Evaluation der Programmiersprache Rust für den Entwurf und die Implementierung einer hochperformanten, serverbasierten Kommunikationsplattform für Sensordaten im Umfeld des automatisierten Fahrens

Michael Watzko[*], **Manfred Dausmann**, **Kevin Erath**

Fakultät Informationstechnik der Hochschule Esslingen – University of Applied Sciences

Sommersemester 2018

Motivation

Der Begriff „autonomes Fahren“ hat spätestens seit den Autos von Tesla einen allgemeinen Bekanntheitsgrad erreicht. Damit ein Auto selbstständig fahren kann, müssen erst viele Hürden gemeistert werden. Dazu gehört zum Beispiel das Spur halten, das richtige Interpretieren von Verkehrsschildern und das Navigieren durch komplexe Kreuzungen.

Externe Sensorik könnte hierbei Informationen liefern, die das Auto selbst nicht erfassen kann. Aber was ist, wenn diese unterstützenden Systeme falsche Informationen liefern? Eine Parklücke, wo keine ist; eine freie Fahrbahn, wo ein Radfahrer fährt; ein angeblich entgegenkommendes Auto, eine unnötig Vollbremsung, ein Auffahrunfall. Ein solches System muss sicher sein – nicht nur vor Hackern. Es muss funktional sicher sein, Redundanzen und Notfallsysteme müssen jederzeit greifen.

Komplexe Prüfalgorithmen und Notfallstrategien lösen diese Anforderung, schaffen aber auch einen Spielraum für Fehler. Schnell kompiliert etwas, dass in einem Randfall einen Speicherbereich doppelt freigibt, in eine Spezifikationslücke der Programmiersprache fällt oder eine Wettlaufsituation ermöglicht. Im Resultat handelt das System total unvorhersehbar und gefährdet damit die Insassen und andere Verkehrsteilnehmer.

Was wäre, wenn es eine Programmiersprache geben würde, die so etwas nicht zulässt: die fehlerhaften Strategien im Speichermanagement zur Compilezeit findet und die Compilation stoppt; die trotz erzwungener Sicherheitsmaßnahmen, schnell und echtzeitnah reagieren kann und sich nicht vor Geschwindigkeitsvergleichen mit etablierten, aber unsicheren Programmiersprachen, scheuen muss?

Diese Arbeit soll zeigen, dass Rust genau so eine Programmiersprache ist und sich für sicherheitsrelevante, hoch parallelisierte und echtzeitnahe Anwendungsfälle bestens eignet.

Die Programmiersprache Rust

Rust hat als Ziel, eine sichere und performante Systemprogrammiersprache zu sein. Abstraktionen sollen die Sicherheit, Lesbarkeit und Nutzbarkeit verbessern aber keine unnötigen Performance-Einbußen verursachen.

Abbildung 1: Offizielles Logo der Programmiersprache Rust

Aus anderen Programmiersprachen bekannte Fehlerquellen – wie vergessene NULL-Pointer Prüfung, vergessene Fehlerprüfung, „dangling pointers“ oder „memory leaks“ – werden durch strikte Regeln und mit Hilfe des Compilers verhindert. Im Gegensatz zu Programmiersprachen, die dies mit Hilfe ihrer Laufzeitumgebung[1] sicherstellen, werden diese Regeln in Rust durch eine statische Lebenszeitanalyse und mit dem Eigentümerprinzip bei der Kompilation überprüft und erzwungen. Dadurch erreicht Rust eine zur Laufzeit hohe Ausführgeschwindigkeit.

Das Eigentümerprinzip und die Markierung von Datentypen durch Merkmale vereinfacht es zudem, nebenläufige und sichere Programme zu schreiben.

Speicherverwaltung

Rust benutzt ein „statisches, automatisches Speichermanagement – keinen Garbage

[*] Diese Arbeit wurde durchgeführt bei der Firma IT-Designers GmbH, Esslingen

[1] u.a. Java Virtual Maschine (JVM), Common Language Runtime (CLR)

Collector“ [2]. Das bedeutet, die Lebenszeit einer Variable wird statisch während der Compilezeit anhand des Geltungsbereichs ermittelt. Durch diese statische Analyse findet der Compiler heraus, wann der Speicher einer Variable wieder freigegeben werden muss. Dies ist genau dann, wenn der Geltungsbereich des Eigentümers zu Ende ist. Weder ein Garbage-Collector, der dies zur Laufzeit nachverfolgt, noch ein manuelles Eingreifen durch den Entwickler (zum Beispiel durch einen Aufruf von *free()*, wie in C/C++ üblich) ist nötig.

Falls der Compiler keine ordnungsgemäße Nutzung feststellen kann, wie zum Beispiel eine Referenz, die ihren referenzierten Wert überleben möchte, wird die Kompilation verweigert. Dadurch wird das Problem des „dangling pointers“ verhindert, ohne Laufzeitkosten zu erzeugen.

Eigentümer- und Verleihprinzip

Bereits 2003 beschreibt Bruce Powel Douglass im Buch „Real-Time Design Patterns“, dass „passive“ Objekte ihre Arbeit nur in dem Thread-Kontext ihres „aktiven“ Eigentümers tätigen sollen (Seite 204, [3]). In dem beschriebenen „Concurrency Pattern“ werden Objekte eindeutig Eigentümern zugeordnet, um so eine sicherere Nebenläufigkeit zu erlauben.

Diese Philosophie setzt Rust direkt in der Sprache um, denn in Rust darf ein Wert immer nur einen Eigentümer haben. Zusätzlich zu einem immer eindeutig identifizierbaren Eigentümer, kann der Wert auch ausgeliehen werden, um einen kurzzeitigen Zugriff zu erlauben. Eine Leihgabe ist entweder exklusiv und ermöglicht sowohl Lese- als auch Schreibzugriffe, oder sie ist auf einen Lesezugriff beschränkt und erlaubt im Gegenzug an mehreren Stellen gemeinsam genutzt zu werden.

Eigentümerschaft kann auch übertragen werden, der vorherige Eigentümer kann danach nicht mehr auf den Wert zugreifen. Ein entsprechender Versuch wird mit einer Fehlermeldung durch den Compiler bemängelt.

Die statische Lebenszeitanalyse garantiert, dass es nur einen Eigentümer, eine exklusive Schreiberlaubnis oder mehrere Leseerlaubnisse auf eine Variable gibt.

Sichere Nebenläufigkeit

Eine sichere Nebenläufigkeit wird in Rust durch das Eigentümerprinzip in Kombination mit zusätzlichen Typmerkmalen erreicht. Dabei ist diese sichere Nebenläufigkeit meist unsichtbar (Seite 41, [1]), da der Compiler eine unsichere und damit syntaktisch falsche Verwendung nicht übersetzt. Ein Rust Programm das kompiliert, ist daher, in vielerlei Hinsicht, sicher in der Nebenläufigkeit. Einzig ein „Deadlock“ kann nicht statisch ermittelt und verhindert werden.

Eine Wettlaufsituation (englisch „race condition“) um einen Wert ist in Rust nicht möglich. Das Eigentümer- und Leihprinzip verhindert dies, denn es kann nur exklusiv schreibend auf einen Wert zugegriffen werden. Für einen Datenwettlauf muss dagegen, gleichzeitig zu einem schreibenden, ein lesender Zugriff erfolgen.

Datentypen, die einen gemeinsamen Zugriff auf veränderliche Werte ermöglichen, liefern immer ein Ergebnis, ob der Versuch, einen exklusiven Schreib- oder Lesezugriff zu erhalten, geklappt hat. Erst nach einer Fehlerauswertung kann auf den Wert zugegriffen werden.

Anwendung im Projekt MEC-View

Das MEC-View Projekt befasst sich mit der Thematik hochautomatisierter Fahrzeuge und ist vom Bundesministerium für Wirtschaft und Energie (BMWi) gefördert. Im Rahmen des MEC-View Projekts soll erforscht werden, ob und in wie weit eine durch externe Sensorik geleistete Unterstützung nötig und möglich ist, um in eine Vorfahrtstraße automatisiert einzufahren.

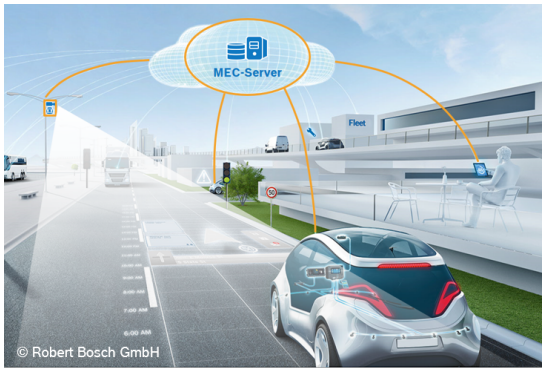

Abbildung 2: MEC-View-Schaubild der Robert Bosch GmbH

Das Forschungsprojekt ist dabei ein Zusammenschluss mehrerer Unternehmen mit unterschiedlichen Themengebieten. Die IT-Designers Gruppe beschäftigt sich mit der Implementation des Kommunikationsservers, der auf der von Nokia zur Verfügung gestellten Infrastruktur im 5G Mobilfunk als MEC-Server betrieben wird. Erkannte Fahrzeuge und andere Verkehrsteilnehmer werden von den Sensoren von Osram via Mobilfunk an den Kommunikationsserver übertragen. Der Kommunikationsserver stellt diese Informationen dem Fusionsalgorithmus der Universität Ulm zur Verfügung und leitet das daraus gewonnene Umfeldmodell an die hochautomatisierten Fahrzeuge der Robert Bosch GmbH und der Universität Ulm weiter. Durch hochgenaue, statische und dynamische Karten von TomTom und den Fahrstrategien von Daimler

und der Universität Duisburg soll das Fahrzeug daraufhin automatisiert in die Kreuzung einfahren können.

Die Kommunikationsplattform des Servers nimmt für das Forschungsprojekt eine zentrale Rolle ein. Falsche, verfälschte oder verspätete Informationen können für das Fahrzeug und andere Verkehrsteilnehmer verheerende Auswirkungen haben. Eine gewissenhafte und funktional sichere Implementation ist deswegen vonnöten und wird bereits durch eine Implementation in C++ bereitgestellt.

Systemprogrammiersprachen, wie C und C++, werden in diesen Situationen gerne genutzt, da sie keine Laufzeitumgebung benötigen, sondern zu Maschinencode kompiliert werden. Auch Just-In-Time (JIT) Compiler oder Garbage Collectoren (GC) sind bei Anwendungsfällen im echtzeitnahen Umfeld ein Ausschlusskriterium. Mit ihnen variieren die Reaktionszeiten während der Laufzeit und eine maximale Reaktionszeit kann nicht garantiert werden.

Eine funktional sichere Implementation in C++ zu schaffen ist jedoch eine erhebliche Herausforderung. Schnell können sich Fehler im Speichermanagement oder in Strategien zur Nebenläufigkeit einschleichen, die sich erst nach der Kompilation in Tests, Kontrollen durch Programme wie Valgrind oder im schlimmsten Fall während des Produktivbetriebs zeigen.

Diese Bachelorarbeit beschäftigt sich mit der Implementierung der Kommunikationsplattform in Rust. Durch die zuvor erwähnten Garantien von Rust, sollte eine höhere Stabilität und damit eine bessere Eignung für den Einsatzbereich in funktional sicheren Anwendungen resultieren, die Programmiersprachen wie C und C++ erst durch eine hohe Sorgfalt bei der Entwicklung erreichen. Zuletzt wird die Reaktionszeit der Rust-Kommunikationsplattform der C++-Referenzimplementation gegenübergestellt, um zu prüfen, ob sich die Programmiersprache auch für den Einsatz in echtzeitnahen Systemen eignet.

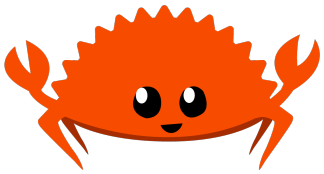

Abbildung 3: Ferris, das inoffizielles Maskottchen der Programmiersprache Rust

[1] Jim Blandy und Jason Orendorff: Programming Rust, 2017, ISBN: 1491927283

[2] Frequently Asked Questions – The Rust Programming Language `https://www.rust-lang.org/en-US/faq.html`

[3] B.P. Douglass. Real-time Design Patterns: Robust Scalable Architecture for Real-time Systems, ISBN 9780201699562, 2003

Bildquellen:

- Abbildung 1: `https://www.rust-lang.org/logos/rust-logo-256x256.png`
- Abbildung 2: `http://mec-view.de/`
- Abbildung 3: `http://www.rustacean.net/`