Hochschule Esslingen
University of Applied Sciences

Fakultät Informationstechnik
im Studiengang Softwaretechnik und Medieninformatik

Bachelorarbeit

Entwurf und Implementierung einer hochperformanten, serverbasierten Kommunikationsplattform für Sensordaten im Umfeld des automatisierten Fahrens in Rust

Michael Watzko

Sommersemester 2018
14.02.2018 - 22.06.2018

Erstprüfer: Prof. Dr. rer. nat. Dipl.-Inform. Manfred Dausmann
Zweitprüfer: ... Hannes Todenhagen

Firma: IT Designers GmbH
Betreuer: Dipl. Ing. (FH) Kevin Erath M.Sc.

[git] • master @ ea0bab9 • 22. Februar 2018

Sperrvermerk

U SHALL NOT PASS

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Esslingen, den 22. Februar 2018 _____

Michael Watzko

Danksagungen

„Alle Zitate aus dem Internet sind wahr!“

Albert Einstein

„Rust is a vampire language, it does not reflect at all!“

https://www.youtube.com/watch?v=-Tj8Q12DaEQ

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Der Begriff „autonomes Fahren“ hat spätestens seit den Tesla Autos einen allgemeinen Bekanntheitsgrad erreicht. Um ein Auto selbstständig fahren lassen zu können, müssen erst viele Hürden gemeistert werden, zum Beispiel das Spur halten, auch bei fehlenden Fahrspurmarkierungen, das Interpretieren von Stoppschildern und navigieren durch komplexen Kreuzungen. TODO: ref tesla.com?

Bevor das Auto Entscheidungen treffen kann, muss es zuallererst ein Modell seines Umfelds erstellen oder zur Verfügung gestellt bekommen. Aber vielleicht kann ein Auto nicht immer selbständig genügend Informationen zu seinem Umfeld sammeln? TODO: (huhuhu Server implied huhuhu) TODO: fix 404

1.2 Projektkontext

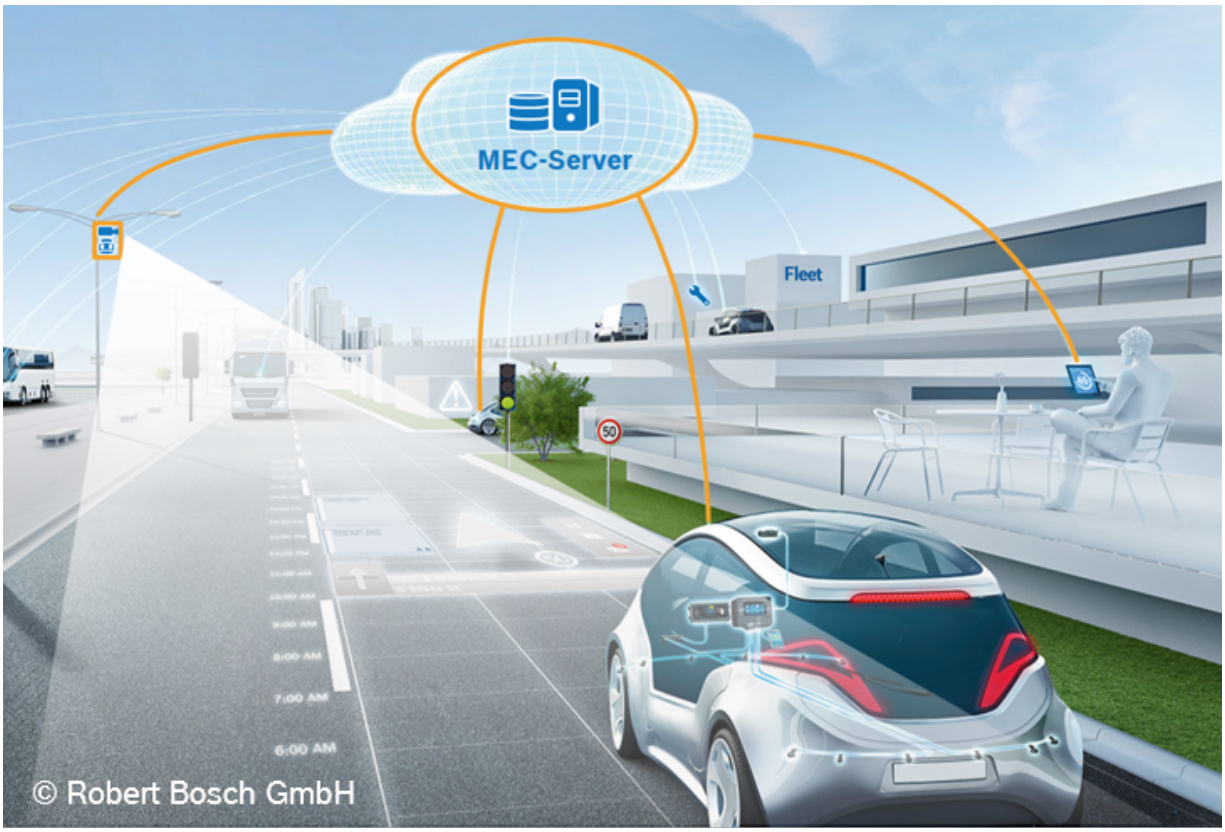

Abbildung 1.1: MEC-View Schaubild der Robert Bosch GmbH [MEC]

Quelle: https://www.uni-due.de/~hp0309/images/Schaubild_BoschStyle_V2.png

Diese Abschlussarbeit befasst sich mit dem MEC-Server, der Teil des MEC-View Forschungsprojekt ist. Das MEC-View Projekt wird durch das BMWi gefördert und befasst sich mit der Thematik autonom fahrender Fahrzeuge. Es soll erforscht werden ob und in wie weit eine durch externe Sensorik geleistete Unterstützung nötig und möglich ist um in eine Vorfahrtsstraße autonom einzufahren.

Das Forschungsprojekt ist dabei ein Zusammenschluss verschiedener Unternehmen:

Unternehmen	Aufgabenbereich
Bosch	Hochautomatisiertes Fahrzeug
Osram	„Intelligente“ Infrastruktursensoren
Nokia	5G Mobilfunk, Mobile Edge Computing (MEC)
Universität Ulm	Sensordatenfusion, Prädiktion
IT-Designers Gruppe	MEC-Server Architektur Mikroskopische Verkehrsanalyse Verhaltensanalyse
TomTom	Hochgenaue statische und dynamische Karten
Daimler	Fahrstrategien Verhaltensanalyse Streckenfreigabe
Universität Duisburg	Mikroskopisch, stochastische Simulationsmodelle

1.2.1 Ablauf

Externe Sensoren übermitteln erkannte Fahrzeuge via Mobilfunk an einen MEC-Server, der direkt am Empfängerfunkmast angeschlossen ist. TODO: platform, vm? Nachdem die erkannten Fahrzeuge der verschiedenen Sensoren zusammengeführt wurden (Fusions-Algorithmus), sollen sie an das autonom fahrende Fahrzeug über Mobilfunk übermittelt werden. Somit erhält das Fahrzeug bereits im Voraus Einsicht über eventuelle Möglichkeiten in die Vorfahrtsstraße einzufahren und könnte deshalb beispielsweise die Geschwindigkeit anpassen. Zudem sollen bei unübersichtlichen Kreuzungen somit zuverlässiger andere Verkehrsteilnehmer erkannt werden.

1.2.2 Sicherheit

TODO: überhaupt relevant? Da bei Fehlern möglicherweise andere Verkehrsteilnehmer zu Schaden kommen können, müssen diverse Sicherheitsrichtlinien beachtet werden. Die Industrienorm ISO 26262 beschreibt dabei verschiedene Vorgehensweisen, unter anderem eine FBA, Risikoabschätzung durch Einstufung nach SILs und beschreibt Gegenmaßnahmen.

1.3 Zielsetzung

Das Ziel ist es, eine alternative Implementierung des MEC-View Servers in Rust zu schaffen. Durch die Garantien (Abschnitt 2.12) von Rust wird erhofft, dass der menschliche Faktor als Fehlerquelle gemindert wird und somit eine fehlertolerantere und sicherere Implementation geschaffen werden kann.

TODO: ? Für eine bessere Wartbarkeit und Nachvollziehbarkeit soll die Implementation in Ihrer TODO: Struktur/Architektur der C++ Implementation ähneln.

1.4 Aufbau der Arbeit

Diese Arbeit ist im wesentlichen in die folgenden Themengebiete aufgeteilt: Grundlagen, Anforderungs- und Systemanalyse, Systementwurf und Implementation und Auswertung.

Im Themengebiet Grundlagen sollen wesentliche Bestandteile dieser Arbeit erläutert und erklärt werden. Hierzu zählt zum einen die Programmiersprache Rust in ihrer Entstehungsgeschichte TODO: ref, Garantien TODO: ref und Sprachfeatures TODO: ref, zum anderen die hochperformante, serverbasierte Kommunikationsplattform mit ihren Protokollen TODO: ref und dem Systemkontext in dem diese betrieben wird.

In der Anforderungs- und Systemanalyse wird der Kontext in dem das System betrieben werden soll genauer betrachtet. Umzusetzende funktionale und nicht-funktionale Anforderungen werden aufgestellt sowie eine Übersicht von Systemen mit denen interagiert wird.

Das Themengebiet Systementwurf und Implementation befasst sich mit dem theoretischen und praktischen Lösen der im vorherigen Kapitel aufgestellten Anforderungen. Aufgrund der Tatsache, dass es sich hierbei um eine alternative Implementation handelt, wird zur bestehenden C++ Implementation Bezug genommen. Auf architektonische Unterschiede im Systementwurf, die sich aufgrund von Sprach- und Bibliotheksunterschiede, werden hier genauer beschrieben.

Zuletzt wird eine Auswertung der Implementation aufgezeigt. TODO: michael.write_more();

2 Die Programmiersprache Rust

Rust ist eine Programmiersprache, die versucht performant – und daher durch Abstraktionen mit keinem zusätzlichen „Kosten“ TODO: ref zero cost abstractions – sichere Programmierung zu ermöglichen. Ziel ist eine TODO: Systemprogrammiersprache, die sowohl sicher TODO: cite chapter als auch performant ist und ohne eine Laufzeit ausgeführt werden kann. Verschiedene Fehlerquellen – wie „dangling pointers“, „double free“ oder „memory leaks“ TODO: ref – werden durch Abstraktionen und mit Hilfe des Compilers verhindert. Anders als Programmiersprachen, die dies mit Hilfe einer Laufzeit ermöglichen (zbsp. Java oder C#), wird dies in Rust durch eine statische Analyse und einem Eigentümerprinzip bei der Compilation gewährleistet.

2.1 Geschichte

In 2006 [Rusa] begann Graydon Hoare die Programmiersprache Rust in seiner Freizeit als Hobbyprojekt zu entwickeln. Als Grund nannte er seine Unzufriedenheit mit der Programmiersprache C++, in der es sehr schwierig sei, fehlerfreien, speichersicheren und nebenläufigen Code zu schreiben. Zudem beschrieb er C++ als „ziemlich fehlerträchtig“. [Sch13]

Auch Federico Mena-Quintero – Mitbegründer des Gnome projekts TODO: cite `https://people.gnome.org/~federico/` or so – äußerte in einem Interview mit Golem im Juli 2017 seine Bedenken an der Verwendung der „feindseligen“ Sprache C [Grü17]. In Vorträgen TODO: nix mehrzahl? vermittelt er seither, wie Bibliotheken durch Implementationen in Rust ersetzt werden können [Qui].

Ab 2009 begann Mozilla die Weiterentwicklung finanziell zu fördern, als einfache Tests und die Kernprinzipien demonstriert werden konnten. Die Entwicklung findet dabei öffentlich einsehbar auf GitHub unter `https://github.com/rust-lang/rust` statt und wird dabei nicht ausschließlich von Mozilla Angestellten koordiniert. Die Stabilität des Compilers trotz hoher Flexibilität während der Entwicklung wird durch Unterscheidung von drei Veröffentlichungskanälen – release, stable und nightly – in Kombination mit automatisierten Tests TODO: ref? gewährleistet. [Rusa]

TODO: hobbyprojekt, mozilla, open-source, Entwicklung auf GitHub - jeder kann sich beteiligen, test(coverage), automatisierte builds, stable/beta/nightly

2.2 Anwendungsgebiet

Das Ziel von Rust ist es, das Designen und Implementieren von sicheren, nebenläufig und auch praktisch tauglichen Systemen möglich zu machen [Rusa]. TODO: intro paragraph

Da Rust den LLVM[1]-Compiler nutzt, erbt Rust auch eine große Anzahl der Zielplattformen die LLVM unterstützt. Die Zielplattformen sind in drei Stufen unterteilt, bei denen verschieden stark ausgeprägte Garantien vergeben sind. Es wird zwischen

- „Stufe 1: Funktioniert garantiert“ (u.a. X86, X86-64),
- „Stufe 2: Compiliert garantiert“ (u.a. ARM, PowerPC, PowerPC-64) und
- „Stufe 3“ (u. a. Thumb)

unterschieden [Rusb]. Diese Unterscheidung wirkt sich auch auf die Stabilisierungsphase und Implementation neuer Funktionen aus (Beispiel „128-bit Integer Support“ [atu]).

2.3 Aufbau eines Projektverzeichnisses

2.3.1 Klassisch

```
src/
|-- main.rs
|-- functionality.rs
|-- module/
    |-- mod.rs
    |-- functionality.rs
    |-- submodule/
        |-- mod.rs
        |-- functionality.rs
```

Listing 2.1: Verzeichnisstruktur des Quelltext-Verzeichnisses

Das Quelldatei-Verzeichnis sollte entweder eine *main.rs* für Ausführbare Programme oder eine *lib.rs* für Bibliotheken enthalten. Während der Paketmanager Cargo (Unterabschnitt 2.3.2) eine solche Benennung als Standardkonvention erwartet, kann bei manueller Nutzung des Compilers auch ein anderer Name für die Quelldatei vergeben werden.

Der Compiler startet in der Wurzeldatei und lädt weitere Module, die durch `mod module;` gekennzeichnet sind (ähnlich `# include "module.h"` in C/C++). Ein Modul kann dabei eine weitere Quelldatei oder ganzes Verzeichnis sein. Ein Verzeichnis wird aber nur als Modul interpretiert, wenn sich eine *mod.rs* Datei darin befindet.

[1] Früher „Low Level Virtual Machine“ [Wik17], heute Eigenname; ist eine „Ansammlung von modularen und wiederverwendbaren Kompiler- und Werkzeugtechnologien“ [LLVa]. Unterstützt eine große Anzahl von Zielplatformen, u.a. X86, X86-64, PowerPC, PowerPC-64, ARM, Thumb, ... [LLVb].

2.3.2 Mit Cargo

TODO: text is shit Im Gegensatz zu einem klassischen Aufbau (Unterabschnitt 2.3.1) wird von der Rust Gemeinschaft das Werkzeug „Cargo“ (dt. Fracht/LadungTODO: .) angeboten. Mit Cargo können ähnlich wie zum Beispiel mit Maven TODO: cite? in Java, Abhängigkeiten zu anderen Bibliotheken verwaltet werden. Ein Cargo Projekt wird dabei als „Crate“ (dt. Kiste/KastenTODO: .) bezeichnet. Eine offizielles Verzeichnis befindet sich auf https://crates.io/. Von https://crates.io/ werden standardmäßig Abhängigkeiten nachgeladen. Jeder kann neue Bibliotheken hochladen/veröffentlichen, für den Namen gilt dabei „first come, first serve“.

```
crate/
|-- Cargo.toml
|-- src/
    |-- ...
```

Listing 2.2: Vereinfachte Verzeichnisstruktur einer „crate“

TODO: dependencies

TODO: Cargo init –bin <name>

TODO: missing .gitignore / .git mention / git alltogether

TODO: Cargo.toml

TODO: [crates.io], Anzahl Pakete

TODO: Compilierablauf, downloaden, compilieren von crates

2.4 Hello World

```
fn main() {
    println!("Hello World");
}
```

Listing 2.3: „Hello World“ in Rust

TODO: let, optionaler datentyp, macros, generics, () statt void

TODO: official format/naming convetion, use, function, macro

TODO: Variables, Structs, Enums, Traits

TODO: type safety langauge

TODO: Rust -> MIR -> assembler

TODO: MIR/assemblerbeispiele?

[Jim17]

2.5 Standardbibliothek

Die Rust Community ist darum bemüht, die Standardbibliothek sehr leichtgewichtig zu halten. Sie wird selbst als eine Crate (siehe Unterabschnitt 2.3.2) zur Verfügung gestellt, auf die standardmäßige eine Abhängigkeit besteht. Für die Verwendung von Rust im Embedded Bereich, kann diese Abhängigkeit, die für Microcontroller sehr umfangreich ist, durch `#![no_std]` unterbunden werden. Daraufhin sind nur noch die in der `core` Crate zur Verfügung gestellten Sprachkonstrukte verwendbar.

2.5.1 core

Die in der `core` Crate zur Verfügung gestellten Sprachkonstrukte sind im wesentlichen die üblichen Verdächtigen: `bool` für boolische Ausdrücke; `char` für ein einzelnes Unicode Zeichen; `str` für eine Zeichenkette; `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64`, `i64`, (bald `u128`, `i128` TODO: cite) und `usize`, `isize` für ganze Zahlen; `f32`, `f64` für Fließkommazahlen in einfacher und zweifacher Präzision; Arrays und Slices [Rusc].

Ganzzahlige primitive Datentypen mit `u` beginnend sind vorzeichenlos und mit `i` beginnend sind vorzeichenbehaftet, gefolgt mit der Anzahl der Bits die der Datentyp groß ist. TODO: shit sentence Die einzige Ausnahme bildet der Datentyp `usize` bzw `isize`, da dieser immer so groß ist, wie die Architektur der Zielplattform (X86 -> 32 Bit, X86_64 -> 64 Bit). Ein Anwendungsfall von `usize` ist dabei die Indexierung eines Arrays oder einer Slice (TODO: siehe nächster paragraph?), da der Index hierfür niemals negativ und niemals größer sein kann, wie die Architektur der Zielplattform darstellen kann TODO: erwähnen?: größer könnte man garnicht addressieren.

Durch dieses Schema bei der Bezeichnung der Datentypen wird eine Verwirrung wie zum Beispiel in C unterbunden, wo die primitiven Datentypen (`short`, `int`, `long`, ..) keine definierte Größe haben, sondern dies abhängig vom eingesetzten Compiler und der Zielplattform ist [DD13, S. 187]. Erst ab C99 wurden zusätzliche, aber optionale, ganzzahlige Datentypen mit bestimmter Größe definiert [GD14, S. 141].

Konstanten können eindeutig einem Datentyp zugewiesen werden, indem dieser angehängt wird. `4711u16` ist somit vom Datentyp `u16`. Des weiteren dürfen Ziffern durch beliebiges setzen von `_` getrennt werden, um die Lesbarkeit zu erhöhen: `1_000_000_f32`. Eine Schreibweise in Binär (`0b0000_1000_u8`), in Hexadezimal (`0xFF_08_u16`) oder Oktal (`0o64_u8`) ist auch möglich. Konstante Zeichen und Zeichenketten können auch als Bytes (`b'b'` entspricht `u8` und `b"abc"` entsricht `&[u8]`) TODO: hinterlegt werden.

Arrays haben immer eine zur Compilezeit bekannte Größe und Initialisierungswert (siehe Unterabschnitt 2.12.3). Dynamische Arrays gibt es nicht, da diese zu oft Fehlerquellen seien TODO: cite! (Abhilfe: `Vec<_>`, siehe Unterabschnitt 2.5.2). Die Notation ist `[<Füllwert>; <Größe>]`. `[0_u8; 128]` steht also für ein 128 Byte langes Byte Array, das mit 0-en vom Datentyp `u8` gefüllt ist.

„Slices“ (dt. Scheibe/Stück) bezeichnet Rust Referenzen auf Arrays, die auch nur Teilbereiche umfassen können. Die Größe einer Slice wird dabei mit der Referenz auf den Startwert gespeichert TODO: explain Fat-Pointer? und bei Funktionsaufrufen übergeben. Ein zusätzlicher Parameter für die Größe eines Buffers, wie in C üblich, ist somit unnötig. Die Notation ähnelt die eines Arrays, aber ohne Größenspezifikation: `[<Datentyp>]`. Eine Slice kann von einem Array oder einer anderen Slice erzeugt werden, dabei wird der Start- und Endindex des Teilbereiches angegeben. Falls kein Start- oder Endindex angegeben wurde, wird das jeweilige Limit übernommen (0, max) TODO: shit text: `let slice : &[u8] = &array[..8];`

Rust kennt `null` (-Pointer) nicht, bietet aber in `core` bereits `Option<_>` als Ersatz an. Für die Fehlerbehandlung wird nicht auf ein Exception-Handling zurückgegriffen, sondern ein eigener Datentyp angeboten, der entweder den Rückgabewert enthält, oder aber einen Fehler: `Result<_, _>`. Die Funktionsweise und die Vorteile von `Option<_>` und `Result<_, _>` wird in Unterabschnitt 2.12.4 genauer erklärt.

2.5.2 std

Die Crate `std` erweitert `core` um viele TODO: collections etc. `Vec<_>` `HashMap<_, _>` `String` `Box` TODO: heap dinge

TODO: core, datatypes, arrays slices, no null „billion dollar mistake“

TODO: std, Vec, str, String, no_std für embedded

TODO: println!, writeln! formatting

2.6 Alles hat einen Rückgabewert

2.7 use mod pub

2.8 Eigentümer- und Verleihprinzip

2.9 Scope / Memory Management, Lebenszeit

TODO: autodrop, auto file close

2.10 Rust als funktionale Programmiersprache

TODO: functional programming -> no global state, no exceptions, find literature TODO: prove via code

2.11 Rust als Objekt-Orientierte Programmiersprache

TODO: trait TODO: prove via design patterns, a few? from faq:: Is Rust object oriented? It is multi-paradigm. Many things you can do in OO languages you can do in Rust, but not everything, and not always using the same abstraction you're accustomed to.

2.12 Versprechen von Rust

2.12.1 Sichere Nebenläufigkeit

TODO: Send, Sync, No dataraces weil Ownership, Channel, Mutex, RwLock

2.12.2 Zero Cost Abstraction

2.12.3 Kein undefiniertes Verhalten

TODO: auch: no unitialized usage TODO: ref oreilly

2.12.4 Keine vergessene Null-Pointer Prüfung

TODO: explain option

2.12.5 Kein vergessene Fehlerprüfung

`Result<_, _>` TODO: explain result

2.12.6 No dangling pointer

TODO: src https://www.youtube.com/watch?v=d1uraoHM8Gg

2.12.7 Statische Speicher- und Lebenszeitanalyse

TODO: while compiling, does not compile on error / unprovable code, trait Drop

2.13 Einbinden von Bibliotheken

Externe Datentypen

Rust bietet durch das Foreign Function Interface[2] die Möglichkeit, andere (System-)Bibliotheken einzubinden. Entsprechende Strukturen und Funktionen werden durch einen `extern`-Block oder im Falle von Strukturen stattdessen optional mit einem `#[repr(C)]` gekennzeichnet.

In einem Beispiel, soll die Nutzung von Foreign Function Interface demonstriert werden.

```
1 typedef struct PositionOffset {
2     long position_north;
3     long position_east;
4     long *std_dev_position_north; // OPTIONAL
5     long *std_dev_position_east;  // OPTIONAL
6
7     // ...
8 } PositionOffset_t;
```

Listing 2.4: Ausschnitt von „PositionOffset“ TODO: ref mecview lib in C, autgen ASN

Die Struktur in Listing 2.4 muss zur Nutzung in Rust zuerst bekannt gemacht werden. Dabei gibt es mehrere Möglichkeiten:

1. Falls der Aufbau der Struktur nicht von Bedeutung ist, kann es ausreichen, den Datentyp lediglich bekannt zu machen: `#[repr(C)] struct PositionOffset;`
2. Der Aufbau ist wie bei Punkt 1 unbedeutend, es soll aber ausdrücklich auf einen externen Datentyp hingewiesen werden: `extern { type PositionOffset; }` [Ruse] (TODO: nightly)
3. Der Inhalt der Struktur ist von Bedeutung, da darauf zugegriffen werden soll oder in Rust eine Instanz erzeugbar sein soll. In diesem Fall muss die Struktur komplett wiedergegeben werden:

[2] Beschreibt den Mechanismus wie ein Programm das in einer Programmiersprache geschrieben ist, Funktionen aufrufen kann, die einer einer anderen Programmiersprache geschrieben wurden. [Wik18]

```rust
use std::os::raw::c_long;

#[repr(C)]
pub struct PositionOffset {
    pub position_north: c_long,
    pub position_east: c_long,
    pub std_dev_position_north: *mut c_long,
    pub std_dev_position_east: *mut c_long,
    // ...
}
```

Listing 2.5: Ausschnitt von „PositionOffset“ TODO: ref mecview lib in Rust

In Listing 2.5 ist die Struktur „PositionOffset“ definiert, die durch das Attribut `#repr(C)` wie eine C-Struktur im Speicher organisiert wird. Somit ist sie kompatibel zu der C-Struktur aus Listing 2.4.

Wenn auf eine C-Struktur zugegriffen wird, sollten auch, wie in Listing 2.5 zu sehen, spezielle Datentypen (`c_long`, `c_void`, `c_char`, ...) verwendet werden, um die Kompatibilität mit verschiedenen Systemen und C-Compilern zu wahren. TODO: u32 immer 32bit, aber int nicht immer gleich (Beispiel!?) –> Probleme

Ein C-Pointer `*long` wird in Rust „Raw-Pointer“ genannt und entweder `*mut c_long` oder `*const c_long` geschrieben. Der Unterschied ist wie zwischen `&mut c_long` und `&c_long` und dient dem TODO: Rusttypsystem!? ref!? zur Unterscheidung TODO: Erzwinungt im Besitz von entsprechender Mutability zu sein, während es für die C-Seite keinen Unterschied macht [Rusd]:

Referenz in Rust	Raw-Pointer in Rust	C-Pointer
`&mut c_long`	`*mut c_long`	`long*`
`&c_long`	`*const c_long`	`long*`

Abbildung 2.1: Vergleich Rust Raw-Pointer und Referenz zu C-Pointer

Externer Funktionsaufruf

Während eine Struktur, die eine externe Struktur wiedergibt, sich optional in einem `extern {}` Block befinden kann, ist es zwingend, eine externe Funktionen darin bekannt zu machen:

```
use std::os::raw::c_void;

#[link(name = "messages", kind = "static")]
extern {
    type asn_TYPE_descriptor_s;
    type asn_enc_rval_t;

    fn uper_encode_to_buffer(
        type_descriptor: *const asn_TYPE_descriptor_s,
        struct_ptr: *const c_void,
        buffer: *mut c_void,
        buffer_size: usize,
    ) -> asn_enc_rval_t;
}
```

Listing 2.6: Externe Funktionsdefinition der ASN.1 Funktion zum Enkodieren

Wie in Listing 2.6 zu sehen ist, können auch `extern {}` Blöcke mit Attributen versehen werden. Zwingend ist bei der Verwendung eines `#[link(..)]` Attributes der Name der Bibliothek, auf die sich der im `extern {}` Block stehende Code bezieht. Optional kann auch wie in Listing 2.6 die Art der TODO: Linkung (dylib, static) angegeben werden.

Die Art der Definition einer externen Funktion unterscheidet sich nicht von einer normalen Funktionsdefinition. Es sollten aber, wie in Abschnitt 2.13 beschrieben, zu C bzw. der externen Sprache kompatiblen Datentypen verwendet werden.

2.14 Warum Rust?

> *„[..]Leute, die [..] sichere Programmierung haben wollen, [..] können das bei Rust haben, ohne die [von D] undeterministischen Laufzeiten oder Abstraktionskosten schlucken zu müssen. “* [Lei17]

> *„It's not bad programmers, it's that C is a hostile language“* (Seite 54, [Qui])

> *„I'm thinking that C is actively hostile to writing and maintaining reliable code“* (Seite 129, [Qui])

> *„[..] Rust makes it safe, and provides nice tools“* (Seite 130, [Qui])

> *„Rust hilft beim Fehlervermeiden“* [Grü17]

> *„Rust is [..] a language that cares about very tight control“* [fgi17]

TODO: unused orly rust [Bla15]

2.15 Kernfeatures

https://www.youtube.com/watch?v=d1uraoHM8Gg
TODO: no need for a runtime, all static analytics
TODO: memory safety
TODO: data-race freedom
TODO: active community
TODO: concurrency: no undefined behavior
TODO: ffi binding Foreign Function Interface
TODO: zero cost abstraction
TODO: package manager: cargo

https://www.youtube.com/watch?v=-Tj8Q12DaEQ
TODO: static type system with local type inference
TODO: explicit notion of mutability
TODO: zero-cost abstraction *(do not introduce new cost through implementation of abstraction)
TODO: errors are values not exceptions TODO: no null
TODO: ßtatic automatic memory management no garbage collection
TODO: often compared to GO and D (44min)

2.16 Schwächen

https://www.youtube.com/watch?v=-Tj8Q12DaEQ
TODO: compile-times
TODO: Rust is a vampire language, it does not reflect at all!
TODO: depending on the field -> majority of libraries?

2.17 Performance Fallstricke

TODO: [Llo]

2.18 Beispiele von Verwendung von Rust

TODO: firefox

https://www.youtube.com/watch?v=-Tj8Q12DaEQ
TODO: GTK binding heavily to rust

TODO: unstable TODO: ffi

3 Stand der Technik (c++ Version)

3.1 Hochperformant -> parallel?

3.2 Serverbasierte Kommunikationsplattform

3.3 Low-Latency + Entwurfsmuster + Patterns? + Algorithmen?

3.4 ASN.1

3.5 PER

3.6 MEC-View Server und Umgebung

4 Anforderungen

4.1 Funktionale Anforderungen

4.2 Nichtfunktionale Anforderungen

4.3 Kein Protobuf weil

5 Systemanalyse

5.1 Systemkontextdiagramm

5.2 Schnitstellenanalyse

5.3 C++ Referenzsystem

6 Systementwurf

6.1 Änderungen bedingt durch Rust

7 Implementierung

TODO: Schwierigkeiten: FFI binding, manuell -> meh, also generieren

8 Auswertung

9 Zusammenfassung und Fazit

Literatur

[atu] aturon. GitHub. Tracking issue for 128-bit integer support (RFC 1504). Englisch. URL: https://github.com/rust-lang/rust/issues/35118#issuecomment-278078118 (besucht am 19. 02. 2018).

[Bla15] Jim Blandy. Why Rust? Trustworthy, Concurrent System Programming. Englisch. 2015. URL: http://www.oreilly.com/programming/free/files/why-rust.pdf (besucht am 01. 06. 2017).

[DD13] P.J. Deitel und H. Deitel. C for Programmers with an Introduction to C11. Deitel Developer Series. Pearson Education, 2013. ISBN: 9780133462074.

[fgi17] fgilcher. Subreddit Rust. fgilcher kommentiert. Englisch. 3. Nov. 2017. URL: https://www.reddit.com/r/rust/comments/7amv58/just_started_learning_rust_and_was_wondering_does/dpb9qew/ (besucht am 14. 02. 2018).

[GD14] J. Goll und M. Dausmann. C als erste Programmiersprache: Mit den Konzepten von C11. SpringerLink : Bücher. Springer Fachmedien Wiesbaden, 2014. ISBN: 9783834822710.

[Grü17] Sebastian Grüner. „C ist eine feindselige Sprache“. Der Mitbegründer des Gnome-Projekts I Deutsch. 22. Juni 2017. URL: https://www.golem.de/news/rust-c-ist-eine-feindselige-sprache-1707-129196.html (besucht am 14. 02. 2018).

[Jim17] Jason Orendorff Jim Blandy. Programming Rust. Fast, Safe Systems Development. O'Reilly Media, Dez. 2017. ISBN: 1491927283.

[Lei17] Felix von Leitner. Fefes Blog. D soll Teil von gcc werden. Deutsch. 22. Juni 2017. URL: https://blog.fefe.de/?ts=a7b51cac (besucht am 14. 02. 2018).

[Llo] Llogiq. Llogiq on stuff. Rust Performance Pitfalls. Englisch. URL: https://llogiq.github.io/2017/06/01/perf-pitfalls.html (besucht am 14. 02. 2018).

[LLVa] LLVM.org. The LLVM Compiler Infrastructure Project. LLVM Overview. Englisch. URL: https://llvm.org/ (besucht am 19. 02. 2018).

[LLVb] LLVM.org. The LLVM Compiler Infrastructure Project. LLVM Features. Englisch. URL: https://llvm.org/Features.html (besucht am 19. 02. 2018).

[MEC] MEC-View. MEC-View. Deutsch. URL: http://mec-view.de/ (besucht am 19. 02. 2018).

[Qui] Federico Mena Quintero. Replacing C library code with Rust. What I learned with librsvg. Englisch. URL: https://people.gnome.org/~federico/blog/docs/fmq-porting-c-to-rust.pdf (besucht am 14. 02. 2018).

[Rusa] Rust. The Rust Programming Language. Englisch. URL: https://www.rust-lang.org/en-US/faq.html (besucht am 16. 02. 2018).

[Rusb] Rust. The Rust Programming Language. Rust Platform Support. Englisch. URL: https://forge.rust-lang.org/platform-support.html (besucht am 19. 02. 2018).

[Rusc] Rust-Lang/Book. The Rust Programming Language. Primitive Types. Englisch. URL: https://doc.rust-lang.org/book/first-edition/primitive-types.html (besucht am 21. 02. 2018).

[Rusd] Rust-Lang/Book. The Rust Programming Language. Unsafe Rust. Englisch. URL: https://doc.rust-lang.org/book/second-edition/ch19-01-unsafe-rust.html#dereferencing-a-raw-pointer (besucht am 20. 02. 2018).

[Ruse] Rust-Lang/RFCs. GitHub. Tracking issue for RFC 1861: Extern types. Englisch. URL: https://github.com/rust-lang/rust/issues/43467 (besucht am 20. 02. 2018).

[Sch13] Julia Schmidt. Graydon Hoare im Interview zur Programmiersprache Rust. Deutsch. 12. Juli 2013. URL: https://www.heise.de/-1916345 (besucht am 16. 02. 2018).

[Wik17] Wikipedia. LLVM — Wikipedia, Die freie Enzyklopädie. 2017.

[Wik18] Wikipedia. Foreign function interface — Wikipedia, The Free Encyclopedia. 2018.

Glossar

Foreign Function Interface Beschreibt den Mechanismus wie ein Programm das in einer Programmiersprache geschrieben ist, Funktionen aufrufen kann, die einer einer anderen Programmiersprache geschrieben wurden. [Wik18] . 11, 14

LLVM Früher „Low Level Virtual Machine“ [Wik17], heute Eigenname; ist eine „Ansammlung von modularen und wiederverwendbaren Kompiler- und Werkzeugtechnologien“ [LLVa]. Unterstützt eine große Anzahl von Zielplatformen, u.a. X86, X86-64, PowerPC, PowerPC-64, ARM, Thumb, ... [LLVb]. . 6

Abkürzungsverzeichnis

BMWi Bundesministerium für Wirtschaft und Energie. 2

FBA Fehler Baum Analyse. 3

MEC Mobile Edge Computing basierte Objekterkennung für hoch- und vollautomatisches Fahren. 2, 3

SIL Safety Integrity Level. 3

Abbildungsverzeichnis