1 Thema

eASy: Resilient **A**utonomous **Sy**stems

2 Zeitraum

Sommersemester 2023 und Wintersemster 2023/2024

3 Veranstalter

Daniel Busch <daniel.busch@cs.tu-dortmund.de>, LS5, Tel. 5776, OH14 / 111
Simon Dierl <simon.dierl@cs.tu-dortmund.de>, LS14, Tel. 7543, OH12 / 2.008
Prof. Dr. Falk Howar <falk.howar@tu-dortmund.de>, LS14, Tel. 7945, OH12 / 2.009

4 Aufgabe

In den letzten Jahren ist die Vision vom autonomen Fahren durch Fortschritte in Forschung und Technik zunehmend Realität geworden. Gleichzeitig ist allerdings klar geworden, dass autonome Fahrsysteme noch nicht in der Lage sind, jede unbekannte und unerwartete Situation zu meistern. Beispiele sind die in Abb. 1 gezeigten Unfälle, bei denen das überraschende Versagen einer Komponente zu schwerwiegenderem Fehlverhalten führte: Der Tesla in Abb. 1a kollidierte im regulären Straßenverkehr spontan mit einem grünen Poller, den die Objekterkennung vermutlich nicht korrekt klassifizieren konnte; während das autonome Rennauo in Abb. 1b nach einem Fehler in der Kartierung in die Streckenbegrenzung abbog und Nase und Frontspoiler zerstörte.

Das Fazit muss sein, dass Softwarearchitekturen für autonome Systeme stärker als andere Programme *resilient* [1] sein müssen. Ausfälle und Fehlfunktionen von Hardware- und Softwarekomponenten müssen vorhergesehen, erkannt und sinnvoll behandelt werden. Während der Ausfall eines zentralen Sensors (z. B. eines Lidars) durch das Fehlen von Daten erkannt und durch eine Gefahrenbremsung behandelt wird, kann ein neuronales Netz schwer zu erkennende Fehler zeigen, die jedoch durch Umschalten auf ein alternatives System behoben werden können.

Der Test derartiger Mechanismen gestaltet sich jedoch als Herausforderung, da bei Tests auf realer Hardware ein Fehler (oder je nach Umständen sogar eine erfolgreiche Behandlung) zu Schäden am Fahrzeug führen kann. Eine Lösung können Simulationen mit integrierten Messungen sein, die es ermöglichen, große Testkampagnen mit hoher Frequenz und ohne Risiko durchzuführen. Idealerweise werden solche Simulationen als Quality Gates in Entwicklungsprozesse integriert.

Vision

Um resiliente autonome Fahrfunktionen zu entwickeln, ist es also zunächst nötig, mögliche Fehler zu ermitteln. Eine etablierte Technik ist hier z. B. die *Fault Tree Analysis* [7]. Im nächsten Schritt muss untersucht werden, welche Umstände zu diesen Fehlern führen. Dies wird aktuell auch unter dem Begriff der *Triggering Condition* [8] erforscht.

Im Anschluss können für jeden Fehlerfall geeignete Behandlungen entworfen und umgesetzt werden. Dies kann durch Anpassung der Komponenten des autonomen Systems (z. B. durch Interpolieren fehlender Bilddaten), durch Implementierung weiterer Komponenten (z. B. eine Notbremse) oder auf einer architekturellen Ebene (z. B. durch *Safety Envelopes* [2]) geschehen.

Die Validierung dieser Resilienz erfordert jedoch eine Möglichkeit, Fehlerfälle reproduzierbar herbeizuführen und das Verhalten des Systems im Fehlerfall auf Korrektheit zu prüfen. Dazu muss

(a) Ein Tesla kollidiert frontal mit einem Poller, der vermutlich von der Objekterkennung nicht klassifiziert wurde.

(b) Das Formula Student-Rennauto des KIT rammt nach einem Fehler in der Kartierung einen Reifenstapel.

Abbildung 1: Beispiele für Unglücke durch unzureichend abgesicherte Fahrsoftware.

das System so instrumentiert werden, dass Fehler injiziert werden können. Eine Validierung kann dann simulationsbasiert stattfinden, indem das Verhalten in der Aufzeichnung der Simulation geprüft wird. Auf realer Hardware, z. B. Modellfahrzeugen, können zwar Hardware- und Vehicle-in-the-Loop-Tests weitere Resultate liefern, sind jedoch schwerer zu evaluieren.

Ziele und Vorgehen

Die Aufgabe der Projektgruppe ist es, ein autonomes System resilient zu gestalten und diese Resilienz strukturiert zu validieren. Dazu sollen Fehlerfälle identifiziert und Maßnahmen gegen sie ergriffen werden und parallel eine simulationsbasierte Validierungsarchitektur aufgestellt werden, die den Erfolg der Maßnahmen überprüft und quantifiziert.

Als konkrete Fallstudio soll die autonome Fahrsoftware Jarvic dienen. Diese wird vom GET racing e. V. für den Formula Student-Rennwagen FS223 (s. Abb. 2c) auf Basis des ROS-Frameworks[1] [4, 5] entwickelt. Jarvic verfügt bereits über eine Infrastruktur, welche die dauerhafte Validierung der Software in Simulationen (z. B. FSSIM [3], s. Abb. 2a) ermöglicht und kann ebenfalls auf zwei Modellautos im Maßstab 1:8 (s. Abb. 2b) ausgeführt werden.

Analyse von Fehlerfällen

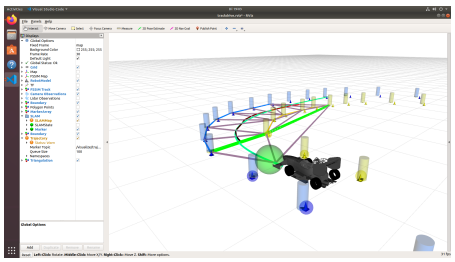
Der erste Schritt ist die Analyse von möglichen *Fehlerfällen*. Dazu sollte die Projektgruppe vergangene Unfälle insbesondere in der Formula Student untersuchen, Jarvic analysieren und auch Personen im GET racing Dortmund e. V. befragen. Hierbei sollen sowohl „klassische“ Bugs wie Speicherschutzverletzungen, logische Fehler in Algorithmen, Versagen von Black-Box-Komponenten wie neuronalen Netzen als auch Hardwarefehler betrachtet werden.

Ziel Eine umfassende Dokumentation möglicher Fehler, die beim Betrieb des Systems auftreten können und geeignet systematisiert wurden.

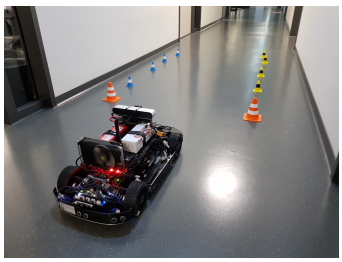
Unterstützung von Fehlerinjektion

Der nächste Schritt ist die Entwicklung einer Methode, um gezielte *Fehlerinjektionen* auszulösen, also die identifizierten Fehler entweder herbeizuführen oder „glaubwürdig“ zu simulieren. Ansätze können hier eine direkte Modifikation der von Jarvic definierten ROS-Komponenten, die Nutzung von bereits vorhandenen ROS-Features, aber auch die Implementierung eigener Komponenten, die – ähnlich einem AOP-Aspekt – den Datenstrom modifizieren, sein. Z. B. kann das vollständige Versagen einer Komponente durch `kill`-en des Prozesses, ROS-seitiges Deaktivieren

[1] https://www.ros.org/



(a) Ein autonomes Trackdrive-Rennen wird in FSSIM simuliert; sichtbar ist die Trajektorienplanung.



(b) Ein AADC-Modellauto fährt ein autonomes Beschleunigungsrennen im Maßstab 1:8.

(c) Der FS223-Rennwagen fährt bemannt ein Ausdauerrennen bei der Formula Student Germany.

Abbildung 2: Von JARVIC unterstützte Test- und Produktivplattformen.

oder durch Einfügen eines Knotens, der alle Daten verwirft, simuliert werden. Eine verschmutzte Kameralinse ist hingegen leichter durch Veränderung des Bilddatenstroms simulieren. Es sollen also jeweils geeignete Möglichkeiten genutzt werden. Ebenso wichtig ist es, dass Fehlermodi zur Laufzeit konfiguriert und ausgelöst werden können.

Ziel JARVIC verfügt über parametrisierte Schnittstellen, die es ermöglichen, zur Laufzeit automatisiert gezielt relevante Fehler zu simulieren.

Validierung durch Fehlerszenarien

Um die Schnittstellen für Fehlerinjektion für Continuous Simulation zu erschließen, ist es notwendig, *Fehlerszenarien* zu unterstützen, die ein Fahrszenario (z.Ḃ. ein Trackdrive-Rennen auf der FSG 2023-Strecke) mit einem oder mehreren auftretenden Fehlern (z. B. Lidar-Ausfall nach Abschluss der ersten Runde) kombinieren. Der naive Ansatz, Testfälle in Quellcode umzusetzen und parallel zur Fahrsoftware auszuführen, führt zu einer wachsenden Menge an schwer wart- und lesbarem Testcode. Daher soll untersucht werden, wie Fehlerszenarien deklarativ notiert werden können (z. B. als YAML) und eine entsprechende Sprache entworfen werden. Ergänzend soll untersucht werden, welche Metriken geeignet sind, um zu bewerten, ob sich das System erfolgreich resilient verhalten hat. So könnte nicht die Rundenzeit aussagekräftig sein, sondern das Verbleiben im Kurs. Hierfür könnte unser STARS-Framework [6] evaluiert werden.

Ziel Die Fehlerinjektion kann automatisch in der Simulation und optional auf dem Modellauto ausgelöst werden. Resilienz ist damit Teil der Continuous Validation.

Implementierung von Resilienz

Die aufgebaute Validierungsarchitektur kann nun die Entwicklung von Resilienz begleiten und absichern. Es gibt diverse Ansätze, die sich für unterschiedliche Aspekte eignen: möglich sind direkte Veränderungen an der JARVIC-Codebasis (z. B. Anpassen von Algorithmen), das Implementieren von Monitoren und Safety Envelopes, die sich inkorrekt verhaltende Komponenten überstimmen, aber auch Hilfskomponenten (z. B. zum Interpolieren fehlender Eingaben).

Ziel Eine Sicherheitsarchitektur für JARVIC, welche die Fehlerfälle zuverlässig erkannt und so gut wie möglich behandelt, die durch das CI/CD-System dauerhaft validiert wird.

5 Teilnahmevoraussetzungen

Für eine Teilnahme sind folgende Kenntnisse *notwendig*:

- Sichere Beherrschung einer Programmiersprache (z. B. C++, JAVA, oder PYTHON)
- Grundkenntnisse in der Arbeit mit Linux

Folgende Kenntnisse sind *hilfreich*, werden jedoch nicht vorausgesetzt:

- Grundkenntnisse zu cyber-physischen Systemen (Regelungstechnik etc.)
- Erfahrungen mit ROS
- Vorwissen im Bereich des autonomen Fahrens

6 Minimalziel

Die Minimalziele der PG sind:

(Z1) Durchführung einer Fehleranalyse für JARVIC, in der möglichst vollständig Fehlermodi identifiziert und geeignet klassifiziert werden

(Z2) Umsetzung von Injektion für die identifizierten Fehlerklassen, d. h. der Möglichkeit die Fehler zur Laufzeit programmatisch zu emulierten, soweit technisch möglich.

(Z3) Entwicklung von Fehlerszenarien, die in der Simulation ausgeführt werden können und ermöglichen, zu testen, inwiefern JARVIC die Fehlerszenarien korrekt behandelt

(Z4) Umsetzung von *Resilienz*: JARVIC kann in der Simulation in den Szenarien die Fehlermodi erkennen und im Rahmen der technischen Möglichkeiten so behandeln, dass zumindest Schäden am Fahrzeug vermieden würden und idealerweise eine erfolgreiche Fortsetzung des Rennens möglich wird.

7 Literatur

[1] Saurabh Bagchi u. a. „Vision paper: grand challenges in resilience: autonomous system resilience through design and runtime measures“. In: *IEEE Open Journal of the Computer Society* 1 (2020), S. 155–172. DOI: 10.1109/OJCS.2020.3006807.

[2] Matthew Brown u. a. „Safe driving envelopes for path tracking in autonomous vehicles“. In: *Control Engineering Practice* 61 (Apr. 2017), S. 307–316. DOI: 10.1016/j.conengprac.2016.04.013.

[3] Juraj Kabzan u. a. „AMZ driverless: the full autonomous racing system“. In: *Journal of Field Robotics* 37.7 (2020), S. 1267–1294. DOI: 10.1002/rob.21977.

[4] Steven Macenski u. a. „Robot Operating System 2: design, architecture, and uses in the wild“. In: *Science Robotics* 7.66 (2022), S. 1–12. DOI: 10.1126/scirobotics.abm6074.

[5] Morgan Quigley u. a. „ROS: an open-source robot operating system“. In: *ICRA 2009 Workshop on Open Source Software in Robotics*. ICRAOSS 2009. 2009.

[6] Till Schallau u. a. *Tree-based scenario classification: a formal framework for coverage analysis on test drives of autonomous vehicles*. Techn. Ber. arXiv:2307.05106. arXiv, Juli 2023. DOI: 10.48550/arXiv.2307.05106.

[7] Liudong Xing und Suprasad V. Amari. „Fault tree analysis“. In: *Handbook of Performability Engineering*. Hrsg. von Krishna B. Misra. London: Springer, 2008, S. 595–620. DOI: 10.1007/978-1-84800-131-2_38.

[8] Zhijing Zhu u. a. „Systematization and identification of triggering conditions: a preliminary step for efficient testing of autonomous vehicles“. In: *2022 IEEE Intelligent Vehicles Symposium (IV)*. IV 2022. Juni 2022, S. 798–805. DOI: 10.1109/IV51971.2022.9827238.

8 Rechtlicher Hinweis

Die Ergebnisse der Projektarbeit und die dabei erstellte Software sollen der Fakultät für Informatik uneingeschränkt für Lehr- und Forschungszwecke sowie dem GET racing Dortmund e. V. für den Betrieb autonomer Fahrsysteme zur freien Verfügung stehen. Darüber hinaus sind keine Einschränkungen der Verwertungsrechte an den Ergebnissen der Projektgruppe und keine Vertraulichkeitsvereinbarungen vorgesehen.