationen auf derselben Hardware aufgrund beschränkter Rechenressourcen nicht möglich waren.

Aus den Ausgabedateien des FLEX-Generators erstellt der Executor die Konfigurationsdatei für die nachrichtenbasierte Fehlerinjektion. In einigen Fällen ist keine 1:1 Übersetzung der beiden Dateien möglich. Dies liegt daran, dass die allgemeinen Fehlerbeschreibungen in FLEX unabhängig von den spezifischen implementierten Faults in der Fehlerinjektion entwickelt wurden. Daher existiert im Executor noch Parsing-Logik, um diese Übersetzung vorzunehmen. Beispielsweise muss der Fehler `STEERING_OFFSET` von FLEX übersetzt werden. In FLEX enthält dieser einen (positiven) Offset und eine Seite, zu welcher der Offset angewandt werden soll. Die Fehlerinjektion hingegen erwartet einen vorzeichenbehafteten Offset, welcher implizit schon die Seite des Offsets enthält. Ebenfalls werden die Timestamps der Faults in das für die Konfiguration benötigte Format überführt. Eine Herausforderung stellen dabei sich zeitlich überlappende Fehler dar, welche vom gleichen Fault-Injection-Node verarbeitet werden. Ein solches Beispiel ist in Abbildung 9.3 dargestellt. Sowohl `STEERING_OFFSET` als auch `ENGINE_DAMAGE` werden vom Control-Fault-Injection-Node verarbeitet. Damit dieses Timing durch die Fault-Injection ausgeführt werden kann, müssten drei Einträge in der Konfiguration erstellt werden: Von t_1 bis t_2 mit dem `STEERING_OFFSET`, von t_2 bis t_3 mit beiden Faults und von t_3 bis $_4$ mit dem `ENGINE_DAMAGE`. Das FLEX-Zwischenaggregat enthält allerdings nur die einzelnen Faults. Im Beispiel sind dies zwei Einträge: einmal von t_1 bis t_3 für den `STEERING_OFFSET` und t_2 bis t_4 für den `ENGINE_DAMAGE`. Da das Zusammenführen der Fehler in die für die Fault-Injection benötigte Struktur mehr Logik im Executor erfordert hätte, welche die Komplexität gesteigert hätte, wird diese Art von Überlappung zurzeit nicht erlaubt. Dazu wird bereits in FLEX bei Erkennung einer solchen Überlappung eine Warnung ausgegeben.

Nicht alle Fehler werden über die nachrichtenbasierte Methode der Fehlerinjektion (vgl. Unterabschnitt 7.2.4) injiziert. Für Fehler in der Perception wird in FSSIM und PacSim der in Abschnitt 7.3 vorgestellte Track Modifier verwendet. Dazu wird der Track Modifier vom Executor gestartet. Der Executor ruft die Funktionen des Track Modifiers mit dem passenden Verhalten zu dem spezifizierten Szenario aus. Dazu wird im Szenario über FLEX die Basismap angegeben, die verändert werden soll. Diese gibt Executor an den Track Modifier weiter. Auch wird die Anzahl an Leitkegeln angegeben, auf die die Veränderung angewendet werden soll. Die fertige Map wird gespeichert und die Simulation wird anschließend mit dieser gestartet.

Aufgrund des in FLEX gebildeten Kreuzprodukts beim Loop-Konstrukten kann die Anzahl der Szenarien sehr schnell sehr stark ansteigen. Beispielsweise würde aus einer FLEX-Beschreibung zweier Faults mit je 5 Parametervariationen schon 25 verschiedene Szenarien generiert werden. Falls noch ein weitere Fault mit 5 Parametervariationen hinzugefügt wird, resultieren daraus schon 125 Szenarien. Da das Ausführen der einzelnen Szenarien rechen- und zeitintensiv ist, beschränkt der Executor die Liste der Szenarien. Dazu werden aktuell maximal 10 Szenarien pro FLEX-Beschreibung zufällig ausgewählt und ausgeführt. Eine beispielhafte Auswahl der Szenarien mit dieser Strategie und zwei Parametern ist in Abbildung 9.4a dargestellt. Problematisch hierbei ist, dass zufällig ggf. nicht der ganze Parameterraum getestet wird, sondern relativ ähnliche Szenarien. Ein anderer Ansatz, welcher dieses Problem mitigiert, ist Latin Hypercube Sampling. Ein Beispiel dieser Sampling-Strategie ist in Abbildung 9.4b dargestellt. Hier wird der Parameterraum anhand der Anzahl zu ziehender Samples unterteilt und je Unterteilung mindestens ein Sample gezogen. Somit ist eine größere Abdeckung des möglichen Parameterraums möglich. Latin Hypercube Sampling ist im Executor allerdings nicht implementiert, da hierfür der Executor Kenntnis über den Parameterraum haben müsste und dafür die Szenarien selbst interpretieren können müsste.

Der Executor kann einfach in die CI-Pipeline integriert werden, indem das Python-Skript des Executors mit Übergabe einer FLEX-Datei in der Pipeline aufgerufen wird. Dadurch wird ermöglicht, anhand von FLEX-Szenarienbeschreibungen, die einfacher zu verstehen sind als die direkten Konfigurationen der Fehlerinjektion, eine Vielzahl von Fehlerszenarien automatisiert bei Codeänderungen auszuführen und somit die Änderung bewertbar zu machen. Bislang wurde der Executor jedoch nur zur manuellen/visuellen Bewertung der Fehlerszenarien verwendet. Eine automatische Auswertung der Szenarien nimmt der Executor daher nicht vor, sondern müsste extern (z. B. durch Rosbags) geschehen. Ein Skript, welches einen einzelnen PacSim-Simulationsdurchlauf auswertet, indem eine Karte des gefahrenen Pfads geplottet wird, wurde im Rahmen der Projektgruppe bereits entwickelt.

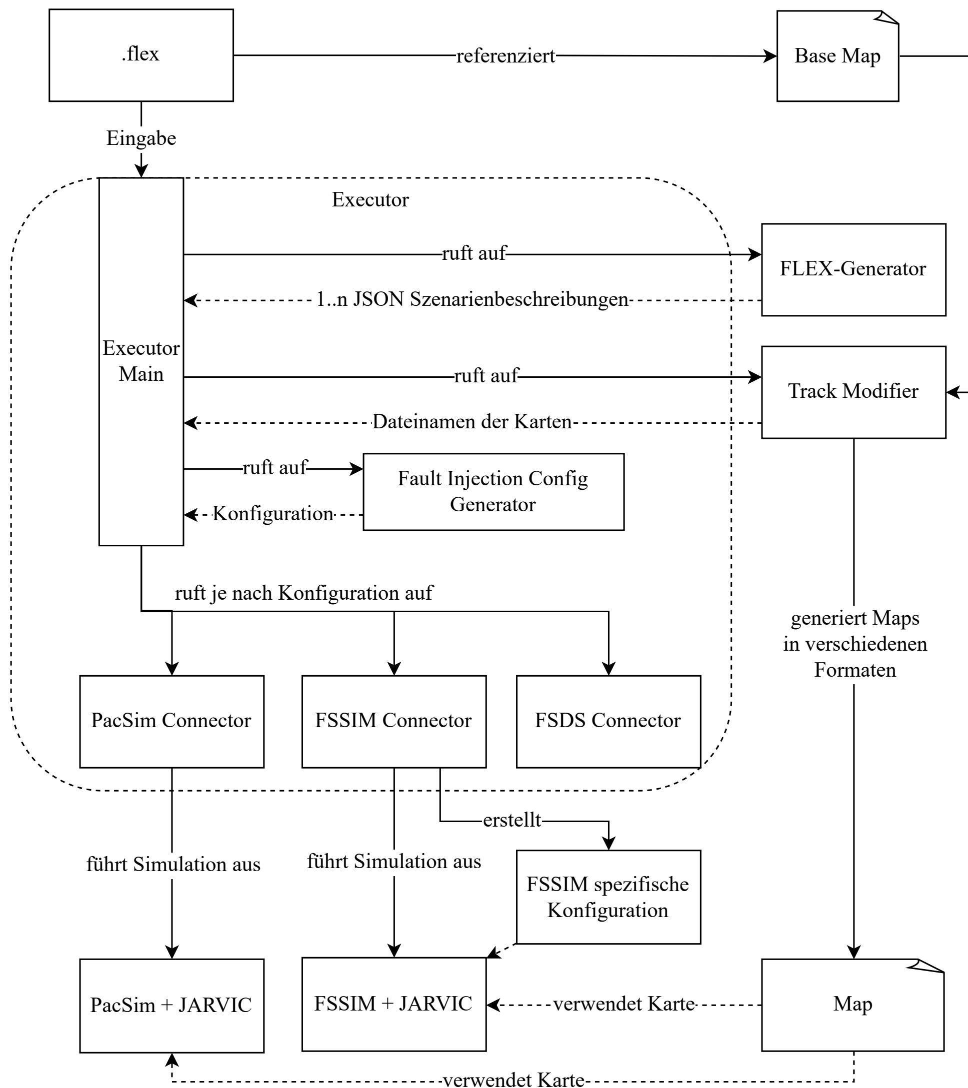


Abbildung 9.2: Schematische Struktur und Informationsfluss des Executors.

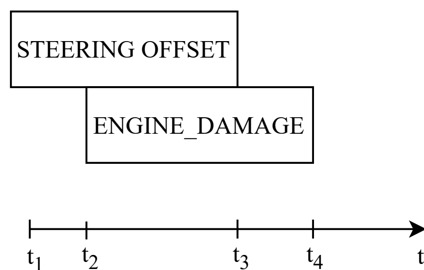
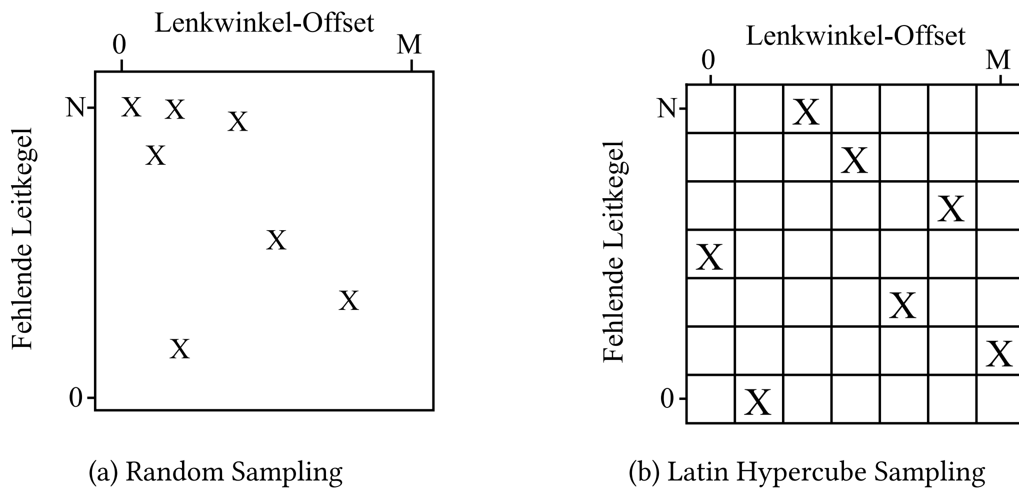


Abbildung 9.3: Zeitlich überlappende Fehler, welche vom gleichen Fault-Injection-Node verarbeitet werden



(a) Random Sampling

(b) Latin Hypercube Sampling

Abbildung 9.4: Beispiele von Samplingstrategien der auszuführenden Szenarien anhand zweier Parameter. Für jede Strategie wurden jeweils 7 Parametermeterkombinationen/Szenarien ausgewählt (gekennzeichnet durch X).

Kapitel 10

Resilienzmaßnahmen

In diesem Kapitel werden die implementierten Resilienzmaßnahmen erläutert. Mit diesen Maßnahmen werden Fehler erkannt und anschließend wird auf diese reagiert. Resilienzmaßnahmen können in zwei Kategorien eingeteilt werden. Bei einer präventiven Resilienzmaßnahme wird früh im Entwicklungsprozess präventiv nach Fehlern gesucht. Sie reduzieren dadurch das Risiko von Schäden, haben aber nur begrenzte Möglichkeiten, Fehler zu finden. Zu den präventiven Resilienzmaßnahmen zählt eine manuelle Überwachung einer Simulation und die Behebung beobachteter Fehler. Die Behebung der gefundenen Fehler kann in Abhängigkeit der Schwere des Fehlers auch automatisch erfolgen. Eine manuelle Suche nach Fehlern ist zeitaufwändig, sodass die automatische Suche von Fehlern in diesem Abschnitt im Fokus steht. Dazu gehören vor allem Methoden der Softwarequalität. Während der Rennwagen in der Realität fährt, greifen präventive Maßnahmen zu spät, sodass im Fehlerfall reaktive Resilienzmaßnahmen erforderlich sind, um Schaden zu begrenzen oder zu verhindern. Diese Resilienzmaßnahmen werden zu Beginn eines Rennens gestartet und suchen während der Fahrt nach Fehlern. Bei einer Erkennung eines Fehlers wird die dazu passende Fehlerbehandlung gestartet.

Im Folgenden werden zunächst die präventiv getroffene Resilienzmaßnahmen vorgestellt. Dazu zählt die Komplexitätsreduktion durch Fahren ohne SLAM (Abschnitt 10.1), die Erstellung von Unit Tests (Abschnitt 10.2), die Erweiterung der statischen Codeanalyse (Abschnitt 10.3) und das Programm TopicParser, das die Konfiguration der nachrichtenbasierten Kommunikation überprüft (Abschnitt 10.4). Anschließend werden die reaktiven Resilienzmaßnahmen zur Fehlerbehandlung beschrieben, zu denen die Behandlung eines Ausfall der SLAM-Node in JARVIC (Abschnitt 10.5), das Erkennen und Reparieren defekter Daten der Perception (Abschnitt 10.6) und die Implementierung von Safety Envelopes (Abschnitt 10.7) gehören.

10.1 Fahren ohne SLAM

In der aktuellen Version von JARVIC wird eine von GET selbst erstellte Implementierung des FastSLAM-Algorithmus verwendet [Mon+02]. Dieser Algorithmus dient dazu, das SLAM

Problem zu lösen. Bei diesem Problem soll eine Karte der Umgebung anhand von Beobachtungen, wie die durch die Kameras oder LiDAR, erstellt werden. Gleichzeitig soll zudem die aktuelle Position des Fahrzeugs in der erstellten Karte festgestellt werden. Vereinfacht funktioniert der Algorithmus so, dass Odometriedaten, Sensordaten und die bislang erstellte Karte verwendet werden. Diese Daten werden mit verschiedenen Versionen der Karte (sogenannte Partikel), ggf. mit zufälligem Rauschen, fusioniert, um neue Versionen der Karten zu erstellen. Die wahrscheinlichsten Versionen der Karten, entsprechend der aktuellen Beobachtungen, werden beibehalten. Somit kann der Algorithmus als Ausgabe die Karte sowie die Position des Fahrzeugs anhand des wahrscheinlichsten Partikels liefern. Anhand dieser Informationen kann anschließend der Track erkannt und eine zu fahrende Trajektorie generiert werden. Um die aktuelle Fahrzeugposition zu bestimmen, ist zudem noch der sogenannte Localizer zwischengeschaltet. Dieser hat die Aufgabe die höherfrequenten Odometriedaten auf die letzte bestimmte Fahrzeugposition des niederfrequent laufenden SLAM-Algorithmus zu integrieren und somit eine möglichst hochfrequent aktualisierte Fahrzeugposition den nachfolgenden Algorithmen (Boundary-Detection, Control, …) bereitzustellen. Der grobe Kommunikationsablauf rund um den FastSLAM-Algorithmus in der aktuellen Version von JARVIC ist in Abbildung 10.1a dargestellt.

Die Verwendung von FastSLAM hat einige Probleme aufgeworfen. Der Algorithmus ist relativ komplex, besitzt viele einstellbare Parameter und ist teilweise schwer nachzuvollziehen, da dieser stark auf Wahrscheinlichkeitstheorie basiert. Ebenfalls ist der Algorithmus selbst durch GET implementiert, und nur ein kleiner nicht mehr bei GET aktiver Personenkreis ist mit dem Algorithmus vertraut. Somit sind Änderungen und Verbesserungen am Algorithmus, sowie dessen Parametrierung, schwierig. Dies hat die Arbeit vor allem bei der Portierung auf ROS2 und den neuen Simulator PacSim vor Herausforderungen gestellt. Der FastSLAM-Algorithmus wurde damals implementiert, als noch mit Kamera- und LiDAR-Daten gefahren wurde und eine Fusionierung dieser Daten notwendig war. Da das neue FS225-Fahrzeug nur noch mit einer Kamera fährt, ist dies nicht mehr notwendig. Ebenfalls ein Problem des Algorithmus ist die Fragilität bei kleinen Parameteränderungen oder kleinen Fehlern in der Perception. Bei kleinen Fehlern können z. B. Leitkegel mehrfach erkannt werden, somit eine ungültige Karte erstellt und die darauffolgenden Algorithmen wie zur Boundary-Detection mit ungültigen Daten versorgt werden. Dies kann schwerwiegende Folgen haben, wie bspw. das Verlassen des Tracks durch die Fehlerkennung.

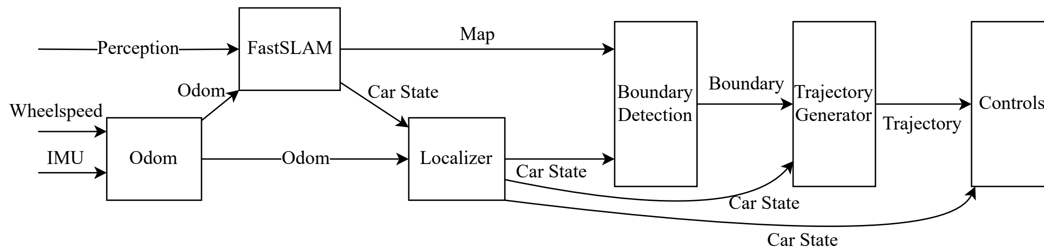
Im Rahmen der Projektgruppe wurde daher ein Ansatz ausprobiert, ohne SLAM zu fahren und somit die Komplexität und Fehleranfälligkeit zu reduzieren. Dabei werden nur die jeweils aktuellen Daten der Perception verwendet. Diese beinhalten jeweils eine Liste an Leitkegeln und deren Kategorie relativ zum Fahrzeug. Anhand dieser Daten kann direkt ohne eine globale Karte die Boundary detektiert und anschließend eine Trajektorie generiert werden. Damit wird eine Art lokale on-the-fly Pfadplannung implementiert. Der Kommunikationsablauf für den Ansatz ist in Abbildung 10.1b dargestellt. Für die Implementierung musste kein neuer ROS-Node implementiert, sondern nur bestehende Nodes und deren Kommunikationsfluss angepasst werden. Für die Tests in der Simulation wurden die geplanten Parameter des FS225-

Fahrzeugs mit Weitwinkelkamera (fast 180 Grad Blickfeld) verwendet. In der Simulation konnten durch diesen Ansatz in der Trackdrive Disziplin gute Ergebnisse erzielt werden, indem mehrere Runden gefahren wurde. Ein Video der Simulation ist hier zu finden[1].

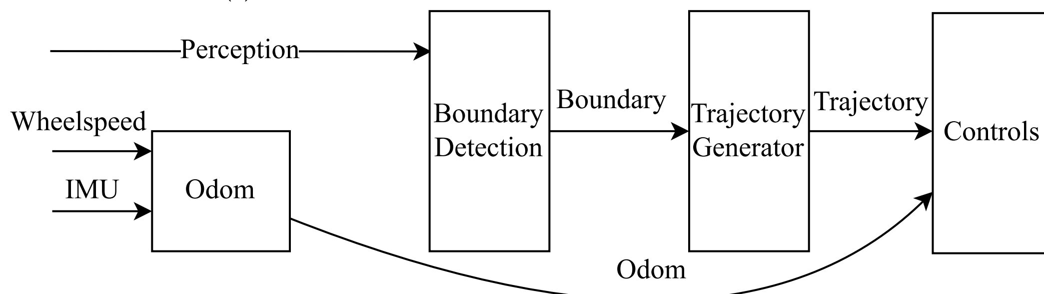
Bei den Tests in der Simulation konnten jedoch auch einige Nachteile des Ansatzes identifiziert werden. Zum einen muss die Perception für diesen Ansatz konstant gute Daten liefern, damit keine ungültige Trajektorie generiert wird. Falsch oder nicht erkannte Leitkegel können schnell zu einem Problem werden. Um die Zuverlässigkeit der eingehenden Perceptiondaten zu steigern, kann ein ebenfalls im Rahmen der Projektgruppe entwickeltes Tool verwendet werden, welches falsche Erkennungen bzw. Karten korrigiert. Dieses Tool wird in Abschnitt 10.6 vorgestellt. Es ist primär für die Korrektur der FastSLAM-Karte entwickelt worden. Das allgemeine Funktionsprinzip ist theoretisch aber auch auf das Fahren ohne SLAM anwendbar, indem es zwischen die Perception und der Boundary Detection geschaltet wird und z. B. fehlende Leitkegel hinzufügt oder falsch erkannte Leitkegel korrigiert. Ebenfalls problematisch sind Spitzkehren, da beim Einfahren in die Kehre ggf. nicht deren kompletter Verlauf im Bereich der Perception liegt und an einem Rand keine Leitkegel erkannt werden. Ein solches Beispiel ist in Abbildung 10.2 dargestellt. Da das Fahrzeug jedoch weiter der bereits geplanten Trajektorie folgt und die Kurve einleitet, auch wenn die Trajektorie nicht richtig anhand der Leitkegel angepasst werden kann, wird die Kurve bei ausreichend geringer Geschwindigkeit trotzdem richtig durchfahren. Ab einem gewissen Zeitpunkt in der Kurve werden wieder auf beiden Seiten Leitkegel erkannt und die Trajektorie wird korrekt angepasst. Dies konnte im getesteten Fall durch das große Blickfeld von fast 180 Grad kompensiert werden, sodass dieser Fall erst sehr kurz vor Spitzkehren auftritt. Ebenfalls basiert der aktuelle Rundenzähler von JARVIC, welcher die Mission stoppt, sobald die geforderte Rundenanzahl gefahren wurde, auf einer globalen Karte, sodass dieser nicht mehr eingesetzt werden kann. Seitens GET ist ein Rundenzähler bereits in Arbeit, der nur anhand der Perception die Runden zählt, sodass dieser auch mit dem SLAM-losen Ansatz einsetzbar wäre.

Zusammenfassend konnte mit dem SLAM-losen Ansatz die Komplexität der Fahrsoftware wesentlich verringert werden. Jedoch muss nun die Perception eine hohe Zuverlässigkeit aufweisen, damit gute und sichere Ergebnisse erzielt werden. Diese Zuverlässigkeit ist aber voraussichtlich durch die starke Fokussierung der Fahrsoftware in der aktuellen Saison auf Vision-Only, der Kamera mit weitem Blickfeld und dem vorgeschalteten Tool zur Korrektur möglicher Fehler der Perception erreichbar. Seitens GET wurde sich für die Saison 2025 jedoch gegen diesen Ansatz entschieden, sondern für einene Stateless-SLAM oder einen auf dem GraphSLAM-Algorithmus basierenden bestehenden SLAM-Implementierung.

[1] https://tu-dortmund.sciebo.de/s/8AFIMpYlHcU1BXe

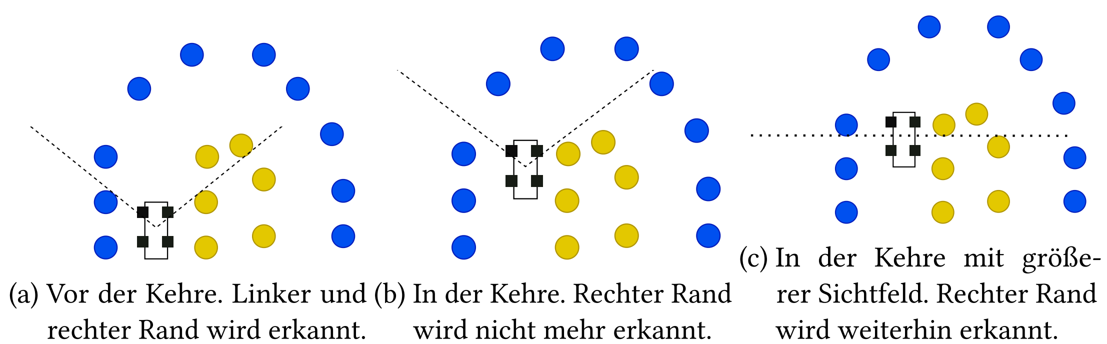


(a) Aktueller Kommunikationsablauf mit FastSLAM.



(b) Kommunikationsablauf ohne SLAM-Algorithmus.

Abbildung 10.1: Kommunikationabläufe mit und ohne SLAM Algorithmus.



(a) Vor der Kehre. Linker und rechter Rand wird erkannt.

(b) In der Kehre. Rechter Rand wird nicht mehr erkannt.

(c) In der Kehre mit größerer Sichtfeld. Rechter Rand wird weiterhin erkannt.

Abbildung 10.2: Verhalten in Spitzkehren ohne SLAM.

10.2 Unit Tests

Unit Testing ist eine Methode, um das erwartete Verhalten von Software – aufgeteilt in ihre Komponenten – zu überprüfen. Dabei können nur die Fehler gefunden werden, die getestet werden. Darüber hinaus kann keine Aussagekraft über die Funktionsfähigkeit des gesamten Systems getroffen werden. Daher kann Unit Testing nur ein Teil der Resilienzmaßnahmen sein. Da frühzeitig gefundene Fehler schnell und kostengünstig behoben werden können, rechtfertigt sich ihre Anwendung und wurde in JARVIC umgesetzt [GS17].

Bei ROS bietet es sich an, Nodes einzeln zu testen. Dazu bietet ROS mithilfe zusätzlicher Pakete, wie zum Beispiel rostest[2], die Möglichkeit, Nodes mittels Unit Tests zu überprüfen. Jede Node erhält dabei ihre eigene Test-Suite mit bestenfalls mehreren Testcases. Über spezielle Launchdateien kann eine Test-Suite unabhängig von dem verwendeten Framework zum Testen gestartet werden. Unter Ausnutzung vom ROS-Nachrichtensystem wird neben dem Testen über einen direkten Aufruf von Funktionen auch das Testen über Nachrichten auf Topics ermöglicht, die nachfolgend erklärt und verglichen werden.

Testen mittels direktem Funktionsaufruf Beim Testen von Funktionen wird eine Funktion aufgerufen, wobei ihr bestimmte Parameter übergeben oder die Attribute der zugehörigen Klasse verändert werden. Das Ergebnis des Aufrufs wird mit dem erwarteten Wert verglichen. In JARVIC kommunizieren Nodes über das Nachrichtensystem von ROS. Nodes fehlt es an Zugriffsmethoden zu den in ihnen gespeicherten Informationen, da ihre Informationen von außerhalb einer Node über Nachrichten verschickt werden und innerhalb einer Node die entsprechenden Variablen in der Regel nur über einen direkten Zugriff verändert werden. Zum Testen über diesen Ansatz können die fehlenden Methoden und Zugriffsrechte [Goo24] ergänzt werden.

Nachrichtenbasiertes Testen Test-Suiten können Nachrichten über Topics an die zu testende Node senden. Gleichzeitig können sie Subscriber erstellen, um die Nachrichten zu empfangen, die die zu testende Node veröffentlicht. Auf diese Weise kann ein Test Case Nachrichten auf den entsprechenden Topics versenden, auf die Antwort der Node warten und bei erfolgter Antwort diese mit ihrem Sollwert vergleichen.

Listing 10.1 ist ein nachrichtenbasierter Test dargestellt. Dabei ist eine Test-Suite für die Node LineDetector dargestellt. Zwecks Übersichtlichkeit gibt es nur einen Test Case mit dem Namen `test_line_detector`. Die Node implementiert die Erkennung der Ziellinie in einer Rennstrecke der Acceleration-Disziplin. Im Konstruktor der Test Suite werden die Parameter aus einer separaten Datei geladen und die passenden Publisher und Subscriber erstellt (Z. 4 – 6), damit die Test Suite und das Testobjekt miteinander kommunizieren können. Über einen

[2] http://wiki.ros.org/rostest

Aufruf der Callback-Funktion (Z. 11 – 14) werden Nachrichten von LineDetector empfangen. Eine empfangene Nachricht wird nur dann gespeichert, wenn sie sich von einem vorher festgelegten Wert unterscheidet. Damit wird erreicht, dass eine Nachricht behalten wird, die die kürzlich veröffentlichte Nachricht eines Test Cases berücksichtigt. Denn in der Regel dauert die Verarbeitung innerhalb einer Node länger als das Intervall zwischen zwei Nachrichten. In diesem Beispiel erstellt der Test Case eine für die Acceleration-Disziplin passende Map und sendet diese an die Node (Z. 12). Im Anschluss wird auf die Verarbeitung der gesendeten Map gewartet, der mit dem Erhalt einer neuen Nachricht beendet wird (Z. 14, 19 – 22). Schließlich wird überprüft, ob der mitgeteilte Endpunkt mit den Erwartungen übereinstimmt (Z. 24).

Das Testen mittels direktem Funktionsaufruf kann tiefer in die Struktur einer Node eindringen und zusätzlich Werte abfragen, die nicht über Nachrichten veröffentlicht werden. Dies bietet sich vor allem für Nodes an, die intern mehrere Berechnungen durchführen, dessen Ergebnisse nicht unmittelbar in Nachrichten erhalten sind. Nachrichtenbasiertes Testen hat den Vorteil, dass eine Veränderung der internen Struktur einer Node ohne die Anpassung der betroffenen Test Cases auskommt. Darüber hinaus können über diese Methode Test-Suiten auch Programmiersprachen-agnostisch geschrieben werden, sodass zum Beispiel eine Klasse in C++ auch durch eine Python-Tests-Suite getestet werden kann. Das Testen mittels Python ermöglicht ein schnelles Schreiben des benötigten Codes. Der Trade-off ist eine längere Ausführungszeit [Rub+12]. Da eine verlängerte Ausführungszeit der Unit Tests keinen negativen Einfluss auf die eigentliche Laufzeit hat, wird für den Großteil der Test-Suiten diese Methode verwendet, um eine schnellere Entwicklung zu ermöglichen.

In JARVIC bestand bereits eine Grundkonstruktion zum Starten von Unit Tests, wobei keine Test Cases implementiert wurden. Auf Basis dieser Struktur wurden zum Zeitpunkt des Schreibens dieses Berichts für 20 der insgesamt 70 Nodes jeweils eine Test-Suite erstellt, bei denen hauptsächlich mittels Publisher und Subscriber getestet wird. Es gibt auch Test-Suiten, die über den direkten Funktionsaufruf testen. Sie dienen als Prototyp für das Testen von Nodes, deren Funktionalität über das nachrichtenbasierte Testen nicht ausreichend geprüft werden kann und bei dem das Aufrufen von Funktionen notwendig ist. Dies kann z. B. der Fall sein, wenn mehrere komplexe Berechnungen durchgeführt werden, dessen Ergebnisse nicht direkt aus den gesendeten Nachrichten ersichtlich sind.

Auf ROS2 wurden drei Test-Suiten migriert. Neben grundlegenden API-Änderungen sind hauptsächlich Änderungen am Starten der Test-Suiten aufgetreten, sodass die Launchdatei in der Datei der jeweiligen Test-Suite enthalten ist. Eine Verbesserung an dieser Struktur ist, dass über das aktualisierte Format der Launchdatei zunächst die zu testende Node gestartet werden kann und erst nach erfolgtem Start die Test-Suite ausgeführt wird. Dies verhindert, dass ein Test Case Bedingungen überprüft, ohne dass eine benötigten Node noch nicht vollständig gestartet ist. Allerdings kommt dies auch zu dem Nachteil, dass Test-Suiten in C++ nicht umfassend unterstützt werden, sodass dort ein Umweg notwendig ist. Zum Zeitpunkt des Schreibens wurde keine Migration von C++ Test-Suiten erfolgreich durchgeführt.

```python
class LineDetectorTest(TestCase):
    def __init__(self, methodName="runTest"):
        super().__init__(methodName)
        map_topic, end_point_topic = self.load_parameters()
        self.pub_map = Publisher(map_topic, Map)
        self.sub_end_point = Subscriber(end_point_topic, self.callback)
        self.default_value = Point()
        self.stop_waiting = False
        self.actual_value = None

    def callback(self, point):
        if self.default_value != point:
            self.actual_value = point
            self.stop_waiting = True

    def test_line_detector(self):
        self.pub_map.publish(Map(...))

        iterations = 0
        while (not self.stop_waiting) and iterations < 10:
            rospy.sleep(0.5)
            iterations += 1

        self.assertEqual(self.actual_value, Point(...))
```

Listing 10.1: Gekürzte Test-Suite der Node LineDetector.

Durch das Erstellen von Test Cases wurden bereits an zwölf Stellen die Softwarequalität verbessert. Die Verbesserungen variieren von simplen Veränderungen der Formatierung, die nicht mitgezählt wurden, hin zu Behebungen von Fehlern, die indirekt das Fahrverhalten beeinflusst haben. Der Code wurde verständlicher gestaltet und unnötige Komplexität entfernt. Beispielsweise wurde gefunden, dass in einer Node von der in JARVIC festgelegten Kodifizierung der Farbe von Cones abgewichen wird, und dieses vereinheitlicht. Zudem wurde eine fehlerhafte Zuordnung von Vektorkomponenten identifiziert und beseitigt. Außerdem wurde ein Fehler beim Lesen von Parametern behoben. Dieser führt aufgrund einer älteren YAML-Spezifikation dazu, dass Zahlen in wissenschaftlicher Notation als Strings erkannt werden, wenn ihnen ein Dezimalpunkt fehlt [Glä18]. Die Liste der gefundenen Fehler zeigt, dass Testen der Software essenziell ist, um die Software-Qualität auf einem hohen Stand zu halten bzw. zu bringen und kann in ihrer beschränkter Aussagekraft die Funktionalität sicherstellen.

10.3 Statische Codeanalyse

Die statische Codeanalyse ist ein Teil der eingesetzten präventiven Resilienzmaßnahmen und ermöglicht das Auffinden von häufigen Fehlern in einem Programm, ohne dieses tatsächlich auszuführen [Lou06]. Im Vergleich zum Unit Testing ist die Analyse der Software vollständig automatisiert, sodass Feedback unmittelbar nach dem Schreiben von Code verfügbar ist. Auch die Aussagekraft der statischen Codeanalyse ist beschränkt. Häufig sind diese Programme universell, sodass ihnen das Wissen der Anwendungsdomäne fehlt.

In JARVIC wurden bislang die Programme cpplint[3] und pycodestyle[4] verwendet, um unter anderem bei Merge Requests die Softwarequalität zu beurteilen. Ihre gefundenen Anmerkungen wurden bisher unterdrückt, sodass diese zum erfolgreichen Mergen einer Merge Request nicht behoben werden müssen. Um bei JARVIC die bestehenden Möglichkeiten zur automatisierten Codeanalyse zu erweitert, wurden im Rahmen der PG folgende Maßnahmen ergriffen:

Behebung von Fehlern Die erkannten, aber unterdrückten Fehler der bestehenden Codeanalyse wurden korrigiert. Enthaltene Änderungen sind beispielsweise eine Beschränkung der Zeilenlänge auf 120 Zeichen und eine Aktualisierung der zu ignorierenden Ordner. Um weitere Fehler zu verhindern, wurde die Option deaktiviert, dass zum erfolgreichen Abschließen einer Merge Request die CI fehlschlagen darf. Diese Änderungen führten dazu, dass die CI-Pipeline ohne Warnungen erfolgreich beendet wird.

Compilerargumente GET racing hat für C++ den Compiler g++[5] so verwendet, dass der Code unter anderem auf unbenutzte oder uninitialisierte Variablen sowie Funktionsaufrufe

[3] https://github.com/cpplint/cpplint

[4] https://github.com/PyCQA/pycodestyle

[5] https://gcc.gnu.org/

mit fehlenden Rückgabewerten untersucht wird. Weitere Argumente wurden ergänzt, um ISO-C++-Standard einzuhalten und zusätzlich weitere Fehler erkennen zu können, wobei die Version des verwendeten Standards zwischen einzelnen Nodes variiert. Zum Beispiel werden Funktionen auf nicht verwendete Parameter oder konditionale Anweisungen auf leeren Inhalt überprüft.

Clang-Tidy Als weiteres Tool zur statischen Codeanalyse wurde Clang-Tidy[6] für C++ hinzugefügt. Unter anderem empfiehlt es, Variablen als konstant zu deklarieren, wenn sie nicht weiter verändert werden, oder warnt bei Verwendung von Gleitkommazahlen bei einer Zuweisung zu einer Integer-Variable.

Formatierung Ein einheitlicher, festgeschriebener Stylestandard wurde für C++ festgelegt, sodass Inkonsistenzen in der Formatierung vermieden werden und die Lesbarkeit nicht verschlechtert wird. Dies wird mit dem Programm ClangFormat[7] umgesetzt, wobei das Format an den Google Styleguide[8] angelehnt ist.

In JARVIC gibt es auch einige Nodes, die in Python geschrieben sind. Python ist durch seine dynamische Typisierung schwer zu analysieren. Um eine gründliche Analyse zu ermöglichen, sind Type Hints [RLL15] hilfreich. Sie zeigen den statischen Typ einer Variable an. Die Einführung von Type Hints in allen Dateien von JARVIC ist aufwändig, da der dynamische Typ einer Variable nicht immer offensichtlich ist. Zudem plant GET racing, aufgrund der besseren Performance, Python-Nodes in C++ umzuschreiben. Deswegen wurde sich gegen die Einführung von Type Hints und weiterer Analysewerkzeuge für Python entschieden. Die Vereinheitlichung des Build-Prozesses wird in Zuge der ROS2 Migration über einen Leitfaden zur Migration von Nodes angestoßen, sodass unter anderem mittels Makros die Lesbarkeit verbessert und nur der C++-Standard in der Version 17 verwendet wird.

10.4 TopicParser

Universelle Programme zur statischen Codeanalyse kommen ohne Anwendungswissen aus und können kein tiefgreifendes Verständnis für spezielle Anwendungsfälle erlangen. Daher können die jeweils typischen Fehler eines Anwendungsfalls nicht durch diese Programme berücksichtigt werden. Bei JARVIC ist ein typischer Fehlerfall die inkorrekte Konfiguration von Topics in Nodes. Dabei werden Parameter aus dem Code in eine YAML-Datei pro Node ausgelagert. Mit dem Programm Roslaunch können diese Konfigurationsdateien über Launchdateien in ihre jeweilige Node eingebunden und gestartet werden. Dieses Auslagern ermöglicht es, Parameter zu verändern, ohne nach jeder Änderung die betroffene Node neu kompilieren zu müssen.

[6] https://clang.llvm.org/extra/clang-tidy/
[7] https://clang.llvm.org/docs/ClangFormat.html
[8] https://google.github.io/styleguide/cppguide.html

Ein Auszug einer Konfigurationsdatei ist in Listing 10.2 zu sehen. Darin sind die Benennung der verwendeten Topics und die Rate, mit der die Node immer wieder aktiviert wird, eingestellt. Durch den Key eines Parameters kann dessen Wert zur Laufzeit in die Node geladen werden. Keys bilden die hierarchische Struktur der Parameter über Schrägstriche ab, sodass in diesem Beispiel der Key des ersten Parameters `topics/sub_lidar_cluster_topic_name` ist. Dieser wird über die API von ROS in dem Code der Node angegeben, um `/jarvic/estimation/map` als dessen Wert zu erhalten.

Eine fehlerhafte Konfiguration bei der Benennung der Topics in einer Node führt dazu, dass die Kommunikation zwischen dieser und weiterer Nodes, die miteinander über Topics verbunden sind, gestört ist. Dies kann teilweise schweres Fehlverhalten verursachen. Ein einfacher Tippfehler in einer Konfigurationsdatei kann bereits ausreichen, um Probleme auszulösen. Als Resilienzmaßnahme gegen eine Störung durch die fehlerhafte Benennung von Topics in den Konfigurationsdateien dient das Programm TopicParser.

Mithilfe von TopicParser wurden sechs Konfigurationsfehler gefunden. Da zuvor ein größeres Umbenennen der Topics durch GET racing bei JARVIC stattgefunden hat, wurden Topics gefunden, die der alten Schreibweise entsprechen, und entsprechend angepasst. TopicParser wurde in die CI-Pipeline integriert, sodass nah am Entwicklungsprozess fehlerhafte Konfigurationen gefunden werden können.

Um Fehler in den Konfigurationsdateien zu erkennen, können verschiedene Möglichkeiten gewählt werden, die entweder eine zentrale Struktur der zum Laden der Parameter einführen oder die dezentrale Struktur beibehalten, wobei auch ein hybrider Ansatz möglich ist.

Zentraler Ansatz Mit einer Datei, die die Benennung aller Topic übernimmt und in alle Nodes geladen wird, werden solche Fehler durch die vorgegebene Struktur in der Konfigurationsdatei vermieden. Für JARVIC existiert bereits eine solche Datei, in der die Höchstgeschwindigkeit eingestellt werden kann. Die Zentralisierung zwei weiterer Parameter ist geplant.

Hybrider Ansatz Statt einer zentralen Datei können auch mehrere Dateien für jeweils unterschiedliche Themenbereiche verwendet werden. Zum Beispiel sind in einer Datei die Topics zur Steuerung des Fahrzeugs zusammengefasst und in einer weiteren sind die Topics für die Visualisierung enthalten.

Dezentraler Ansatz Die dritte Möglichkeit behält die dezentrale Struktur in allen Nodes bei der Konfiguration ihrer Parameter bei. Bei diesem Ansatz werden die Fehler nicht durch die Struktur vermieden, sondern es wird ein Programm benötigt, das die Benennung der Topics überprüft. Dieses Programm hat das Ziel, Änderungen anzumerken, bei denen die die Benennung der Topics fehlerhaft ist.

Mit zunehmender Anzahl an Topics wird eine zentralisierte Konfigurationsdatei unübersichtlicher. Der hybride Ansatz behebt diesen Nachteil, aber dies sorgt beim Finden und Einbinden eines gewünschten Topics für einen höheren Aufwand, wenn die Gruppierung

```
1 topics:
2   sub_lidar_cluster_topic_name: /jarvic/estimation/map
3   pub_end_point_topic_name: /jarvic/planning/end_point
4 node_rate: 50 # [Herz]
```

Listing 10.2: Gekürzte Konfigurationsdatei der Node LineDetector.

nicht klar geregelt ist. Beispielsweise ist dies der Fall, wenn ein Topic nicht eindeutig in einen Bereich kategorisiert werden kann. Diese beiden Ansätze würden die aktuelle Struktur zum Laden von Parametern grundlegend verändern. Dafür ist es erforderlich, neben den Konfigurationsdateien auch die Launchdateien anzupassen. In ihnen wird die Einbettung der Konfigurationsdateien angegeben. Zusätzlich wird eine doppelte Struktur eingeführt, da jede Node durch diese Ansätze mehrere Konfigurationsdateien benötigt. Neben der zentralisierten oder gruppierten Parameter können Nodes auch Parameter haben, bei denen es sich lohnt, die bestehende dezentrale Struktur beizubehalten. Dazu gehört zum Beispiel die Aktivierungsrate einer Node. Diesen Parameter hat in der Regel jede Node, wobei sie jeweils unterschiedliche Werte annehmen können. Die Zentralisierung dieser Parameter kommt wegen verschlechterter Übersichtlichkeit nicht infrage. Zudem kommt hinzu, dass bei diesen Ansätzen nicht verhindert wird, falls dennoch die Benennung eines Topics in der Konfigurationsdatei einer Node geregelt wird.

Bei dem dezentralen Ansatz bedarf es keiner Änderung der bestehenden Struktur. Somit bleibt die Parameterbelegung in der jeweiligen Node enthalten. Dies bewahrt die Modularität der Nodes, sodass unter anderem die Namen neuer Topics in der entsprechenden Node hinzugefügt werden, ohne die Ordnerstruktur der Node zu verlassen. Zur Vermeidung von Konfigurationsfehlern wird jedoch ein Programm benötigt, dass diese erkennt. Zur Analyse der Konfigurationsdateien für ein solches Programm werden die Parameter benötigt, die eine Node einbindet. Dabei kann eine Liste der Parameter nicht aus der Launchdatei direkt ausgelesen werden, da das Starten von Nodes und die Einbindung von Parametern über Umgebungsvariablen zur Laufzeit abhängt. Diese Evaluation kann mittels Roslaunch ermittelt werden. Über die Angabe eines weiteren Arguments neben der zu startenden Launchdatei können die Parameter evaluiert werden, ohne dass die in der Launchdatei aufgeführten Nodes gestartet werden.

Da sich für den dezentralen Ansatz entschieden wurde, wurden zunächst Anpassungen getroffen, um das Lesen der Parameter zur Topic-Benennung aus der Konfigurationsdatei zu verbessern. Wie in Listing 10.2 dargestellt, sind dafür die Parameter für die Benennung von Topics unter dem Begriff `topics` zusammengefasst. Des Weiteren gibt ein Präfix gibt an, ob die entsprechende Node auf den verwendeten Topics Nachrichten versendet oder empfängt: Die entsprechenden Präfixe sind `pub` zum Senden einer Nachricht und `sub` zum Empfangen. Demnach hat die Node LineDetector einen Publisher für das Topic `/jarvic/planning/end_point` und einen Subscriber für das Topic `/jarvic/estimation/map`.

```
/est/slam/state has 4 subscribers and 1 publishers
/est/localization/state has 0 subscribers and 1 publishers
/est/slam/state_feedback has 1 subscribers and 1 publishers
/est/slam/map has 1 subscribers and 1 publishers
/est/slam/visual_map has 0 subscribers and 1 publishers
/est/slam/visual_state has 0 subscribers and 1 publishers
/planning/boundary has 1 subscribers and 1 publishers
/con/speed_reference has 1 subscribers and 0 publishers
/con/emulated_control_command has 0 subscribers and 1 publishers (whitelist reason: "Subscribed by FSSIM")
```

Abbildung 10.3: Beispielausgabe des TopicParsers. Drei Topics besitzen Publisher aber keine Subscriber (gelb; Warnung). Ein Topic besitzt einen Subscriber aber keinen Publisher (rot; Fehler).

TopicParser ist in Python geschrieben und benötigt als Eingabe eine Launchdatei und optional eine Datei, die Topics auflistet, die von dieser Analyse ausgeschlossen werden sollen. Solche Ausschlüsse sind zum Beispiel nötig, wenn ein Simulations- oder Visualisierungsprogramm separat gestartet wird. Den betroffenen Topics fehlt daher in der Analyse ein Publisher oder Subscriber. Da es in JARVIC Launchdateien gibt, die nicht alle benötigten Nodes auf einmal starten, wird zudem ermöglicht, mehrerer Launchdateien anzugeben. Über Roslaunch wird die Liste der zu startenden Nodes und ihrer Parameter aus der angegebenen Launchdatei ermittelt. Aus der Gesamtheit der Parameter werden die Parameter zur Benennung von Topics herausgefiltert. Über eine Map mit den Benennungen der Topics als Key werden ihre jeweiligen Nodes als Subscriber oder Publisher festgehalten. Danach wird überprüft, ob jedes Topic genau einen Publisher und mindestens einen Subscriber besitzt. Ein Topic kann demnach nur einen Publisher haben, um inkonsistente Nachrichten verschiedener Nodes zu verhindern. Daher verwendet JARVIC prinzipiell keine Topics mit mehreren Publishern. Mehrere Subscriber für ein Topic sind aber notwendig, um allen Nodes ihre benötigten Informationen zu liefern. Wenn ein Publisher fehlt, wird ein Fehler geworfen. In diesem Fall erhalten Nodes ihre angegebenen Nachrichten nicht. Wenn es keinen Subscriber gibt, erfolgt nur eine Warnung. Dies ermöglicht, dass Topics, die zum Debuggen oder Loggen dienen, nicht als Fehler aufgefasst werden. Fehler oder Warnungen, die ein ausgeschlossenes Topic betreffen, werden unterdrückt. Eine Beispielausgabe des TopicParsers ist in Abbildung 10.3 dargestellt.

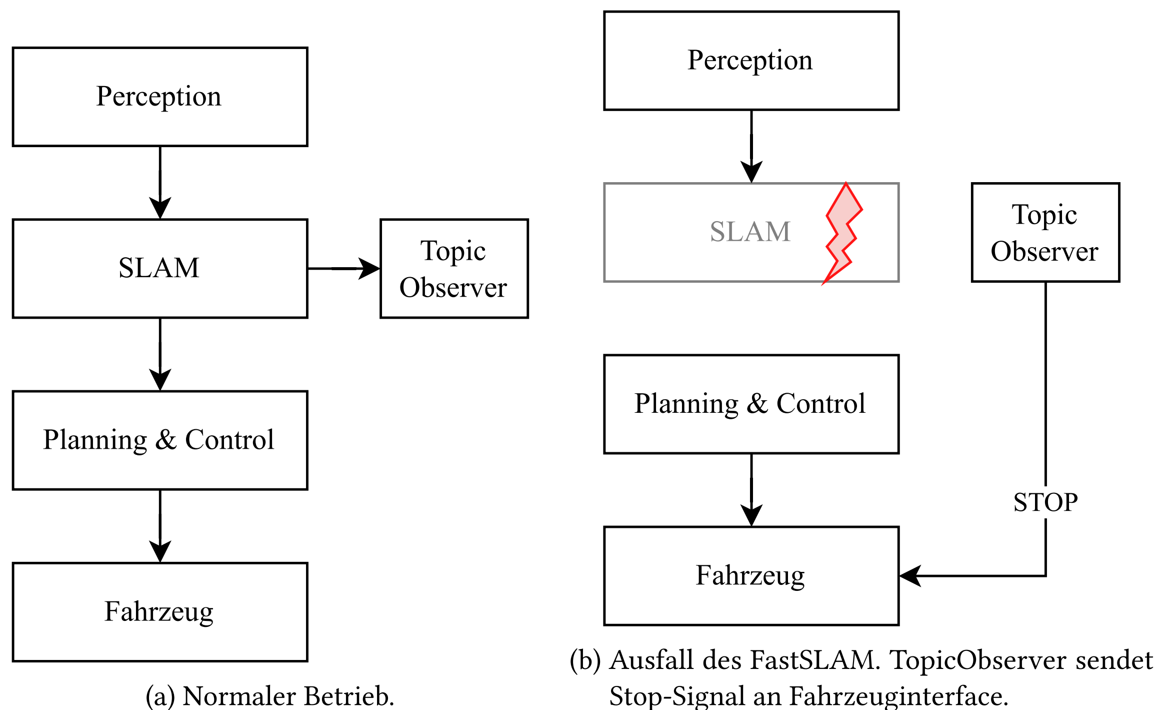
Bei Hardcoding von der Benennung von Topics kann TopicParser eine Kommunikationsstörung nicht erkennen, da der entsprechende Parameter in der Konfigurationsdatei nicht vorkommt. Auch die anderen eingangs erwähnten Ansätze können Hardcoding bei Topics nicht vermeiden. TopicParser ist für ROS Noetic implementiert. Für ROS2 wurde ein ähnliches Konzept für die Ermittlung der verwendeten Konfigurationen erstellt und an GET racing zur Migration übergeben.

10.5 FastSLAM-Ausfall und TopicObserver

Als erster injizierbarer Fehler wurde ein Ausfall der SLAM-Node von JARVIC behandelt, die den FastSLAM-Algorithmus [Mon+02] implementiert. Dabei wird aus den Daten der Perception erst eine Map angefertigt und das Auto in der erstellten Map lokalisiert. Bei einem Ausfall des SLAM-Node wurden in einigen Nodes alte Nachrichten wiederholt verarbeitet, wenn keine neuen Nachrichten vom SLAM veröffentlicht wurden. Dadurch bleibt nach dem Absturz der FastSLAM-Node das Fahrverhalten konstant, insbesondere verbleiben Lenkwinkel und Geschwindigkeit in der letzten Einstellung wie vor dem Ausfall oder es wurde weiter der geplanten Trajektorie gefolgt, die jedoch nicht mehr aktualisiert wurde und ggf. aus der Streckenbegrenzung hinaus führte.

Zur Behandlung dieses Fehlers soll erkannt werden, wenn keine weiteren Daten zur Position des Autos auf der Map und keine Updates der Map selbst veröffentlicht werden. Danach soll der Wagen abgebremst werden, da ohne aktualisierte Map die Trajektorie des Wagens nicht bestimmt werden kann. Dafür wurde der in JARVIC bereits bestehende TopicObserver verwendet. Der TopicObserver ist eine reaktive Resilienzmaßnahme, welcher für kritische Topics in regelmäßigen Intervallen prüft, ob dort Nachrichten veröffentlicht werden. Zu den kritischen Topics können bspw. die Topics für die Kommunikation der Trajektorie aber auch der Map vom SLAM gehören. Der Zeitstempel der letzten ankommenden Nachricht auf diesen Topics wird gespeichert. Wenn keine neue Nachricht auf einem der Topics empfangen wird, wird ab einer gewissen Zeitspanne ein Ausfall der jeweiligen Node angenommen und der TopicObserver sendet eine Fehlerstatus-Nachricht. Dieses Nachricht kann vom Interface des Fahrzeugs empfangen werden und dafür sorgen, dass die eingehenden Steuerbefehle von JARVIC ignoriert werden und das Auto abgebremst wird. Somit wird automatisch ein Notstopp eingeleitet. Um den Ausfall des TopicObserver selbst zu bemerken, veröffentlicht dieser im Normalfall eine Ok-Nachricht als Keepalive. Da noch keine koordinierte Bootreihenfolge in JARVIC existiert, erkennt der TopicObserver erst den Ausfall von Nodes, wenn diese mindestens eine Nachricht auf einem kritischen Topic gesendet haben. Ausfälle in der Bootphase von Nodes (bspw. weil der FastSLAM Node gar nicht erst gestartet werden konnte) können nicht erkannt werden. Zur Erkennung des FastSLAM-Ausfalls wurden dem TopicObserver die beiden Topics des FastSLAM zu Veröffentlichung der Karte und der Position des Fahrzeugs hinzugefügt. Der Informationsfluss bei Ausfall des FastSLAM-Nodes ist in Abbildung 10.4 dargestellt.

Der TopicObserver war zu Beginn der Projektgruppe noch nicht vollständig in JARVIC integriert. So wurde dieser in Simulationen bisher nicht verwendet. Im Rahmen der Projektgruppe wurde der TopicObserver in FSSIM und PacSim integriert, indem die Logik zum Ignorieren der Steuerbefehele von JARVIC und das Betätigen der Bremse in die Simulatorinterfaces hinzugefügt wurde. Ebenfalls wurde durch die Projektgruppe der Node auf ROS2 migriert und die Logik des Nodes vereinfacht. Durch Tests mittels eines in FLEX definierten Fehlerszenarios und des in Kapitel 9 vorgestellten Executors konnte überprüft werden, dass das Fahrzeug in



(a) Normaler Betrieb.

(b) Ausfall des FastSLAM. TopicObserver sendet Stop-Signal an Fahrzeuginterface.

Abbildung 10.4: Verhalten des TopicObserver bei Ausfall des FastSLAM. Pfeile geben den Nachrichtenfluss an.

der Simulation durch die Änderungen bei Ausfall der FastSLAM-Node stoppt. Videos vom Verhalten, einmal ohne und einmal mit TopicObserver, sind hier zu finden[9][10].

10.6 On-The-Fly-Kartenreperatur

Ziel dieses Tools ist es, die erkannte Karte vor der weiteren Verarbeitung und Anwendung des SLAM-Verfahrens auf Fehler zu überprüfen und ggf. zu korrigieren. Dazu wird eine zusätzliche Node vorgeschaltet, die die Karte analysiert, Fehler identifiziert und eine optimierte Version weitergibt. Ursprünglich war geplant, ein KI-basiertes Tool zu entwickeln, das fehlerhafte Bereiche erkennt und korrigiert. Da die trainierten Modelle jedoch Schwierigkeiten hatten, die Streckenstruktur zuverlässig zu erfassen und häufig zufällig Leitkegel hinzufügten, wurde stattdessen ein regelbasiertes Verfahren umgesetzt. Die manuell definierten Regeln orientieren sich an der typischen Streckenstruktur: Auf einer Seite befinden sich gelbe, auf der anderen blaue Leitkegel, mit einem konstanten Abstand zwischen den Reihen. Zur Fehlererkennung und -korrektur werden mehrere aufeinanderfolgende Funktionen angewendet.

[9]Verhalten ohne TopicObserver: https://tu-dortmund.sciebo.de/s/yPZDNZWCKDMtGYH
[10]Verhalten mit TopicObserver: https://tu-dortmund.sciebo.de/s/UqdAWdBuMXkE9Ow

Für die Verarbeitung werden relevante Nachrichten ausgelesen. Da sich das Fahrzeug stets im Ursprung befindet, werden die erkannten gelben und blauen Leitkegel jeweils in Listen sortiert – beginnend mit dem nächsten Punkt zum Ursprung und anschließend ergänzt durch die jeweils nächsten Punkte der Gruppe. Das grundlegende Konzept basiert auf dem Nearest Neighbor-Ansatz. Für jeden Leitkegel wird geprüft, ob sich innerhalb definierter Grenzen eine ausreichende Anzahl an Nachbarn befindet – sowohl innerhalb der eigenen Reihe (gelb oder blau) als auch gegenüberliegend. Typische Abstände zwischen den Leitkegeln wurden anhand mehrerer korrekter Karten bestimmt und dienen als Referenzwerte. Die Korrekturen erfolgen in mehreren Schritten:

1. Doppelte Erkennung: Zunächst wird geprüft, ob zwei Punkte zu nah beieinanderliegen. Dies deutet auf eine doppelte Erkennung desselben Leitkegels hin. In diesem Fall wird zwischen den beiden Punkten interpoliert, der neue Punkt ersetzt die ursprünglichen.

2. Fehlende Leitkegel: Anschließend wird geprüft, ob Leitkegel fehlen. Bei Lücken von bis zu drei Leitkegeln wird durch Interpolation zwischen den benachbarten Punkten die Lücke geschlossen und neue Leitkegel ergänzt.

3. Längere Abschnitte: Für größere Lücken kommt ein Verfahren auf Basis gerichteter Vektoren zum Einsatz. Dieses eignet sich besonders für längere Streckenabschnitte und ist in Abbildung Abbildung 10.5 dargestellt. Zunächst wird der letzte Abschnitt identifiziert, in dem sich gelbe und blaue Leitkegel innerhalb einer definierten Distanzschwelle gegenüberliegen. Dazu wird für jeden Punkt geprüft, ob sich innerhalb dieser Grenze ein nächster Nachbar der jeweils anderen Gruppe befindet. Ist dies nicht der Fall, wird der letzte Punkt der eigenen Gruppe als Referenz verwendet, der noch einen entsprechenden Nachbarn hatte. Ausgehend von diesem Referenzpunkt auf der vollständigen Seite wird ein Richtungsvektor zwischen zwei aufeinanderfolgenden Leitkegeln derselben Gruppe berechnet: einem Punkt, der dem letzten gegenüberliegenden Punkt zugeordnet ist, und einem weiteren Punkt daneben, der direkt gegenüber der Lücke liegt. Anschließend wird vom letzten Punkt auf der Seite mit der Lücke ein Vektor mit gleicher Richtung konstruiert. An dessen Endpunkt wird ein neuer Leitkegelpunkt platziert. Dabei wird zusätzlich berücksichtigt, dass die äußeren Kegel im Vergleich zu den inneren in der Regel einen leicht größeren Abstand zueinander aufweisen. Dieser Unterschied wurde durch eine Mittelwertbildung quantifiziert und fließt als Skalierungsfaktor in die Platzierung des neuen Punktes mit ein. Die Anwendung dieses Verfahrens auf reale Daten ist in Abbildung Abbildung 10.6 dargestellt.

Die vorgestellten Korrekturverfahren wurden teilweise in einen vorgelagerten Korrektur-Node vor dem FastSLAM-Modul eingebunden. Aufgrund technischer Schwierigkeiten und begrenzter Rechenkapazitäten war es jedoch nur eingeschränkt möglich, das Systemverhalten mit dem neuen Code zu beobachten und zu bewerten. Dennoch zeigen die präsentierten Abbildungen einen Proof of Concept und belegen das theoretische Potenzial des Ansatzes. Die entwickelte Idee sowie die zugrunde liegenden Methoden können als Basis für weiterführende Arbeiten genutzt werden. Die Grundidee bestand darin, den fehleranfälligen FastSLAM zu

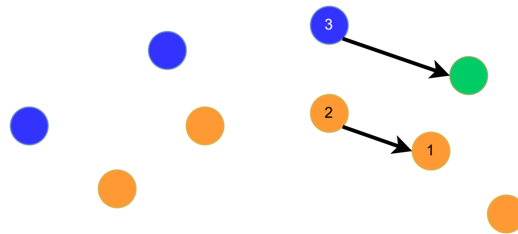


Abbildung 10.5: Verfahren zur Berechnung von fehlenden Leitkegeln bei mehreren fehlenden Leitkegeln. Zunächst wird der Richtungsvektor zwischen Punkt 2 und Punkt 1 berechnet. Ausgehend von Punkt 3 wird dann der gleich Richtungsvektor berechnet, um die Position des grünen, fehlenden Punktes zu berechnen.

ersetzen und stattdessen eine On-the-Fly-Korrektur durchzuführen. Um dieses Konzept zu validieren, muss das Systemverhalten auch ohne FastSLAM überprüft und evaluiert werden.

10.7 Safety Envelopes

Safety Envelopes stellen eine „Überapproximation der erreichbaren Zustände eines Systems unter normalen Bedingungen“ [Tiw+14] dar. Dabei wird zuerst ein Rahmen um diese Zustände gezogen, z. B. dass für die Geschwindigkeit s gilt $-20 \leq s \leq 40$. Während der Ausführung des Systems wird dann geprüft, ob dieser überschritten wird. Da bei normaler Operation dies nicht passieren sollte, kann daraus abgeleitet werden, dass es sich um einen Fehler handeln dürfte. Nun kann eine Fehlerbehandlung ausgeführt werden, welche entscheidet, wie damit umgegangen werden sollte, z. B. könnte stattdessen der vorherige Geschwindigkeitswert beibehalten werden.

In JARVIC ist die Node Control-Command-Master die Komponente, die Nachrichten zu der Fahrzeugsteuerung sendet. Die Fahrzeugsteuerung ist abhängig von dem zu steuernden Vehikel, wie etwa das real existierende Fahrzeug oder ein in FSDS simuliertes Fahrzeug. Aufgrund dessen ist die Node ideal für die Implementierung von Safety Envelopes. In ROS haben Nodes üblicherweise nur eine Funktionalität. Control-Command-Master fügt die Ergebnisse der separaten Berechnung des Lenkwinkels und des Throttles in eine Nachricht zusammen. Das Einfügen von Safety Envelopes in die Node passt thematisch und widerspricht diesem Prinzip somit nicht. Dafür stehen der Lenkwinkel und der Throttle zur Verfügung. Mit ihnen als Grundlage wurden zwei Angriffspunkte ausfindig gemacht.

Ruckartiges Lenken Wenn die Lenkung weit in eine Richtung zeigt und abrupt in die andere wechselt, ist das ein Zeichen von starker Unsicherheit. Dies kann mehrere Ursachen

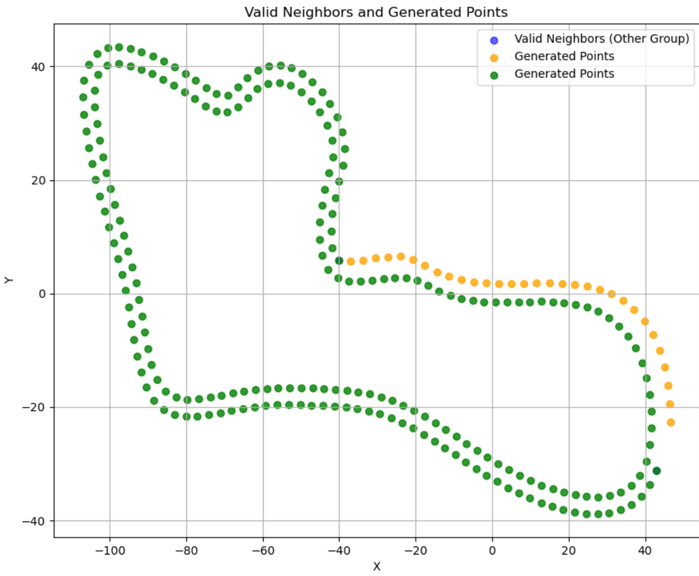


Abbildung 10.6: Anwendung des Verfahrens mit gerichteten Vektoren

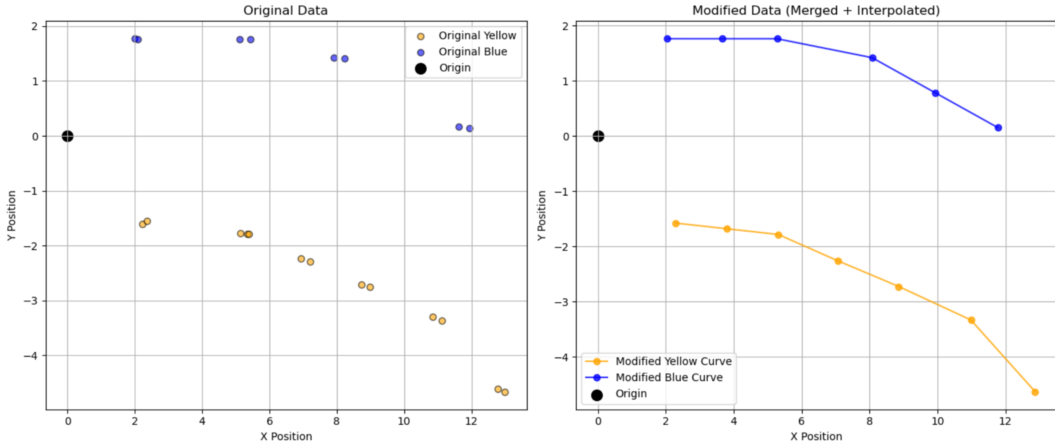


Abbildung 10.7: Links: Originaldaten, Rechts: Nach Entfernung von Duplikaten und mit Interpolation fehlender Punkte

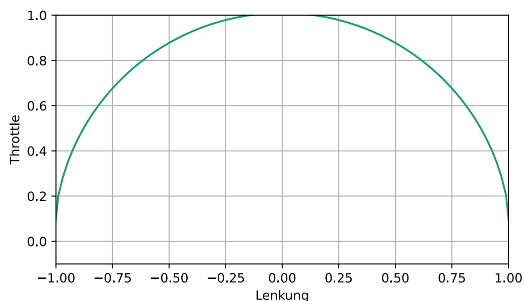
haben. Ein harmloser Grund ist die zu grobe Glättung der Trajektorie in einer Kurve. Problematischer Ursachen sind falsch-positiv erkannte Hütchen oder eine Trajektorie, die aus der Strecke fährt.

Daher wird bei einer Änderung des Lenkwinkels um 0.5 rad (ca. 28°) angenommen, dass eine ruckartige Steuerung vorgenommen wird. Entsprechend sollte der Throttle verringert werden, um das Fahrzeug zu stabilisieren. Die andere Möglichkeit ist die Anpassung des Lenkwinkels. In diesem Fall besteht große Unsicherheit, auf welchen Wert die Lenkung begrenzt werden soll, da die Gefahr besteht, Fahrbahn zu verlassen. Das Gleiche gilt, wenn der Throttle-Wert klein ist. Hier gilt, dass die Wahrscheinlichkeit, in einer Kurve einzugreifen, zu hoch ist. Dementsprechend wird der Throttle-Wert verringert, um die Stabilität des Wagens abzusichern. Der Throttle-Wert liegt in dem Intervall $[1, -1]$, wobei negative Zahlen bedeuten, dass der Wagen bremst. Es wurde festgelegt, dass bei einem Throttle größer als 0.5 0.1 abgezogen wird. Eine Änderung der Werte ist schnell und einfach möglich, sodass sie entsprechend dem Fahrstil angepasst werden können.

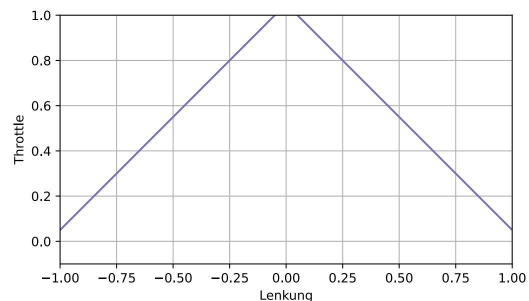
Hoher Throttle-Wert bei starker Einlenkung Eine hohe Beschleunigung in einer Kurve ist gefährlich, denn es kann zur Über- oder Untersteuerung kommen. Demnach gilt es zu verhindern, dass Lenkwinkel und Throttle nicht gleichzeitig hoch sind. Das gilt, wenn der Throttle nahe eins und der Lenkwinkel nahe eins oder minus eins ist. Bremsen beeinträchtigt nicht die Stabilität des Fahrzeugs. Diese Seite des Intervalls des Throttles ist demnach für Safety Envelopes nicht relevant. Für die Einteilung der Wertepaare wurden drei Modelle aufgestellt, die die beiden Werte in eine Relation bringen und die erlauben Werte von den gefährlichen abgrenzen. Diese Modelle sind in Abbildung 10.8 als Graphen dargestellt. Auf der x-Achse wird im Abschnitt von $[-1, 1]$ auf der x-Achse die Lenkung angezeigt, wobei Werte links der y-Achse eine Lenkung nach links anzeigen. Ist der Throttle in dem Bereich $[0, 1]$ auf der y-Achse, zeigt dies eine Beschleunigung an. Negative Werte bedeuten, dass gebremst wird, was in der Grafik nur angedeutet wird. Die Modelle sind in der folgenden Liste aufgeführt:

1. Das erste Modell, das den erlaubten Bereich mit dem Ursprung als Mitte einkreist, ist das liberalste, da mittelhohe Werte in beiden Dimensionen zugelassen werden.
2. Als zweite Möglichkeit kann jeweils eine Gerade auf beiden Seiten der Lenkung gezogen werden, um die Bereiche zu trennen. Eine Verschiebung zur y-Achse oder von ihr weg gibt an, wie viel Lenkung bei hoher Steuerung erlaubt ist.
3. Zwei Kreise mit dem Mittelpunkt bei den gefährlichsten Wertepaaren ist das restriktivste der drei Modelle, bei dem – im Gegensatz zum ersten Modell – das Kreisinnere den nicht erlaubten Wertebereich umfasst. Mittlere Werte werden hier im Vergleich zu den anderen Modellen besonders häufig auch als gefährlich eingestuft.

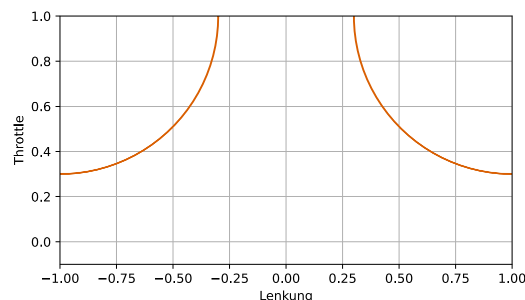
Nach einer Analyse des Fahrstyles von JARVIC wurde festgestellt, dass die Lenkung bei maximalem Throttle-Wert gering ist und mittlere Werte nicht vorkommen. Die Betrachtung



(a) Modell mit einem Kreis



(b) Lineares Modell



(c) Modell mit zwei Kreisen

Abbildung 10.8: Modelle zur Verhinderung von Über- und Untersteuerung.

mehrerer Fahrten, die in Abbildung 10.9 zu sehen sind, hat gezeigt, dass das Fahrverhalten bei positivem Throttle meistens entlang der Achsen verläuft und nur wenig von der y-Achse abweicht. Der maximale Throttle ist zudem auf 0.3 beschränkt. Daher wurde das restriktivste Modell, das dritte, als angemessen beurteilt. Es bietet im Vergleich zu dem zweiten Modell den Vorteil, dass nur ein Wert für die Konfiguration benötigt wird. Zusätzlich ist es in dem ersten und dritten Modell möglich, statt Kreise Ellipsen als Begrenzung zu verwenden. Dies erhöht die Komplexität der Implementierung und der Konfiguration. Der Radius des Kreises für das dritte Modell kann anhand der Daten auf 1.1 gesetzt werden. Die Aussagekraft der für die Entscheidung zu Grunde liegenden Steuerungen von vergangenen Fahrten mit Jarvic ist begrenzt auf die verwendete Karte. Die Telemetrie stammt aus ROS Noetic, wobei die Safety Envelopes in ROS2 implementiert sind. Wegen dieser Einschränkungen bei der Validität wird genügend Abstand eingeplant, sodass der Radius auf 0.87 verringert wurde. Dadurch werden falsch-positive Ereignisse, bei denen die Safety Envelopes noch nicht greifen sollen,

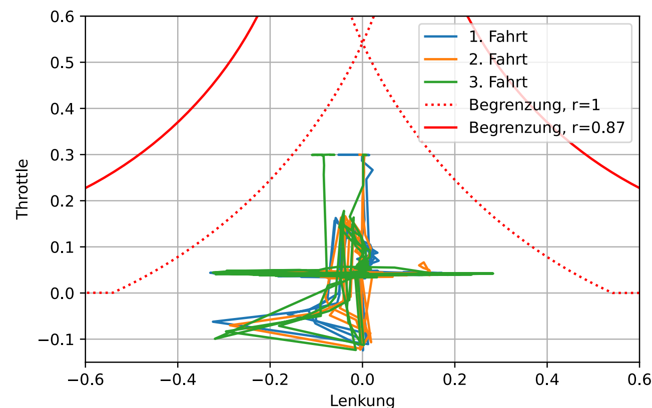


Abbildung 10.9: Steuerung bei mehreren Trackdrive-Fahrten mit JARVIC und zwei verschiedenen Begrenzungen.

vermieden. So wird sichergestellt, dass in der Kurve keine Zeit bei sicherer Geschwindigkeit eingebüßt wird. Der Radius kann wie die Werte aus dem vorherigen Angriffspunkt problemlos an das Fahrverhalten von JARVIC angepasst werden, sodass eine genauere Bestimmung des Radius möglich ist.

Um gegen die Gefahr des Unter- oder Übersteuerns vorzugehen, wird die Steuerung angepasst. Wie auch bei dem Safety Envelop zu ruckartigem Lenken, führt ein Eingriff in die Lenkung zu einer Ungewissheit, ob die Strecke verlassen wird. Daher wird die Lenkung beibehalten und der Throttle auf den durch das Modell vorgegebenen maximalen Wert bei gegebener Lenkung herabgesetzt. Mit dem ausgewählten Modell wird, wenn die Steuerung im Kreisinneren liegt, auf den Kreisbogen senkrecht vertikal verschoben. Dies ist in Abbildung 10.10 zu sehen ist. Darin ist zu sehen, dass das Fahrzeug zunächst gerade ausfährt und dann bei hoher Geschwindigkeit leicht einlenkt. Diese Lenkung ist nach dem Modell akzeptabel. Im Anschluss liegt die Steuerung des Wagens ohne Safety Envelopes im Kreisinneren. Mit Safety Envelopes wird der Throttle angepasst.

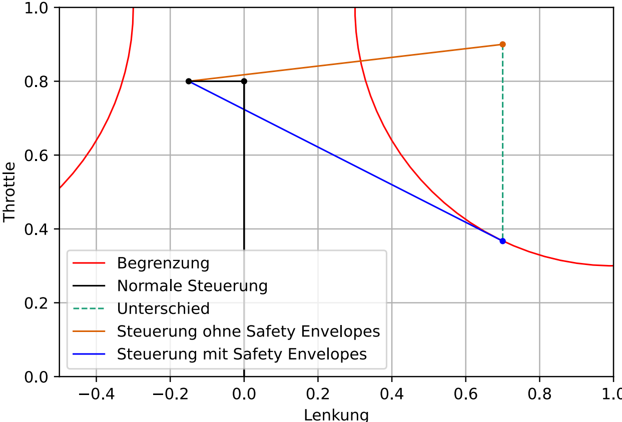


Abbildung 10.10: Mit Safety Envelopes wird exemplarisch die Steuerung verändert.

Kapitel 11

Diskussion

Im folgenden Kapitel werden die Ergebnisse der Projektgruppe zusammengefasst und ein Ausblick auf mögliche Anwendungen gegeben. Der Fokus liegt dabei insbesondere auf dem übergeordneten Ziel der Implementierung von Resilienz. Mitglieder von GET Racing haben die Arbeit der Projektgruppe bereits als zufriedenstellend bewertet. Dies deutet darauf hin, dass die erzielten Resilienzmaßnahmen eine ausreichende Qualität für das Team aufweist.

11.1 Zusammenfassung

Die Projektgruppe eASy ist mit dem Ziel an Resilienz der Fahrsoftware JARVIC von GET Racing zu arbeiten gestartet. Insbesondere war es geplant, eine umfangreiche Fehlerliste von möglichen Problemen aufzustellen und die Software gegen diese Probleme zu schützen. Dabei sollte Risiko minimiert werden. Der Erfolg im Bezug auf die Risikominimierung kann durch Betrachtung des Verlaufes und der Ergebnisse der Projektgruppe analysiert werden.

Im Verlauf der Projektgruppe wurden zahlreiche Herausforderungen bewältigt und wertvolle Erkenntnisse gewonnen. Die erste Hälfte der Arbeit war von Unklarheiten geprägt, doch nach der Formula Student entstand in direkter Zusammenarbeit mit GET Racing ein konkretes Ziel. Dieses bestand in der Entwicklung einer grundlegenden Basis von Resilienzmaßnahmen und Werkzeugen zur Bewertung künftiger Arbeiten. Insbesondere wurden hierfür präventive und behandelnde Formen der Resilienz entwickelt.

Zur Implementierung von Resilienz wurden Fehler systematisch analysiert und definiert, woraus eine Fehlerliste entstand. Ziel war es, diese Liste im Laufe des Jahres vollständig abzuarbeiten. Sie stellte einen zentralen Schwerpunkt der Projektarbeit dar. Basierend auf den definierten Fehlern konnte ein strukturierter Vorgehensplan erstellt werden. Trotz verschiedener Herausforderungen wurde innerhalb eines Jahres eine Vielzahl an Werkzeugen zur Resilienzanalyse und -optimierung entwickelt.

Die gefundenen Fehler mussten für eine gezielte Behandlung klassifizierbar und beschreibbar sein. Zu diesem Zweck wurde die Szenariensprache FLEX entwickelt. FLEX ermöglicht eine

intuitive und benutzerfreundliche Beschreibung verschiedener Fehlerarten. Durch die detailreiche Funktionionalität werden Fehler präzise erfasst. Dabei ist auch die Schwere und der Zeitpunkt des Auftretens definiert. Dies schuf eine solide Grundlage für die Fehlerbehandlung. Jede analysierte Fehlerart kann mithilfe von FLEX beschrieben werden.

Zur Behandlung der beschriebenen Fehler wurden verschiedene Methoden zur Fehlerinjektion entwickelt, die einen zentralen Aspekt der Projektarbeit bildeten. Die Fehlerinjektion ermöglicht es, analysierte Fehler gezielt in einer Simulation zu erzeugen, um deren Auswirkungen zu untersuchen. Die detaillierte Beschreibung eines Fehlers und seine Injektion führen zu einem spezifischen Verhalten in der Simulation, das anschließend analysiert werden kann. Dadurch lassen sich Resilienz- und Präventionsmaßnahmen gezielt testen.

Die Simulationsfunktionalität bezüglich JARVIC wurde durch die Projektgruppe erweitert. Der Schwerpunkt lag dabei auf den zu injizierenden Fehlern und deren Voraussetzungen. Die Migration auf eine aktuelle Version des ROS-Frameworks ermöglichte eine Modernisierung der bestehenden Simulationssoftware. Zudem wurde die Simulation mit FSDS integriert. Dadurch entstand eine erweiterte Simulationsbasis, die mehr Fehlerarten abdeckt als das ursprüngliche System. Besonders die Simulation von Perception-Fehlern, die zuvor nicht möglich war, wurde durch die neuen Simulationsmöglichkeiten realisiert.

Mithilfe dieser Komponenten wurde eine strukturierte Pipeline entwickelt, die eine automatisierte Abwicklung aller Bestandteile sicherstellt. Der eigens entwickelte Executor der Projektgruppe automatisiert die Simulation eines in FLEX spezifizierten Fehlers. Dabei ist sowohl der Simulator als auch die Karte frei wählbar. Dies ermöglicht eine direkte Simulation beliebiger Fehler durch Eingabe einer FLEX-Datei. Die Ergebnisse werden automatisiert über die Pipeline des Executors verarbeitet.

Es wurden bereits einige definierte Fehlerbehandlungen implementiert. Dabei wurden unterschiedliche Maßnahmen wurden für die jeweilige Fehlerkategorie genutzt. Diese Implementierungen dienen als Proof-of-Concept für die Arbeit an Resilienz, da in Zukunft weitere Fehlerquellen identifiziert werden. Zudem erhöhte die Migration auf das neue ROS-Framework die Resilienz des Systems. Während ROS2 weiterhin unterstützt wird, wird ROS1 bald nicht mehr gewartet. Die zukünftige Arbeit auf ROS2-basierte Systeme ist somit zugänglicher als bei einem veralteten Framework.

Diese Arbeitsergebnisse stellen gemeinsam eine Sammlung von Werkzeugen dar. Die Projektgruppe verfolgte das Ziel, eine stabile Grundlage für die künftige Entwicklung von JARVIC und dessen Resilienz zu schaffen. Mithilfe der entwickelten Pipeline lassen sich zukünftige Resilienzmaßnahmen effizienter umsetzen. Die enge Zusammenarbeit mit GET Racing erleichterte die Bewältigung vieler Herausforderungen. Durch die implementierten Werkzeuge und die stabilere Codebasis von JARVIC wurde die Software widerstandsfähiger und bleibt gleichzeitig zukunftsfähig.

11.2 Ausblick

Die Arbeit der Projektgruppe wurde von Mitglieder von GET Racing als hilfreich bewertet, sodass eine zukünftige Nutzung der entwickelten Werkzeuge wahrscheinlich ist. Einige Teilnehmer:innen der Projektgruppe werden ihre Mitarbeit bei GET Racing fortsetzen, wodurch eine reibungslose Übergabe der Ergebnisse sichergestellt werden kann. Durch die Weitergabe der erarbeiteten Ergebnisse an GET Racing werden künftige Projektgruppen darauf aufbauen können. Die stabilere Grundlage soll als Basis für zukünftige Entwicklungen dienen und anfängliche Frustrationen verringern. So wird ein einfacherer Einstieg ermöglicht und die weitere Arbeit erleichtert.

Die entwickelte Pipeline bietet eine erweiterbare Grundlage. Künftige Arbeiten an der Resilienz können durch Modifikationen der bisherigen Ergebnisse gezielt entwickelt werden. Die konkrete Schwierigkeit zukünftiger Arbeiten lässt sich nicht pauschal festlegen. Dennoch zeigen die bisherigen Ergebnisse, dass eine Vereinfachung zukünftiger Aufgaben realistisch ist. FLEX kann erweitert werden, um zusätzliche Fehlerarten abzubilden. Ebenso können weitere Fehlerarten durch Anpassungen der Injektionsmethoden integriert werden. Eine von der Projektgruppe entwickelte Schnittstelle erlaubt zudem eine direkte Verbindung zwischen ROS-basierten Systemen und dem FSDS-Simulator, wodurch vielfältige Erweiterungs- und Anpassungsmöglichkeiten bestehen. Dank des entwickelten Executors bleibt eine automatisierte Untersuchung und Bearbeitung von Fehlerszenarien auch in Zukunft problemlos umsetzbar.

JARVIC wird weiterhin laufend aktualisiert. Die parallel zur Projektgruppe entwickelten Funktionalitäten profitieren von der Migration auf ROS2. Das neue Framework ist stabiler und wird langfristig unterstützt. Damit bleibt JARVIC auch in Zukunft auf einer aktuellen und stabilen Basis erhalten. Die Arbeit an Resilienz ist ein kontinuierlicher Prozess, da neue Störungen stets auftreten können. Die Ergebnisse der Projektgruppe sind daher langfristig relevant und bieten eine wertvolle Grundlage für zukünftige Arbeiten. In der aktuellen Form existieren nur wenige direkte Implementationen von Resilienzmaßnahmen. Die Implementation von weiteren Fehlerbehandlungen ist eine gute Möglichkeit das Konzept zu erweitern.

Neben der Arbeit an JARVIC bieten sich weitere künftige Anwendungsmöglichkeiten der erbrachten Leistungen an. Die Pipeline für Simulation von Fehlerszenarien hilft bei der Identifikation von Fehlverhalten. Eine künftige Anpassung an andere Systeme ist ein Ansatzpunkt für verbesserte Anwendbarkeit. Ein Beispiel sind die Roboter des RoboCup-Teams der TU Dortmund. Zusätzlich zu bereits durchgeführten praktischen Tests ist eine Simulation von festgelegten Fehlerszenarien voraussichtlich hilfreich. Dabei ist die Motivation analog zu der Motivation hinter dieser Projektgruppe. Simulierte Tests sind resourcensparender als physische Tests. Fehlende Resilienzmaßnahmen führen auch in diesem Kontext zu Schäden verschiedener Arten. Diese Situation ähnelt den Umständen der Projektgruppe. Die entwickelte Pipeline ist durch Anpassung an ähnliche Systeme eine Möglichkeit, die Arbeit an diesen Maßnahmen durch Fehlerinjektionen und Testen zu vereinfachen. So werden fehlende

Behandlungsformen einfacher und schneller entdeckt. Häufigeres Testen ist somit einfacher möglich.

11.3 Fazit

Das ursprüngliche Ziel der vollständig abgehandelten Fehlerliste war nicht realistisch. Insbesondere traten viele Schwierigkeiten mit der Fahrsoftware zu Beginn auf. Die Software musste erst in Zusammenarbeit mit GET Racing auf einen aktuell vollständig funktionsfähigen Stand gebracht werden. Die Arbeit der Projektgruppe hat viele Aspekte der Resilienz von JARVIC verbessert und diese Schwierigkeiten behoben. Sowohl die Software als auch das Framework wurden angepasst, um ein widerstandsfähigeres System zu schaffen. Die entwickelten Werkzeuge bilden eine solide Grundlage für weiterführende Arbeiten an der Resilienz von JARVIC. Obwohl nicht sämtliche Fehlerarten in einem Jahr abgedeckt werden konnten, war diese Unvollständigkeit erwartbar. Die erarbeiteten Methoden und Werkzeuge stellen eine wertvolle Ressource für zukünftige Entwicklungen dar.

Das Hauptergebnis der Projektgruppe ist eine stabilere JARVIC-Software mit geringeren internen Fehlern, eine Sammlung ausgewählter Beispielbehandlungen und ein Set an Werkzeugen zur Fehleranalyse und -behandlung für GET Racing. Viele entwickelte Komponenten wurden so gestaltet, dass sie unabhängig von JARVIC nutzbar sind, wodurch sie auch für andere Systeme Anwendung finden können. Zusätzlich ist eine Erweiterung der entwickelten Pipeline für andere Systeme theoretisch möglich, was eine Anwendung in anderen ROS-basierten Systemen erlaubt. Auch eine Erweiterung der geleisteten Resilienzmaßnahmen um weitere Implementationen anhand der bereits existierenden Proof-Of-Concept-Maßnahmen ist eine realistische Anwendung.

Obwohl keine direkte Implementierung von Resilienzmaßnahmen aller festgelegten Fehlerarten stattfand, wurde das übergreifende Ziel der Resilienzentwicklung erreicht. Mithilfe der entwickelten Werkzeuge und Pipeline existiert eine Grundlage für Weiterarbeit, welche zuvor nicht existierte. Weitere Entwicklung an Resilienzmaßnahmen ist anhand der Proof-Of-Concept-Implementierungen analog durchführbar. Es existieren zudem grundlegende Implementationen von verschiedenen Konzepten. Durch Unit Tests und dynamischere Behandlungen von Resilienz ist erste Arbeit erledigt. Zudem existiert JARVIC auf einem stabileren Stand mit weiter unterstütztem Framework. Die geleistete Arbeit schuf somit einen Ansatzpunkt für künftige Projektgruppen und weitere Generationen von Mitgliedern von GET Racing. Eine vollständige Form von Resilienz ist unerreichbar. Dennoch ist für weitere Arbeit eine stabile, sichere und grundlegende Basis gegeben. Diese Art der Resilienz ist eine Grundform für weitere Entwicklungen. Somit wurde durch die Projektgruppe eASy die Resilienz des betrachteten Systems verbessert und in Zukunft für weitere Gruppen vereinfacht.

Literatur

[Boz+10] Marco Bozzano u. a. „Safety, Dependability and Performance Analysis of Extended AADL Models“. In: *The Computer Journal* 54.5 (März 2010), S. 754–775. ISSN: 0010-4620. DOI: 10.1093/comjnl/bxq024 (siehe S. 41, 43).

[Dom21] Francesc Domene. *Rendering options*. Juli 2021. URL: https://carla.readthedocs.io/en/0.9.8/adv_rendering_options/ (besucht am 01. 10. 2024) (siehe S. 30).

[Dos+17] Alexey Dosovitskiy u. a. „CARLA: An Open Urban Driving Simulator“. In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, S. 1–16 (siehe S. 29).

[Fer+20] Angelo Ferrando u. a. „ROSMonitoring: A Runtime Verification Framework for ROS“. In: *Towards Autonomous Robotic Systems*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020. ISBN: 978-3-030-63486-5. DOI: 10.1007/978-3-030-63486-5_40 (siehe S. 60).

[For24] Formula Student Germany GmbH. *FSG Competition Handbook 2024*. 2024. URL: https://www.formulastudent.de/fsg/rules/ (besucht am 24. 03. 2024) (siehe S. 6–8).

[For25] Formula Student Germany GmbH. *Formula Student Rules 2025*. 2025. URL: https://www.formulastudent.de/fsg/rules/ (besucht am 25. 02. 2025) (siehe S. 5, 6, 8, 9).

[Fow24] Martin Fowler. *Continous Integration - Less Bugs*. 18. Jan. 2024. URL: https://martinfowler.com/articles/continuousIntegration.html#PracticesOfContinuousIntegration (besucht am 03. 04. 2024) (siehe S. 10).

[FS17] Brian Fitzgerald und Klaas-Jan Stol. „Continuous software engineering: A roadmap and agenda“. In: *Journal of Systems and Software* 123 (Jan. 2017), S. 176–189. ISSN: 0164-1212. DOI: 10.1016/j.jss.2015.06.063 (siehe S. 9).

[Git24a] GitHub, Inc. *About continuous deployment*. 2024. URL: https://docs.github.com/en/actions/deployment/about-deployments/about-continuous-deployment (besucht am 08. 03. 2024) (siehe S. 9).

[Git24b] GitLab B.V. *Get started with GitLab CI/CD*. 2024. URL: https://docs.gitlab.com/ee/ci/ (besucht am 08. 03. 2024) (siehe S. 10).

[Git24c] GitLab B.V. *GitLab CI/CD artifacts reports types (Requirements).* 2024. URL: https://docs.gitlab.com/ee/ci/yaml/artifacts_reports.html/#artifactsreportsrequirements (besucht am 08. 03. 2024) (siehe S. 12).

[Git24d] GitLab B.V. *Issue boards.* 2024. URL: https://docs.gitlab.com/ee/user/project/issue_board.html (besucht am 08. 03. 2024) (siehe S. 12).

[Git24e] GitLab B.V. *Requirements management.* 2024. URL: https://docs.gitlab.com/ee/user/project/requirements/ (besucht am 08. 03. 2024) (siehe S. 12).

[Glä18] Thomas Gläßle. *Numbers in scientific notation without dot are parsed as string · Issue #173 · yaml/pyyaml.* en. Juni 2018. URL: https://github.com/yaml/pyyaml/issues/173 (besucht am 30. 09. 2024) (siehe S. 90).

[Goo24] Google LLC. *GoogleTest - Testing Reference.* Juli 2024. URL: http://google.github.io/googletest/reference/testing.html#FRIEND_TEST (besucht am 29. 09. 2024) (siehe S. 87).

[GS17] Shekhar Gulati und Rahul Sharma. *Java Unit Testing with JUnit 5.* Berkeley, California, USA: Apress, 2017. ISBN: 978-1-4842-3014-5. DOI: 10.1007/978-1-4842-3015-2. (Besucht am 02. 10. 2024) (siehe S. 87).

[HTI97] Mei-Chen Hsueh, Timothy K. Tsai und Ravishankar K. Iyer. „Fault injection techniques and tools“. In: *Computer* 30.4 (Apr. 1997), S. 75–82. DOI: 10.1109/2.585157 (siehe S. 47, 48).

[Kle18] Trevor A. Kletz. *Hazop & Hazan: Identifying and Assessing Process Industry Hazards, Fouth Edition.* 4. Aufl. Boca Raton: CRC Press, Mai 2018. ISBN: 978-0-203-75222-7. DOI: 10.1201/9780203752227 (siehe S. 41).

[Lou06] Panos Louridas. „Static code analysis“. In: *IEEE Software* 23.4 (Juli 2006), S. 58–61. ISSN: 0740-7459. DOI: 10.1109/ms.2006.114 (siehe S. 90).

[Mon+02] Michael Montemerlo u. a. „FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem“. In: *Eighteenth National Conference on Artificial Intelligence.* Edmonton, Alberta, Canada: American Association for Artificial Intelligence, 2002, S. 593–598. ISBN: 0262511290. DOI: 10.5555/777092.777184 (siehe S. 14, 83, 95).

[NCM16] Roberto Natella, Domenico Cotroneo und Henrique S. Madeira. „Assessing Dependability with Software Fault Injection: A Survey“. In: *ACM Comput. Surv.* 48.3 (Feb. 2016). ISSN: 0360-0300. DOI: 10.1145/2841425 (siehe S. 46–48).

[Qui+09] Morgan Quigley u. a. „ROS: an open-source Robot Operating System“. In: *ICRA workshop on open source software.* Bd. 3. 3.2. Kobe, Japan. 2009, S. 5. DOI: 10.1109/IESTEC62784.2024.10820217 (siehe S. 12).

[RAB19] Anees U. Rehman, Rui. L. Aguiar und João Paulo Barraca. „Fault-Tolerance in the Scope of Software-Defined Networking (SDN)“. In: *IEEE Access* 7 (2019), S. 124474–124490. DOI: 10.1109/ACCESS.2019.2939115 (siehe S. 41).

[RLL15] Guido van Rossum, Jukka Lehtosalo und Łukasz Langa. *PEP 484 – Type Hints*. Mai 2015. URL: https://peps.python.org/pep-0484/ (besucht am 26. 08. 2024) (siehe S. 91).

[RS15] Enno Ruijters und Mariëlle Stoelinga. „Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools“. In: *Computer science review* 15 (2015), S. 29–62. DOI: 10.1016/j.cosrev.2015.03.001 (siehe S. 42).

[Rub+12] Alex Rubinsteyn u. a. „Parakeet: A Just-In-Time parallel accelerator for python“. In: *4th USENIX Workshop on Hot Topics in Parallelism (HotPar 12)*. 2012. DOI: 10.5555/2342788.2342802 (siehe S. 88).

[Sad+18] Caitlin Sadowski u. a. „Modern code review: a case study at google“. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE ’18. ACM, Mai 2018. DOI: 10.1145/3183519.3183525 (siehe S. 12).

[Sal+22] Nadir K. Salih u. a. „A Survey on Software/Hardware Fault Injection Tools and Techniques“. In: *2022 IEEE Symposium on Industrial Electronics and Applications (ISIEA)*. 2022, S. 1–7. DOI: 10.1109/ISIEA54517.2022.9873679 (siehe S. 46–48).

[SAZ17] Mojtaba Shahin, Muhammad Ali Babar und Liming Zhu. „Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices“. In: *IEEE Access* 5 (2017), S. 3909–3943. ISSN: 2169-3536. DOI: 10.1109/access.2017.2685629 (siehe S. 9).

[Smi01] Larry Smith. „Shift-Left Testing“. In: *Dr. Dobb’s Journal* 26.9 (Sep. 2001), S. 56–62 (siehe S. 3).

[SS25] Ken Schwaber und Jeff Sutherland. *Scrum Guide*. 2025. URL: https://scrumguides.org/scrum-guide.html (besucht am 24. 03. 2025) (siehe S. 22).

[Tad+20] Monika Taddicken u. a. „Wirtschaftlicher Nutzen statt gesellschaftlicher Debatte? Eine quantitative Framing- Analyse der Medienberichterstattung zum autonomen Fahren.“ In: *Medien & Kommunikationswissenschaft* 68 (Nov. 2020), S. 406–427. DOI: 10.5771/1615-634X-2020-4-406 (siehe S. 1).

[Tiw+14] Ashish Tiwari u. a. „Safety envelope for security“. In: *Proceedings of the 3rd International Conference on High Confidence Networked Systems*. HiCoNS ’14. Berlin, Germany: Association for Computing Machinery, 2014, S. 85–94. ISBN: 9781450326520. DOI: 10.1145/2566468.2566483 (siehe S. 98).

[TV13] Andrew Tanenbaum und Maarten Van Steen. *Distributed Systems Principles and Paradigms*. Pearson Deutschland, 2013, S. 640. ISBN: 9781292025520. URL: https://elibrary.pearson.de/book/99.150005/9781292038001 (siehe S. 13).

[Woo14] William Woodall. *ROS on DDS*. 2014. URL: https://design.ros2.org/articles/ros_on_dds.html (besucht am 04. 03. 2025) (siehe S. 13).

Tabellenverzeichnis

Tabelle 2.1 Punktetabelle der dynamischen Events des FSG (nach [For25, Kap. A.1]). . 9

Tabelle 6.1 Die wichtigsten Fehler gruppiert nach Schwierigkeitsgrad der Injektion. Injizierbar beschreibt dabei, ob die Fehler mit dem aktuellen Technikstand injiziert werden können. 44

Anhang A

Lesernotizen

Anhang B

FLEX

```
1  grammar Flex
2
3  entry FlexModel:
4    useStatement=UseStatement
5    faultStatements+= (PerceptionFaultStatement | ControlFaultStatement);
6
7  UseStatement:
8    'Use' 'simulator' simulator=SIMULATOR 'with' 'map' map=ID'.';
9
10 // Specification of PerceptionFaultStatement
11 PerceptionFaultStatement:
12   'Fault' 'perception/' perceptionSpec=PerceptionSpecification '.';
13
14 PerceptionSpecification:
15   ('cones/' coneSpec=ConeSpecification | 'fusion_lidar_camera');
16
17 ConeSpecification:
18   ('wrong_color' numberOfWrongColorCones=INT 'cones' | 'phantom_cones/' phantomSpec=PhantomSpecification | 'cones_on_track'
        numberOfConesOnTrack=INT 'cones') {infer ConeSpecification};
19
20 PhantomSpecification:
21   ('correct_side' | 'wrong_side') numberOfCones=INT 'cones' {infer PhantomSpecification};
22
23 // Specification of ControlFaultStatement
24 ControlFaultStatement:
25   'Fault' 'control/' controlSpec=ControlSpecification 'in' timeRange=TimeRange '.';
26
27 ControlSpecification:
28   ('engine_damage/' engineDamageSpec=EngineDamageSpecification | 'steering_actor/' steeringActorSpec=
        SteeringActorSpecification);
29
30 EngineDamageSpecification:
31   'torque_limit_factor' severity=DECIMAL {infer EngineDamageSpecification};
32
33 SteeringActorSpecification:
34   ('angle_deviation' degreeNumber=INT 'degrees' ('right' | 'left') | 'steering_failure') {infer SteeringActorSpecification};
35
36 TimeRange:
37     '[' startTime=INT ',' endTime=INT ']' timeUnit=TIMEUNIT {infer TimeRange};
38
39 // Terminals
40 hidden terminal WS: /\s+/;
41 terminal SIMULATOR: /fssim|fsds/;
42 terminal TIMEUNIT: /seconds|milliseconds/;
43 terminal ID: /[_a-zA-Z][\w_]*/;
44 terminal DECIMAL: /[0-9]+\.[0-9]+/;
45 terminal INT returns number: /[0-9]+/;
46 terminal STRING: \/"(\\.|[^"\\])*"|'(\\.|[^'\\])*'/;
47
48 hidden terminal ML_COMMENT: /\/\*[\s\S]*?\*\//;
49 hidden terminal SL_COMMENT: /\/\/[^\n\r]*/;
```

Listing B.1: Langium Grammatik für FLEX.

```
1  import { FlexModel, ControlFaultStatement, PerceptionFaultStatement } from '../language/generated/ast.js';
2  import * as fs from 'node:fs';
3  import * as path from 'node:path';
4  import { extractDestinationAndName } from './cli-util.js';
5
6  export function generateJson(model: FlexModel, filePath: string, destination: string | undefined): string {
7      const data = extractDestinationAndName(filePath, destination);
8      const generatedFilePath = `${path.join(data.destination, data.name)}.json`;
9      let jsonData = {};
10         jsonData = {
11             simulation_tool: model.useStatement.simulator,  // Nimmt den Simulator aus der UseStatement
12             map_name: model.useStatement.map, // Nimmt die Map aus der UseStatement
13             faults: generateFaultCode(model.faultStatements)// Wandelt FaultStatements in JSON um
14         };
15
16
17     if (!fs.existsSync(data.destination)) {
18         fs.mkdirSync(data.destination, { recursive: true });
19     }
20     fs.writeFileSync(generatedFilePath, JSON.stringify(jsonData, null, 2));
21     return generatedFilePath;
22 }
23
24 function generateFaultCode(faultStatements: (ControlFaultStatement | PerceptionFaultStatement)[] | undefined) {
25     if (faultStatements === undefined) {
26         return null; // or handle the case when faultStatements is undefined
27     }
28
29     const faults = faultStatements.map((faultStatement) => {
30         if (isControlFaultStatement(faultStatement)) {
31             return generateControlFaultCode(faultStatement);
32         } else if (isPerceptionFaultStatement(faultStatement)) {
33             return generatePerceptionFaultCode(faultStatement);
34         }
35
36         return null; // Add a return statement to handle the case when none of the conditions are met
37     });
38
39     return faults;
40 }
```

Listing B.2: Ausschnitt des Code-Generators für FLEX in einer Datei. Dabei handelt es sich um einen speziellen Interpreter.