Technische Hochschule Ingolstadt

Standardisierte Darstellung von realen Short-Range-Radar Sensordaten durch das Open Simulation Interface in einer ROS2 Fahrzeugarchitektur

Technische Hochschule Ingolstadt

Fakultät Informatik

eingereichte Bachelorarbeit im Studiengang Flug- und Fahrzeuginformatik (FFI)

Christoph Dominic Sell

Matrikel-Nr. 00108676

geboren am 29.08.2000 in Kösching

Erstprüfer:	Prof. Dr.-Ing. Werner Huber
Zweitprüfer:	Prof. Dr. techn. Priv.-Doz. Andreas Riener
Betreuer:	Georg Seifert
Beginn der Arbeit:	29.11.2022
Abgabe der Arbeit:	16.02.2023

Prüfungsrechtliche Erklärung

Ich erkläre hiermit, dass ich die Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ingolstadt, den 16.02.2023

Sell Christoph

....

Christoph Dominic Sell

Kurzfassung

Autonomes Fahren wird ein immer wichtigeres Thema. Um dies zu ermöglichen, müssen alle Sensordaten im Fahrzeug über eine gemeinsame Schnittstelle bereitgestellt werden, damit jene Daten gleichermaßen verwendet werden können. Diese werden dann zentral ausgewertet, um damit Entscheidungen bezüglich des Fahrverhaltens zu treffen. Das Open Simulation Interface (OSI) stellt eine solche standardisierte Software-Schnittstelle dar. Dieses bietet ein einheitliches Format zum Austausch von Daten innerhalb von Sensormodellen und Simulationen an. Aber lassen sich darin auch reale Sensordaten, wie die eines Short-Range Radars, abbilden? Dies wird anhand des Radartyps SR73 der Firma Nanoradar in dieser Arbeit erforscht. Um diese Frage zu beantworten, wird zunächst eine Zuordnung der Radardaten zu den Nachrichtendefinitionen in OSI getroffen. Dabei lässt sich feststellen, dass nicht alle, aber bereits viele Informationen des Radars dargestellt werden können. OSI stellt einige Attribute zur Verfügung, welche das Radar nicht bereitstellt. Anschließend wird die Zuordnung programmiertechnisch umgesetzt. Nach der Umwandlung sollen die Daten in einer ROS2 Fahrzeugarchitektur weiterverwendet werden. Daher werden mithilfe der Programmiersprache C++ ein ROS2-Knoten sowie weitere Klassen implementiert, um das Lesen der Radarnachrichten über den CAN zu ermöglichen. Anschließend werden diese Sensordaten in das OSI Format konvertiert und schließlich über ROS verschickt. Zuletzt wird die Software noch funktional mittels gtest und nichtfunktional mittels Performancetests evaluiert. Ebenfalls wird überprüft, ob diese Software Speicherlücken aufweist. Dabei soll vor allem Augenmerk auf die verschiedenen Klassen und Funktionen gelegt werden.

Eine Zuordnung ist demnach großteils möglich. Objektdaten des Radars lassen sich ohne Probleme in OSI darstellen, Konfigurations- und Statusnachrichten hingegen nicht.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Codeverzeichnis

Abkürzungsverzeichnis

Abkürzung	**Bedeutung**
CAN	Controller Area Network
CPU	Central Processing Unit
KCD	Kayak CAN definition
NVM	Non-Volatile Memory
OSI	Open Simulation Interface
RCS	Radar Cross Section
ROS	Robot Operating System
SNR	Signal-to-Noise Ratio

1 Einleitung

50 % der Deutschen halten das automatisierte Fahren für eine sinnvolle Weiterentwicklung [4, S. 24]. 30 % würden sogar ein autonomes Fahrzeug benutzen [5, S. 30]. Die Vorteile davon sind, laut einer weiteren Umfrage, die verbesserte Mobilität für Menschen mit körperlichen Beeinträchtigungen, eine optimale Routenplanung und Streckenführung, weniger Stress und entspannteres Fahren, ein besserer Verkehrsfluss sowie die Möglichkeit, andere Dinge während einer Fahrt tun zu können [46]. Dabei sind wir aktuell noch weit davon entfernt, voll autonome Fahrzeuge auf den Straßen fahren zu lassen.

Der Unterschied zwischen automatisiertem und autonomen Fahren liegt darin, ob Fahrzeuge noch auf Menschen angewiesen sind, welche gewisse Entscheidungen vor oder während der Fahrt treffen, oder ob all diese gänzlich von dem Fahrzeug übernommen werden. In diesem Fall spricht man von autonomen Fahren [3].

Die Entwicklung des automatisierten Fahrens ist in Deutschland bereits in vollem Gange. Mercedes hat seit 17.05.2022 zwei Fahrzeuge im Angebot, die S-Klasse und den EQS. Beide erfüllen die Vorlagen des Gesetzesentwurfes vom 18. Juli 2021, welches das hochautomatisierte Level-3-Fahren bis 60 $\frac{\text{km}}{\text{h}}$ ermöglichte [49].

Seit 1. Januar 2023 ist die limitierende Geschwindigkeit derer nun nicht mehr 60 $\frac{\text{km}}{\text{h}}$, sondern 130 $\frac{\text{km}}{\text{h}}$ [40]. Daher werden neben Mercedes auch andere Marken nachziehen und hochautomatisierte Fahrzeuge bauen. Warum jedoch ist das nicht schon möglich, oder anders gefragt, weshalb dauert die Entwicklung von solchen Fahrzeugen so lange und warum gibt es nicht schon längst autonome Fahrzeuge im Straßenverkehr? Um diese Frage zu klären, ist es wichtig zu verstehen, was ein höherer Grad an Automatisierung für die Hersteller und Entwickler bedeutet. Funktionen wie der Abstandstempomat (ACC), Notbremsassistenten sowie Spurhalteassistenten existieren bereits und sind in den meisten Fahrzeugen heutzutage verbaut.

Diese Systeme sind, trotz der gelegentlichen Nutzung gemeinsamer Sensoren, größtenteils unabhängig voneinander. Das hat den Vorteil, dass sie separat entwickelt und getestet werden können [21].

Ein höherer Grad an Automatisierung würde bedeuten, dass der Austausch der Messdaten der unterschiedlichen Sensoren über eine einheitliche Schnittstelle geschehen muss, damit die Daten, welche aktuell meist nur von einem Subsystem verwendet werden, fusioniert werden können, um diese anschließend in einer zentralen Einheit auszuwerten und auf Basis derer dann Entscheidungen bezüglich des Fahrverhaltens zu treffen. Die einheitliche Schnittstelle zum Austausch stellt jedoch neben einer zentralen Steuereinheit ein großes Problem dar. Wie können die Daten separater Assistenzsysteme durch einen gemeinsamen Datenaustausch bereitgestellt und somit ein Gesamtsystem entwickelt werden? Wie kann dieses anschließend getestet und damit sichergestellt werden, dass alle Systeme korrekt von dieser zentralen Steuereinheit angesprochen werden? Genau das sind die aktuellen Fragestellungen, welche es im Hinblick auf das autonome Fahren zu klären gibt.

1.1 Motivation und Relevanz

Dafür sind gemeinsame Schnittstellen im Fahrzeug, welche es ermöglichen, diese Daten auszutauschen, notwendig.

Eine mögliche Schnittstelle bietet das Open Simulation Interface. Dieses spezifiziert eine standardisierte Software-Schnittstelle zu Sensormodellen und Fahrsimulatoren und stellt ein gemeinsames Datenaustauschformat bereit [6]. Trotz des ursprünglichen Entwicklungszwecks, diese Daten in einer Simulation auszutauschen, kann es auch genutzt werden, um die Daten ohne Simulation zu erzeugen und zu verschicken.

Exakt in diese Richtung wird gerade im C-IAD geforscht. C-IAD, welches die Kurzform von *CARISSMA Institute of Automated Driving* ist, ist ein Forschungsinstitut des CARISSMA's. Dieses wiederum ist das *Center of Automotive Research on Integrated Safety Systems and Measurement Area* und ein Forschungs- und Testzentrum der Technischen Hochschule Ingolstadt (THI). Im Institut wird aktuell ein BMW M8 Competition als Versuchsfahrzeug mit mehreren Sensoren ausgestattet, welche für das hochautomatisierte Fahren benötigt werden. Darunter zählen verschiedene Kameras, Radare sowie ein Lidar. Das Ziel dabei ist es, all diese Sensordaten während einer realen Fahrsimulation verwenden zu können.

1.2 Problembeschreibung

Um diese Daten jedoch alle gemeinsam zu nutzen, ist ein einheitliches Datenformat für alle Sensoren notwendig. Dafür wird das Open Simulation Interface ausgewählt. Da das Open Simulation Interface ursprünglich für einen anderen Verwendungszweck entwickelt wurde, für diese Arbeit jedoch nur die darin enthaltene Beschreibungsspache als einheitliches Format verwendet wird, ist es daher notwendig, reale Sensordaten darin abbilden zu können.

Dies wird in dieser Arbeit anhand von Daten eines realen Short-Range Radars evaluiert. Dabei ist es zunächst wichtig zu klären, ob reale Sensordaten grundsätzlich durch das Open Simulation Interface Format darstellbar sind. Darüber hinaus ist die Frage zu stellen, welche Informationen das Radar bereitstellt und in dem Open Simulation Interface abgebildet werden können. Angenommen wird hier zunächst, dass es nicht möglich ist, alle Daten tatsächlich in das Open Simulation Interface Datenformat umzuwandeln.

1.3 Ziele der Arbeit und Methodik

Ziel der Arbeit ist deshalb, jene Fragestellung aus Abschnitt 1.2 zu beantworten. Dabei wird versucht, zunächst eine Zuordnung der verschiedenen Daten zu erstellen, worin ersichtlich wird, welche Informationen des Radars zu welchen im Open Simulation Interface zugeordnet werden können und bei welchen dies nicht umsetzbar ist. Anschließend soll diese theoretische Zuordnung noch praktisch umgesetzt und daher implementiert werden, um tatsächlich bestätigen zu können, dass eine Umwandlung möglich ist. Dies soll abschließend noch durch unterschiedliche Tests bestätigt werden.
Im Folgenden wird der Aufbau der kompletten Arbeit erläutert und auf die verschiedenen Kapitel verwiesen.

1.4 Aufbau der Arbeit

Kapitel 1 liefert eine Einführung in das Thema Open Simulation Interface, ebenso die Motivation und das Ziel dieser Abschlussarbeit.
Anschließend folgt Kapitel 2 mit den nötigen Grundlagen für diese Arbeit und dem aktuellen Stand der Forschung in den verschiedenen Themen.
In Kapitel 3 wird dann auf eine mögliche Datenzuordnung der Radardaten mit den Nachrichten des Open Simulation Interfaces eingegangen.
Kapitel 4 handelt dann schließlich von der Implementierung der Zuordnung. Dabei werden zunächst Vorüberlegungen und die Vorarbeiten erläutert, anschließend wird detailliert beschrieben, wie die Software implementiert wurde.
Um die korrekte Funktionalität jener nachzuweisen, werden in Kapitel 5 verschiedene funktionale und nichtfunktionale Tests durchgeführt.
Zuletzt enthält Kapitel 6 noch eine Zusammenfassung der Ergebnisse, ein Fazit sowie einen Ausblick.

2 Grundlagen und Stand der Forschung

Im Folgenden werden die Grundlagen zu Radar, dem Controller Area Network, Open Simulation Interface und dem Robot Operating System beschrieben und die relevanten Zusammenhänge für diese Arbeit dargestellt. Bei Radar wird dazu auf den Einsatzbereich und die Eigenschaften eingegangen. Zudem werden die Grundlagen des Bus-Protokolls CAN beschrieben, das für den hier verwendeten Radar als Verbindungsmedium eingesetzt wird. Anschließend wird das Open Simulation Interface hinsichtlich dessen Verwendung in Simulationen und als einheitliches Austauschformat dargestellt. Zuletzt wird dann noch ROS2 als Übertragungsmedium für die Nachrichten des Open Simulation Interfaces erläutert, wobei der Aufbau und die Verwendung erklärt werden. Zusätzlich dazu wird bei allen Themen kurz der aktuelle Forschungsstand angesprochen.

2.1 Radar

Mithilfe eines Radars (kurz für *Radio Detection and Ranging*) können Objekte erkannt sowie deren Parameter durch elektromagnetische Wellen bestimmt werden. Darunter zählen zum Beispiel die Lage eines Objekts, dessen Bewegungszustand und Beschaffenheit [15, S. 1].

Eine Sendeantenne strahlt dabei elektromagnetische Wellen gebündelt in den Raum ab. Beim Auftreffen dieser auf ein Objekt wird ein geringer Teil der Wellen zum Radar zurück reflektiert, welche daraufhin von einer Empfangsantenne erfasst werden. Nach Verstärkung dieses Eingangssignals können daraus die Parameter der Objekte errechnet werden. Diese Informationen können dabei aus der Intensität des Streufeldes (Größe des Objektes), der räumlichen Lage der Phasenfronten (Winkel des Objektes zum Radar), der Ankunftszeit des pulsförmigen Signals (Entfernung) sowie der Änderung der Phase des eintreffenden Signals (Geschwindigkeit) abgeleitet werden [15, S. 2].

Aus den genannten Gründen werden Radare zur Navigation von Flugzeugen, Schiffen und auch im Landverkehr eingesetzt [15, S. 4].

Dort können Radare neben der Erkennung anderer Objekte diese auch klassifizieren, voneinander unterscheiden und deren Parameter ermitteln.

Oben genannte Eigenschaften wie Geschwindigkeit, Abstand und Winkel können ebenfalls von anderen Sensoren wie Kameras oder Ultraschallsensoren bestimmt werden, jedoch besitzen diese gegenüber einem Radar bestimmte Nachteile. Eine Kamera kann die Geschwindigkeit nur indirekt messen und bei Dunkelheit oder schlechter Sicht keine korrekten Informationen bereitstellen [11], ein Ultraschallsensor kann nur über eine kurze Distanz von einigen Metern eine korrekte Entfernung berechnen [56].

Da das Radar durch elektromagnetische Wellen Objekte erkennt, ist es weniger anfällig für schlechte Sicht. Bei Nebel, Regen oder auch Dunkelheit ist es in der Lage, auch bei größeren Reichweiten genaue Entfernungs- und Geschwindigkeitsangaben bereitzustellen [15, S. 1]. Dies ist ein entscheidender Grund für die Benutzung von Radaren in Fahrzeugen.

In diesen werden die drei Kategorien Long-Range, Mid-Range sowie Short-Range Radare eingesetzt. Long-Range Radare erfassen eine Entfernung von maximal 250 m und werden für die adaptive Geschwindigkeitsregulierung benutzt, wohingegen Short-Range nur einen Bereich von weniger als 30 m abdecken. Diese, sowie die sogenannten Mid-Range Radare, welche Objekte in einer maximalen Distanz von 100 m erkennen können, werden zur Kollisionswarnung, für den Toten-Winkel-Assistenten sowie zur Vermeidung von Auffahrunfällen bei stop-and-go verwendet [59].

Beim Radar wird aktuell neben der Verbesserung der eigentlichen Radartechnik mehr an Technologien rund um und mit dem Radar geforscht, wenn man die neusten Veröffentlichungen zum Thema Radar betrachtet. Daraus werden nachfolgend je ein Artikel zum Thema Sensorfusion und zum Thema Simulation kurz angesprochen. Eine tiefere Auseinandersetzung mit den Themen Sensorfusion und Simulation würde jedoch die Kapazitäten dieser Arbeit übersteigen, da hier lediglich die Radartechnologie verwendet wird.

In einer Publikation geht es um die Fusion von Kamera- mit Radardaten. Dabei stehen vor allem die Information über die Tiefe von Objekten im Vordergrund. Darin ist untersucht worden, wie die Kameradaten mit denen eines Radars im Automobil fusioniert werden können, um Echtzeitinformationen bezüglich der Tiefe zu erhalten. Quantitative Ergebnisse haben gezeigt, dass sich die Tiefenschätzungen, nahezu ohne zusätzlichen Rechenaufwand, verbesserten [2].

Da Radare bei Fahrassistenzsystemen eine wichtige Rolle spielen, müssen auch diese und deren Daten beim Entwicklungsprozess und in verschiedenen Fahrsituationen getestet werden. Da es kostentechnisch nicht realisierbar ist, alle möglichen Szenarien zu testen, werden immer mehr Sensoren virtualisiert. Eine Radarmodellierung bietet dann den entscheidenden Vorteil, dass die Funktionalität des Radars und dessen Messwerte in einer Simulation in diversen Situationen getestet werden kann. Ein solches Modell basiert auf dem *Mixture Density Network* (MDN). Es kann anschließend in eine Mehrkörpersimulationsplattform integriert und dessen Performanz mittels einer offenen Simulationsschnittstelle validiert werden [35].

Aktuelle automotive Radare arbeiten im 77 GHz-Band. Die Auswertung der Objekterkennungen geschieht dabei bereits durch das Radar. Die Objekte lassen sich anschließend bei

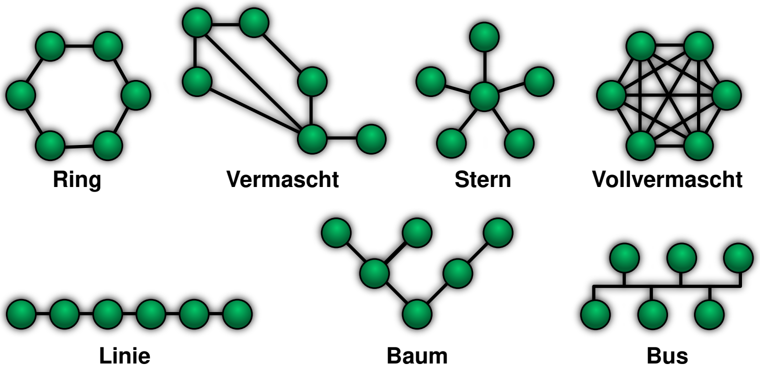

Abbildung 2.1: *Verschiedene Netzwerktopologien [13]*

geringer Datenrate mit CAN, bei detaillierten Informationen wie Cluster oder Rohdaten mittels breitbandiger Schnittstellen bereitstellen. Der hier eingesetzte Radar basiert auf CAN. Daher wird nachfolgend die Funktionsweise und Eigenschaften von CAN beschrieben.

2.2 CAN

Der CAN ist ein Fahrzeug-Bussystem und die Kurzform von *Controller Area Network*.
CAN ist von der Firma Bosch in den 80er Jahren entwickelt worden und deren Spezifikation stellt bis heute die Basis aller existierenden CAN-Implementierungen dar. Er ist ein bitstrom-orientierter Linienbus [60, S. 57 f.]. Ein Bus ist eine spezifische Anordnung von Geräten und Leitungen in einem Netzwerk (*Topologie*). Abbildung 2.1 zeigt unterschiedliche Netzwerktopologien.
CAN verwendet dabei einen CSMA/CR-Buszugriff. Dies bedeutet, dass Kollisionen beim Senden mehrerer Steuergeräte erkannt werden und im Falle eines gleichzeitigen Sendeversuchs die Nachricht mit der niedrigeren ID weiter über den Bus verschickt wird. Die anderen Sender erkennen diese Kollision und starten nach jener Nachricht einen erneuten Sendeversuch. Dafür muss es möglich sein, innerhalb einer Bitzeit eine solche Kollision zu erkennen. Daraus resultiert, dass die Buslänge begrenzt und umso kürzer sein muss, je höher die Bitrate ist [60, S. 58].
Beim CAN beschreiben verschiedene Standards unterschiedliche Ausführungen des Busses. Dabei werden beispielsweise die Bitrate sowie die Datenlänge unterschieden.
ISO 11898-2 definiert mit Bitraten $\geq 250\,\frac{\text{kBit}}{\text{s}}$ (*High Speed CAN*) eine verdrillte Zwei-Draht-Leitung mit einer maximalen Leiterlänge von 30cm zu den einzelnen Steuergeräten. Dieser wird beispielsweise im Antriebsstrang mit Bitraten von typischerweise $500\,\frac{\text{kBit}}{\text{s}}$ verwendet. ISO 11898-3 hingegen spezifiziert für Bitraten von $\leq 125\,\frac{\text{kBit}}{\text{s}}$ (*Low Speed CAN*) ebenfalls eine Zweidrahtleitung, welche Anwendung in der Karosserieelektronik findet [60, S. 59].
Eine CAN-Nachricht (frame) besteht grundsätzlich aus Header, Payload und Trailer [60,

S. 61 f.]. Für diese Arbeit relevant sind der Message Identifier (ID) im Header und die Payload, welche maximal 8 Bytes enthalten kann und die eigentlichen Nutzdaten darstellt.

Der CAN-Bus besitzt mehrere Vor- und Nachteile. Ein Vorteil ist, dass der Verkabelungsaufwand im Vergleich zu einer vollvermaschten Anordnung aller Geräte sehr gering ist. Dies spielt in Fahrzeugen eine wichtige Rolle, weil dadurch auch die Kosten durch weniger Kabelverbrauch und dem daraus resultierenden leichteren Gewicht verringert werden. Dies wird durch das verwendete Übertragunsmedium, einer Zweidrahtleitung, begünstigt. Außerdem können nachträglich weitere CAN-Knotenpunkte zu einem bestehenden Bus hinzugefügt oder auch daraus entfernt werden, da es keinen zentralen „Master-Knoten“ gibt, welche die Buszugriffe steuert, wie es beim *LIN-Bus* der Fall ist. Dies bewirkt, dass durch einen Ausfall eines einzelnen Knotens nicht der gesamte Bus ausfällt.

Ein Nachteil ist, dass die Leitungslänge aufgrund der elektrischen Eigenschaften limitiert ist. Außerdem ist es (besonders bei höheren Übertragungsgeschwindigkeiten) wichtig, dass der Bus durch einen Abschlusswiderstand terminiert wird, um Signalreflexionen zu verhindern [58].

Beim CAN wird vermehrt zum Thema Security geforscht. Da der CAN, dank immer mehr Sensoren, dem daraus resultierenden Informationsfluss und der Implementierung neuer Funktionen, verschiedene Angriffsmöglichkeiten bietet, ist es wichtig, diese zu erkennen und gezielt Gegen- oder Schutzmaßnahmen zu finden.

So wird beispielsweise in der ABS-Sicherheit geforscht. Damit der CAN während eines Bremsversuches nicht manipuliert oder gehackt und somit ein sicherer Bremsvorgang gewährleistet werden kann, ist ein Angriffserkennungsverfahren entworfen worden, um sowohl Sensorangriffe auf die für das ABS benötigten Sensoren, als auch CAN-Bus-Angriffe, zu erkennen. Dieses Verfahren basiert auf einer Fahrzeugzustandsraumgleichung, welche Echtzeit-Straßenreibkoeffizienten berücksichtigt, um Fahrzeugzustände, wie Raddrehzahl und Längsbremskraft, vorherzusagen. Simulationsergebnisse haben hierbei gezeigt, dass deren Angriffserkennungsverfahren Sensor- und CAN-Bus-Angriffe genauer erkennen und dadurch die Auswirkungen des Angriffs auf ein ABS-System nahezu eliminiert werden können [29].

Ein anderer Ansatz ist die Verwendung eines Echtzeit-Eindringungserkennungs-Systems (*real-time intrusion detection systems*, kurz IDS). Systeme dieser Art beschränken sich derzeit in der Automobilindustrie auf Techniken des maschinellen Lernens (*machine learning*). Da diese meist zeitaufwändig zum Erlernen der Algorithmen und Strukturen sind und Hardwareeinschränkungen aufweisen, wird auf *Quantum Annealing* zurückgegriffen, um diese Probleme zu umgehen. Dies ist vereinfacht gesagt ein auf Quantenphysik basierter Optimierungsprozess, um ein globales Minimum einer Funktion zu finden. Ergebnisse haben hierbei gezeigt, dass der Algorithmus deutlich schneller einen Angriff erkennt als ein klassischer Klassifikationsalgorithmus und gleiche Werte im Hinblick auf die Erkennungsgenauigkeit aufweist [8].

Da CAN nur ein Übertragungsmedium ist, braucht man eine Beschreibung der Daten. Dies kann mittels der erwähnten CAN-Datenbank für CAN geschehen. Die daraus interpretierten Werte können allerdings auch anderen Teilnehmern, wie beispielsweise dem standardisierten Open Simlation Interface, welches im nächsten Kapitel erläutert wird, bereitgestellt werden.

2.3 Open Simulation Interface

Das Open Simulation Interface, welches im Folgenden durch *OSI* abgekürzt wird, ist eine Spezifikation für Schnittstellen einer verteilten Simulation. Es ist ursprünglich durch BMW im PEGASUS Projekt initiiert und 2019 an ASAM e. V. (Association for Standardization of Automation and Measuring Systems) übertragen worden [33]. ASAM ist im Dezember 1998 in Stuttgart gegründet worden [18] und ist eine gemeinnützige Organisation, welche die Standardisierung von Werkzeugen in der automobilen Entwicklung sowie des Testens vorantreibt [17].

OSI ist genauer gesagt eine generische Schnittstellenbeschreibung zwischen Sensormodellen, Fahrsimulatoren sowie hochautomatisierten Fahrfunktionen und wird zum Testen und Absichern dieser Funktionen genutzt. Dadurch soll es einfacher sein, Sensormodelle zu integrieren, sowie virtuelle Tests durchzuführen [33]. Da automatisierte Fahrfunktionen immer komplexer werden und damit auch die Anforderungen an Test- und Entwicklungsmethoden stark zunehmen, gewinnt dies immer mehr an Relevanz. Durch Tests in virtuellen Umgebungen können bestimmte Bedingungen kontrolliert erstellt und identisch reproduziert werden, was vor allem viel Zeit und Geld spart, aber auch genauer ist und weniger Risiko für Menschen und die Umgebung darstellt.

OSI enthält eine objektbasierte Umgebungsbeschreibung, welche auf Google's Protocol Buffers, kurz *Protobuf*, basiert. Protobuf ist eine IDL (*Interface Definition Language*), zu Deutsch eine Schnittstellenbeschreibungssprache. Damit lassen sich Datentypen und Schnittstellen unabhängig von einer Programmiersprache oder einer Betriebssystemplattform beschreiben [1]. Ziel dabei ist es, eine Kommunikation zwischen verschiedenen Software-Komponenten, welche unterschiedliche Programmiersprachen verwenden, mithilfe einer definierten Schnittstelle sicherzustellen. Ausgehend von dieser Schnittstellendefinitionssprache kann ein spezieller Compiler - im Falle von Protobuf ist dies der *Protobuf Compiler* (protoc) - die Definitionen in Sourcecode verschiedener Sprachen, wie C++, C#, Python oder Java, umsetzen. Code 1 zeigt ein Beispiel der Protocol Buffers Syntax. Die „Attribute" der Nachricht wie *name* oder *age* werden bei Protobuf als *Felder* bezeichnet.

OSI definiert all seine Nachrichten zum Datenaustausch zwischen den Modellen genau in dieser Beschreibungssprache.

Im OSI können Daten für Verkehrsteilnehmer und Sensoren definiert werden. Abbildung 2.2 stellt die Schnittstellen und Modelle dar, welche für die Modellierung eines

```
message Person {
        optional string name = 1;
        optional int32 age = 2;
        optional string adress = 3;
}
```

Code 1: *Beispiel einer Protocol Buffer Nachrichtendefinition*

Sensors benötigt werden.

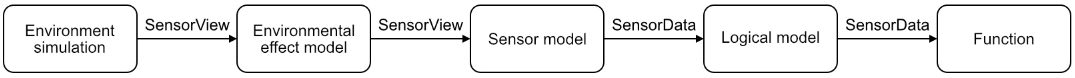

Abbildung 2.2: *Schnittstellen und Modelle eines Sensormodells in OSI [6]*

Zwei der wichtigsten Nachrichtenformate, *SensorView* und *SensorData*, sind bereits in der Abbildung 2.2 zu sehen. Daneben existieren noch *GroundTruth* und *FeatureData*. Eine GroundTruth-Nachricht beschreibt die simulierte Umgebung und Parameter aller simulierter Objekte, wie deren Position im globalen Koordinatensystem der Simulation. SensorView ist dabei die Input-Nachricht von Sensormodellen und von einer GroundTruth-Nachricht abgeleitet. Zusätzlich dazu sind Parameter in Bezug auf das physikalische Sensorkoordinatensystem enthalten, wie z.B. die genaue Position am Fahrzeug. Anschließend können die vom Sensormodell produzierten SensorData-Nachrichten z. B. für bestimmte automatisierte Fahrfunktionen oder für ein Sensorfusionsmodell weiterverwendet werden. Diese enthalten GroundTruth- sowie SensorView-Nachrichten und können zusätzlich noch FeatureData-Nachrichten enthalten, welche bestimmte Sensordaten mit Zusatzinformationen darstellen.

OSI kann in Kombination mit anderen öffentlichen Standards, wie OpenDRIVE und OpenSCENARIO, welche ebenfalls an ASAM übertragen worden sind, benutzt werden. OpenDRIVE beschreibt dabei die statischen Informationen einer Simulation, unter anderem Straßen, Kreuzungen und Gebäude. OpenSCENARIO bringt durch Fahrszenarien dynamisches Verhalten in die Simulation durch Beschreibungen des Verhaltens von z. B. Fahrzeugen oder Fußgängern. Diese können je nach Simulationszweck angepasst werden, wodurch dann das Verhalten der einzelnen Teilnehmer analysiert wird.

Wie bereits erwähnt, findet in der Automobilindustrie das Testen verschiedener Funktionen oder Fahrszenarien immer häufiger in Simulationen statt. Der Grund dafür ist, dass das Testen in diversen Umweltbedingungen und deren exakte Reproduktion einen zu komplexen und kostspieligen Vorgang darstellt und sich damit als schwer realisierbar erweist. Aus diesem Grund wird OSI für diesen Zweck angewendet.

Dafür wird in einem ersten Beispiel ein Co-Simulations-Framework beschrieben, welches OSI dahingehend erweitert, um Umgebungssimulationen mit Funktionssimulationen zu koppeln [39].

In einer weiteren Veröffentlichung wird ein neues generisches Sensormodell zum Testen von ADAS/AD (Advanced Driver Assistance Systems/Autonomous Driving) Funktionen vorgestellt. Dieses wandelt eine eingehende Objektliste in eine sensorspezifische Objektliste um und ist für alle Sensoren geeignet, welche auf Objektebene arbeiten. Das Sensormodell wird dann mithilfe von Vires VTD und OSI in einen virtuellen Prüfstand integriert [41].

Ein weiteres Beispiel ist das SET-Level-Projekt. Es zielt darauf ab, eine Umgebung für simulationsbasiertes Testen und Entwickeln automatisierter Fahrfunktionen bereitzustellen. Dabei ist das Hauptziel, eine offene, flexible und erweiterbare Simulationsumgebung zu schaffen, welche aktuellen Simulationsstandards, wie dem Functional-Mock-Up-Interface (FMI) oder OSI entsprechen. Hierfür ist eine Fahrzeugsimulationskette vorgeschlagen worden, welche die Modelle der Bewegungssteuerung (Aktuatoren inklusive Aktuatormanagement) und der Fahrdynamik mit zwei unterschiedlichen Detailstufen umfasst [7].

OSI ist ursprünglich, wie bereits erklärt, für die Benutzung einer Simulation entwickelt worden. Dies ist für diese Arbeit nicht weiter relevant, da das Datenformat auch unabhängig davon benutzt werden kann.

Um diese Datenstrukturen in einer Fahrzeugarchitektur zu verteilen, kann das ROS framework verwendet werden. Daher wird im folgenden Kapitel auf dessen Funktionsweise eingegangen.

2.4 Robot Operating System

Das Robot Operating System, kurz ROS, ist ein quelloffenes Metabetriebssystem für Roboter [24, S. 326]. Seine Ursprünge hat ROS im Jahr 2005. Zwei Studenten haben an der Stanford Universität für deren Forschung eine Roboterplattform benötigt. Plattformen dieser Art hat es zu dieser Zeit bereits gegeben, jedoch ist dabei die Software spezifisch für die zugrundeliegende Hardware geschrieben worden. Folglich ist es ein großer Aufwand gewesen, diese für einen anderen Zweck oder an einem andersartigen Roboter mit geänderter Hardware zu nutzen. Um dieses Problem zu lösen, haben die Studenten einen neuen, gemeinsamen Software-Stack und einen vielseitigen, physischen Roboter (*PR*) entwickelt. In einem Forschungslabor namens *Willow Garage*, welches auf autonome Fahrzeuge spezialisiert war, ist weiter am Software-Stack und dem PR geforscht worden [42]. Die Software-Implementierung hat 2007 dann erstmals offiziell den Namen *Robot Operating System*, kurz ROS, bekommen. 2009 ist bereits die erste ROS-Version veröffentlicht worden. 2012 hat das Open Source Robotics Foundation (OSRF) dann das ROS-Projekt übernommen und ein Jahr später bereits die Version *Hydro Medusa* veröffentlicht [28, S. 134].

Um die Funktionsweise von ROS zu verstehen, werden nun die Design-Prinzipien sowie die verwendete Nomenklatur erklärt.

Prozesse eines Roboterkontrollprogramms können potentiell auf mehrere Rechner verteilt werden. Das wird beispielsweise dann benötigt, wenn ein Roboter Sensordaten erfasst und diese dann ohne weitere Verarbeitung an einen Computer verschickt, welcher die Informationen dann weiterverarbeitet. Das Verschicken der Daten vom Roboter sowie der Empfang am Computer sind bei ROS im Allgemeinen in einer Peer-to-Peer-Topologie verbunden und benötigen keinen zentralen Server.

ROS kann in verschiedenen Programmiersprachen genutzt werden. Anfangs sind nur C++, Python, Octave und Lisp unterstützt worden, nachfolgend kamen noch Weitere hinzu. Zudem sind in ROS einige Werkzeuge enthalten, wie `rxgraph` zum Visualisieren der Netztopologie, `rosbag` zum Aufzeichnen der Daten oder auch `rosmake` zum Übersetzen und Linken der Software. ROS ist frei verfügbar und quelloffen (open-source) und unterscheidet sich daher von proprietären Entwicklungen, wie dem *Robotics Studio* von Microsoft [24, S. 328].

Die wichtigsten Begriffe und Konzepte in ROS sind Nodes, Messages, Topics, Services und der ROS master. Nodes, zu Deutsch Knoten, sind Daten verarbeitende Prozesse. Diese werden auch Software-Module genannt. Messages (Nachrichten) werden von Nodes ausgetauscht, wenn diese miteinander kommunizieren. Sie bestehen aus primitiven Datentypen, wie zum Beispiel int und bool, bzw. Arrays (Felder) dieser Typen. Messages können wiederum Messages oder auch Felder von anderen Nachrichten enthalten und damit beliebig komplex verschachtelt und an einen bestimmten Zweck angepasst werden. Topics (Themen) stellen einen einfachen String, wie „Radardaten“, dar [24, S. 329].

Somit kann es mehrere Nachrichten zum Austausch von Sensordaten zwischen zwei Knoten mit demselben Format geben, welche dann durch ihren Namen (topic) unterschieden werden können. Der Name eines topics darf dabei nur einmal in einem ROS-System vorkommen.

Im Gegensatz zu Topics, welche meist regelmäßig ausgetauscht werden, sind einzelne *Services* synchrone Transaktionen [24, S. 329], um eine bestimmte Aktivität einmalig durchzuführen. Der ROS master, oder auch *roscore*, ist eine Sammlung von Knoten und Programmen, welcher in ROS benötigt wird, um initial beim Starten eines Netzwerks die Kommunikation zwischen den verschiedenen Nodes herzustellen [42].

Dies stellt bereits den ersten Nachteil von ROS dar. Ohne den ROS-core am Anfang zu starten, ist eine Kommunikation unter den Knoten nicht möglich. Außerdem sind die ROS-Datentransportprotokolle UDPROS und TCPROS nicht echtzeitfähig [54], wie auch ihre Basis-Protokolle TCP und UDP. Dies wird jedoch vermehrt in Roboterplattformen benötigt. Des Weiteren setzt ROS auf Linux auf und kann daher unter Windows, MacOS, RTOS und weiteren Systemen nicht oder nur eingeschränkt verwendet werden [36].

Trotz des Versuchs, diese Probleme zu lösen, war es schwierig, die Gesamtleistung von ROS zu steigern [36], was jedoch heutzutage wichtig ist, da ROS vermehrt in Landfahrzeugen, Luftfahrzeugen und mobilen Robotern jeder Größe eingesetzt wird [48]. Dies führte zur Entwicklung von ROS2 mit einer verbesserten und moderneren Architektur als ROS Version 1. ROS2 benutzt das DDS Protokoll (Data Distribution Service) und führt weitere Designkonzepte, wie Publisher und Subscriber mit ein. Auch wird durch das DDS Protokoll der erwähnte ROS-core nicht mehr benötigt [36].

Diese Verbesserungen im Vergleich zu seinem Vorgänger machen ROS2 zukunftssicherer und kompatibler mit Erneuerungen. Deshalb wird bei aktuellen Projekten vermehrt auf den ROS2-Standard gesetzt.

Da sowohl ROS als auch ROS2 in den verschiedensten Bereichen, angefangen bei Roboterplattformen bis zu autonom fahrenden Fahrzeugen, Verwendung findet, wird damit in viele verschiedene Richtungen geforscht. Zwei der vier folgenden Veröffentlichungen benutzen ROS Version 1. Da beide Modelle, bzw. Algorithmen, jedoch genauso auch in ROS2 umgesetzt werden können und die vorher genannten Vorteile mit sich bringen, werden diese hier mit aufgenommen.

Zur Erfassung der Umgebung eines Fahrzeug wird ein zuverlässiges und robustes System benötigt, um bei jedem Wetter und zu jeder Tageszeit diese gut erkennen zu können. Dafür ist bereits 2019 ein Modell zur Zusammenführung von Kamera- und Radardaten vorgeschlagen worden, welches mithilfe des Robot Operating Systems implementiert wird. Diese Kombination der Sensorfusion bringt die größten Vorteile bezüglich der Erkennung von Objekten und deren Parameter mit sich, was auch in dem Artikel dargestellt wird [34].

ROS2 wird häufig eingesetzt, um dynamisch Pfade zu planen. Dies ist bei Industrieumgebungen wichtig, da sie dort autonom auf statische sowie dynamische Hindernisse reagieren müssen. Da eine solche dynamische Hindernisbehandlung im ROS2-Navigations-Stack (Nav2) nicht vorhanden ist, ist Nav2 2022 durch einen dynamischen Hindernislayer als Plug-and-Play-Lösung erweitert worden [14].

Ebenfalls ist die gleichzeitige Positionsbestimmung und Kartierung (Simultaneous Localization and Mapping, kurz SLAM) eine wichtige Technologie zur autonomen Navigation intelligenter Fahrzeuge. Dabei sind in einem 2022 veröffentlichen Konferenzpapier drei verschiedene Kartierungsalgorithmen vergleichen worden. Außerdem wurde auf Basis mehrerer Algorithmen eine globale und lokale Pfadplanung durchgeführt, um autonome Navigations- und Hindernisvermeidungsfunktionen zu realisieren. Dies ist auch anhand eines Fahrzeugs verifiziert worden, wobei eine genaue Posenschätzung sowie Kartenkonstruktion einer unbekannten Umgebung erfolgt und anschließend eine autonome Navigation darin durchführbar gewesen ist [37].

Da ROS2 für die Entwicklung autonomer Systeme benutzt wird, werden auch Zeitanforderungen immer wichtiger. Ende-zu-Ende Timing-Garantien sind für ein sicheres und vorhersehbares Verhalten in jeder Situation erforderlich. ROS2 bietet solche Echtzeitanforderungen und ein zuverlässiges Timingverhalten mithilfe eines Scheduler-Designs, welches die Ausführung aller Komponenten verwaltet. Dadurch können dann Parameter, wie die maximale Reaktionszeit und maximales Datenalter, bestimmt werden [55].

3 Interfacekonzept und Datenzuordnung

Nachdem die Grundlagen zu den verschiedenen Themen, sowie der aktuelle Stand der Forschung dargelegt wurde, folgt nun eine theoretische Datenzuordnung zwischen den Radardaten und den Nachrichten von OSI. Dabei werden vorerst die Daten beschrieben, welche das Radar bereitstellt und in welcher Frequenz diese versendet werden. Anschließend folgen die Nachrichten aus OSI. Am Ende dieses Kapitels werden dann die Daten des Radars den Nachrichten von OSI zugeordnet. Dabei wird genauer darauf eingegangen, warum und welche Informationen hierbei relevant sind.

3.1 Daten des Radars

Zuerst soll es um die Daten des SR73 Radars gehen. Alle verfügbaren Nachrichten, Signale und Informationen, welche das Radar liefert, sind detailliert im *Communication Protocol* dargestellt. Dieses ist als Quelle unter [52] aufgeführt. Einige Informationen werden einer aktuelleren Version des Communication Protocols [53] entnommen, welche jedoch nicht öffentlich verfügbar ist. Die Unterschiede zwischen den Versionen werden in dieser Arbeit beschrieben und das aktuellere Datenblatt liegt der Arbeit als Anhang bei.

3.1.1 Konfigurations- und Status-Nachrichten des Radars

Das Radar kann durch eine Konfigurations-Nachricht parametrisiert werden. Dabei können verschiedene Funktionen des Radars aktiviert oder auch deaktiviert werden. Für beides ist es wichtig, das zugehörige Gültigkeits-Bit (*Valid-Bit*) auf eins zu setzen, damit die Änderung auch tatsächlich vom Radar übernommen wird. Die wichtigsten Parameter, die man bei diesem Radar setzen kann, sind die Folgenden: *MaxDistance* setzt die maximale Distanz des Radars, in welcher Objekte erkannt werden. Wenn *StoreNVM* und das zugehörige *Valid-Bit* jeweils auf eins gesetzt werden, wird die komplette Konfiguration in den NVM-Speicher abgelegt und ist beim nächsten Betrieb dann die initale Konfiguration des Radars. Mithilfe des *SortIndex* kann das Radar entweder die Objekte nach Entfernung oder deren RCS-Werten sortieren. *SensorID* setzt die ID eines Sensors. Diese ist initial null und kann Werte zwischen null und sieben annehmen. Durch das Setzen dieser SensorID ändern sich auch die IDs aller CAN-Nachrichten, welches ein Radar sendet oder empfängt. Alle CAN-Nachrichten-IDs werden dann wie folgt berechnet:

$$MsgID = MsgID_0 + SensorID * 0x10 \tag{3.1}$$

Wenn also einem Radar mit Sensor die *RadarCfg*-Message mit der ID *0x200* geschickt wird und dessen SensorID auf vier gesetzt wird, kann dieses anschließend unter der CAN-Nachrichten-ID *0x240* weiter parametrisiert werden. Eine Statusnachricht, welche bei einem Radar mit der SensorID null die ID *0x201* besitzt, würde bei solch einer Konfiguration dann folglich unter der CAN-ID *0x241* von diesem gesendet werden.
Wie bereits angesprochen sendet jedes Radar des Typs SR73 eine Status-Nachricht. Die CAN-ID bei SensorID null ist hierbei *0x201*. In dieser sind die in der Konfigurations-Nachricht gesetzten Einstellungsmöglichkeiten zu sehen, wie die *Distanz*, *SensorID* oder auch der *SortIndex*. Außerdem zeigen zwei Bits noch den *MotionRxState* an, ob das Radar gerade eine aktuelle Geschwindigkeit und Gierrate des Fahrzeugs über den CAN erhält. Diese Status-Nachricht wird sekündlich mit einer weiteren Nachricht, der *Version-Information*, gesendet. Diese enthält drei Bytes über die Softwareversion des Radars, welche in Major Version, Minor Version und Patch Level unterschieden wird.
In einer aktuelleren Version des Radar-Datenblattes ist es ebenfalls noch möglich, eine Kollisionserkennung in einem rechteckigen Bereich vor dem Radar zu aktivieren. Dafür muss zuerst zum Aktivieren der Funktion eine spezielle Konfigurationsnachricht zum Radar gesendet werden, worin die Koordinaten der Eckpunkte des Rechtecks enthalten sind. Anschließend sendet es dann Informationen über Objekte, wenn sich diese im definierten Bereich für die Kollisionserkennung befinden. Da die Funktion jedoch für diese Arbeit nicht weiter relevant ist, wird darauf nicht näher eingegangen.

3.1.2 Objektinformationen

Zwischen den sekündlich gesendeten Status-Nachrichten schickt das Radar Informationen über die erkannten Objekte. Diese sind in zwei Teile aufgeteilt.
Eine erste Nachricht mit der ID *0x60A* (bei SensorID gleich null) liefert generelle Informationen über die Anzahl der erkannten Objekte, den aktuellen Zähler der Objektmessungen sowie die CAN-Interface-Version.
Der zweite Teil besteht dann aus CAN-Nachrichten mit der ID *0x60B* für die Objektinformationen, wobei jeweils eine Nachricht für jedes erkannte Objekt gesendet wird. Diese enthalten dann die vom Radar vergebene ID des erkannten Objektes, welches vom Radar verfolgt wird, deren longitudinale und laterale Entfernung und relative Geschwindigkeit, den RCS-Wert sowie genauere Informationen über das jeweilige Objekt, wie, ob es sich bewegt, steht oder ob ein anderer Status dem Objekt zugeordnet wurde.

3.1.3 Benötigte Daten für den Radarbetrieb

Während des Radarbetriebes werden laut dem Datenblatt zwei Nachrichten regelmäßig erwartet. Zum Einen ist das die Information über die aktuelle Geschwindigkeit des Fahrzeugs, sowie die Richtung, also ob es vorwärts oder rückwärts fährt oder gerade steht. Zum Anderen ist es die Gierrate des Fahrzeugs. Die ID's der beiden Nachrichten sind die *0x300* (Geschwindigkeit) und *0x301* (Gierrate). Das Radar würde auch ohne diese Informationen funktionieren und dann mit den Standardwerten von 0 $\frac{m}{s}$ und 1 $\frac{deg}{s}$ arbeiten. Ob das Radar die beiden Informationen erhält, ist in dem *RadarState_Motion_Rx_State*-Signal aus der Radar-Status-Nachricht mit der ID *0x201* ersichtlich.

Da jetzt alle für diese Arbeit relevanten Daten des Radars beschrieben wurden, folgen nun die Daten und Nachrichten, welche OSI bereitstellt beziehungsweise benötigt.

3.2 Verfügbare Daten und Nachrichten des Open Simulation Interfaces

Alle Daten aus OSI, welche relevant für diese Arbeit sind, werden im Folgenden aufgezählt. Dabei wird zuerst genauer auf die RadarDetectionData-Nachricht eingegangen, welche die Objektinformationen beinhaltet und anschließend wird noch die HostVehicleData-Nachricht dargelegt, welche alle Informationen über das Fahrzeug und dessen Fahrparameter beinhaltet.

3.2.1 RadarDetectionData

Wie bereits zuvor erwähnt, ist die Vorgabe, dass für die Daten des Short-Range Radars die OSI-Nachricht RadarDetectionData verwendet werden soll. Da diese ein Teil der FeatureData-Nachricht ist, welche Daten auf niedrigem Level, wie Sensordaten ohne Berücksichtigung von Informationen anderer Sensoren, darstellt, ist diese für diesen Zweck passend. Alternative Nachrichtentypen, wie *DetectedMovingObject* oder *DetectedStationaryObject*, wurden meinerseits kurzfristig in Erwägung gezogen, jedoch boten diese keine Mehrinformation und es hätten auch zusätzliche, nicht relevante Daten gesendet werden müssen. Daher blieb es bei dem Nachrichtentyp *RadarDetectionData* zur Übertragung der Radardaten. Code 2 zeigt die OSI-Definition für diese Nachricht.
Dieser enthält einen Header, den SensorDetectionHeader, mit Informationen über den Zeitstempel der Messung, die Anzahl der gültigen Messungen, Kennzeichner über die Verfügbarkeit der Radardaten, die Lage des Sensors am Fahrzeug sowie die Sensor-ID und einen Messzähler.
Außerdem besteht sie aus einer Folge von Objekterkennungen (*RadarDetection's*). Diese enthalten Informationen über die Objekt-ID, eine Klassifikation, Position sowie das

```
message RadarDetectionData
{
    // Header attributes of radar detection from one radar sensor.
    //
    optional SensorDetectionHeader header = 1;

    // List of radar detections constituting the radar detection list.
    //
    repeated RadarDetection detection = 2;
}
```

Code 2: *Definition der OSI-Nachricht RadarDetectionData [43]*

mittlere Abweichungsquadrat davon (root mean square error, kurz rmse). Außerdem sind darin noch die Radialgeschwindigkeit mit dem dazugehörigen rmse-Wert enthalten, das Signal-Rausch-Verhältnis (signal noise ratio, kurz snr), der RCS-Wert, die Existenzwahrscheinlichkeit des Objektes sowie eine ID, wenn die Messung zweideutig war. Abbildung 3.1 zeigt detailliert den Aufbau der Nachrichten und die dazugehörigen Datentypen.

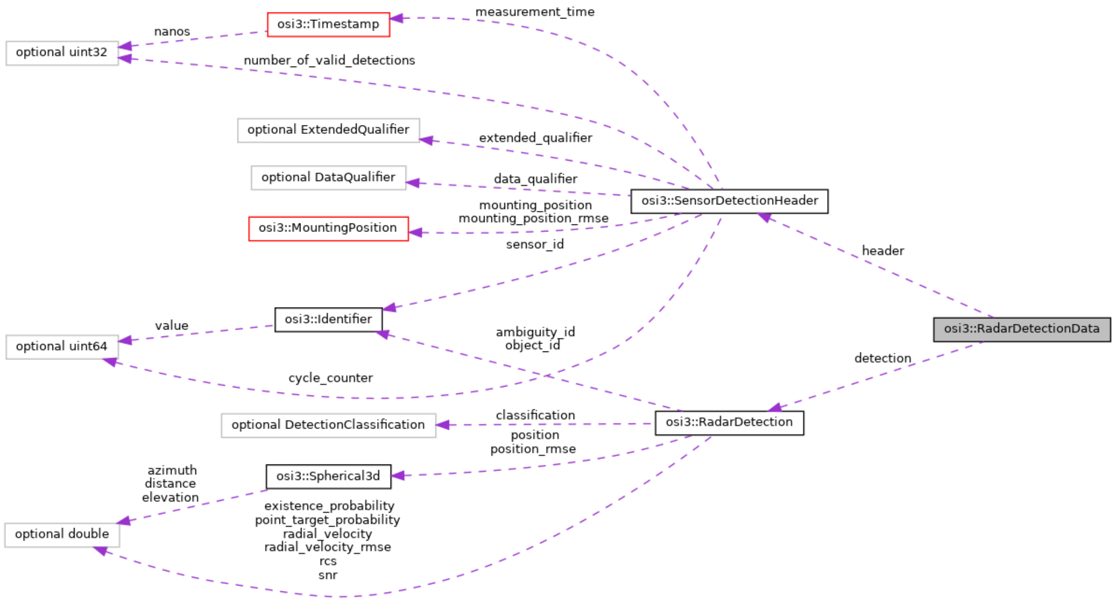

Abbildung 3.1: *Übersicht über die RadarDetectionData-Nachricht [12]*

3.2.2 Benötigte Daten aus HostVehicleData

HostVehicleData ist eine Nachricht, welche den aktuellen Zustand eines Fahrzeugs darstellt. Darin enthalten ist eine BaseMoving-Nachricht, welche Informationen über die Orientierung inklusive Gierrate und die Geschwindigkeit enthält. Zukünftig wird es für diese beiden Fahrinformationen eine eigene OSI-Nachricht geben. Diese ist von *Maikol Drechsler* bereits in einem pull-request vorgemerkt und beim aktuellen Fortschritt der Arbeit noch nicht in den Master-Branch mit aufgenommen worden, jedoch ist sie bereits

mit dem *ReadyToMerge*-Label versehen [38].
Sie ist eine Nachricht in der HostVehicleData-Nachricht und wird *VehicleMotion* genannt. Darin werden zukünftig die genauesten Informationen zum Zustand des Fahrzeugs enthalten sein, welche über das On-Board-Netzwerk verfügbar sind. Diese beinhalten beispielsweise Position, Geschwindigkeit, Beschleunigung und Orientierung des Fahrzeugs.

3.3 Zuordnung der Radardaten zu Nachrichten des Open Simulation Interfaces

Nachdem nun die Daten, Signale sowie Nachrichten, beschrieben wurden, welche das Radar und welche OSI zur Verfügung stellt und welche Informationen jeweils enthalten sind, folgt nun die genaue Zuordnung der einzelnen Informationen zueinander.

3.3.1 Zuordnung der Objektdaten

Die generellen Objektinformationen und die Nachrichten über die einzelnen Objekterkennungen können großteils den SensorDetectionHeader und den RadarDetection's zugeordnet werden. Tabelle 3.1 stellt diese eindeutige Datenzuordnung dar.

CAN-ID	CAN-Signal	OSI-Feld	OSI-Nachricht
0x201	RadarState_SensorID	sensor_id	SensorDetectionHeader
0x60A	Objects_NofObjects	number_of_valid_detections	
	Objects_MeasCount	cycle_counter	
0x60B	Objects_ID	object_id	RadarDetection
	Objects_DistLong	position	
	Objects_DistLat		
	Objects_VrelLong	radial_velocity	
	Objects_VrelLat		
	Objects_RCS	rcs	

Tabelle 3.1: *Eindeutige Datenzuordnung der Objektdaten*

In dieser und den folgenden Tabellen werden als OSI-Felder entweder die Feldnamen oder die Namen von Subnachrichten einer Nachricht bezeichnet, da ein Feld auch eine oder mehrere Subnachrichten darstellen kann.
Beim Radar bleiben somit die *Objects_InterfaceVersion* in der *Object List Status* über, welches die Angabe des CAN-Interfaces darstellt. Diese Information kann in OSI nicht abgebildet werden. Gleiches gilt für die *Objects_DynProp* in *Objects General Information*, welche den Zustand des erkannten Objektes beschreiben. Da die OSI *classification* hier nur zwischen unbekannt, ungültig, clutter, über- und unterfahrbar oder anderem (unspezifiziert aber bekannt) unterscheidet, wird allen erkannten Objekten die zuletzt genannte Eigenschaft *Anders* (*UNKNOWN*) zugewiesen.

In der *SensorDetecionHeader*-Nachricht von OSI können somit nicht alle Felder befüllt werden. Eine *messurement_time*, also eine Zeitmessung oder Zeitstempel der aktuellen Messung, bietet das Radar nicht an. Ebenfalls stellt es keine Klassifikatoren bereit, ob Daten verfügbar sind oder nicht und den Grund dafür. Hier ist das einzige Erkennungsmerkmal das *MotionRxState*-Signal aus der *Radar-Statue*-Nachricht mit der ID *0x201*. Dieses zeigt an, ob das Radar aktuell Geschwindigkeit und Gierrate über den CAN-Bus erhält. Darüber hinaus kann keine Aussage über die Verfügbarkeit und Korrektheit der Radardaten getroffen werden. Ebenfalls kann das SR73 keine Montageposition (*mounting position*) speichern oder ausgeben. Die Sensor-ID wird indirekt durch den Radarstatus und der CAN-Nachrichten-IDs erfasst, was auch in Tabelle 3.1 in Zeile eins dargestellt wird. Diese Information wird nicht zusätzlich in den Objektdaten mitgeschickt.
Bei einer RadarDetection können den Feldern *ambiguity_id*, *existance_probability*, *point_target_probability* und *snr* keine Signale des Radars zugeordnet werden. Ebenfalls können keine Werte für die quadratische Abweichung (rmse), also *mounting_position_rmse* aus dem SensorDetectionHeader und *position_rmse* und *radial_velocity_rmse* aus der RadarDetection angegeben werden.

3.3.2 Zuordnung der Geschwindigkeit und der Gierrate

Diese beiden Informationen können direkt aus der vorher genannten OSI-Nachricht *HostVehicleData* extrahiert werden. Eine aktuell mögliche Datenzuordnung der Informationen zeigt Tabelle 3.2. Eine angestrebte, zukünftige Zuordnung zeigt Tabelle 3.3. Die Richtung der Geschwindigkeit, welche in der Radarnachricht mit der ID *0x300* noch enthalten ist, kann indirekt aus den Informationen aus OSI, wie dem Vorzeichen der Geschwindigkeit, ausgelesen werden. Ein direktes Äquivalent zur Anzeige der Fahrtrichtung gibt es in OSI noch nicht.

CAN-ID	CAN-Signal	OSI-Feld	OSI-Nachricht
0x300	RadarDevice_Speed	velocity	BaseMoving in HostVehicleData
0x301	RadarDevice_YawRate	orientation_rate	

Tabelle 3.2: *Aktuelle Datenzuordnung der Geschwindigkeit und Gierrate*

CAN-ID	CAN-Signal	OSI-Feld	OSI-Nachricht
0x300	RadarDevice_Speed	velocity	VehicleKinematics in HostVehicleData
0x301	RadarDevice_YawRate	orientation_rate	

Tabelle 3.3: *Zukünftige Datenzuordnung der Geschwindigkeit und Gierrate*

Weitere Informationen aus HostVehicleData werden vom Radar nicht benötigt. Daher ist diese Zuordnung damit abgeschlossen.

3.3.3 Abbildung der Konfigurationen und Statusmeldungen

Zuletzt müssen noch die Daten der Konfigurationen und die Status-Nachrichten des Radars auf OSI abgebildet werden. Da nur einzelne Daten der Statusnachricht, wie die Sensor-ID, und auch nur wenige Signale der Konfigurationsnachricht in einer OSI-Nachricht dargestellt werden können, ist eine Abbildung derer durch OSI-Nachrichten vorerst nicht möglich.

Nachdem alle Daten des Radars jetzt größtenteils zugeordnet sind, ist es nicht notwendig, für diese Arbeit eine neue OSI-Nachricht zu entwerfen bzw. die RadarDetectionData zu erweitern.

Da die Datenzuordnung gezeigt wurde, folgt nun die praktische Umsetzung dieser. Dafür wird zuerst das Vorgehen erläutert und anschließend die einzelnen Schritte detailliert erklärt.

4 Anwendung des Interfacekonzeptes

4.1 Vorüberlegungen, Vorgaben und Vorgehen

Nachdem die Datenzuordnung abgeschlossen ist, muss diese nun praktisch auf die tatsächlichen Daten angewendet werden. Um dies zu ermöglichen, wurden im Vorhinein bereits einige Vorgaben bezüglich der Umsetzung vom Forschungsinstitut vorgegeben. Zum Einen wurde als Programmiersprache C++ gewählt, da damit sehr einfach ROS2-Knoten erstellt und programmiert werden können. Außerdem bietet es gegenüber Python, welches ebenfalls häufig zur Programmierung von ROS-Knoten verwendet wird, den Vorteil, dass es direkt in Maschinencode kompiliert wird und daher generell schneller und effizienter ist, als Python, welches von einem Interpreter ausgeführt wird. Anschließend daran wurde nur noch ROS2 zum Übertragen der Nachrichten und OSI als Datenformat festgelegt, der Rest wurde vorerst unspezifiziert gelassen.

Mit diesen Vorgaben wird dann begonnen. Zuerst wird mit verschiedenen CAN-Tools gearbeitet, um sich auf die Kommunikation mit dem Radar vorzubereiten, da dies nicht von Beginn an zur Verfügung stand. Anschließend wird ein CAN-Description-File geschrieben, um den Bytes der CAN-Nachrichten tatsächliche physikalische Werte zuordnen zu können. Daraufhin beginnt die Arbeit mit ROS2, vorerst zum Einarbeiten in ROS allgemein und dann genauer in Themen wie Knoten, Executer, Topics und Nachrichten. Anschließend wird OSI in ein bestehendes ROS-Paket integriert und damit dann eine Konverter-Klasse geschrieben, welche die Umwandlungen von CAN nach OSI vornimmt. Diese müssen dann mittels ROS2 versendet werden. Dabei wird ebenfalls eine CAN-Klasse programmiert, welche die Kommunikation in C++ mit dem CAN-Bus mittels SocketCAN [32] ermöglicht.

Alle genannten Schritte werden in den folgenden Kapiteln nochmals genauer ausgeführt, die darin aufgetretenen Probleme und Hindernisse aufgezeigt und die daraus entstandenen Lösungen erläutert.

4.2 Cantools und CAN-Utilities

Mithilfe der SocketCAN userspace utilities and tools [50] können viele Funktionen eines CAN-Busses auf Linux getestet und erprobt werden. Das wurde für diese Arbeit vor allem am Anfang verwendet, als das SR73 Short-Range Radar für mich noch nicht verfügbar war. Die CAN-Utilities ermöglichen es mittels SocketCAN [32], einer Sammlung von

CAN-Treibern und einer Netzwerkschicht, durch Verwendung einfacher Befehle, das Senden und Empfangen von CAN-Nachrichten auf einem virtuellen CAN-Bus, beispielsweise dem *vcan0*. Die wichtigsten Befehle sind dabei *cansend* zum Schicken einzelner Nachrichten und *candump* zum Darstellen, Filtern und Aufzeichnen von CAN-Nachrichten. Die CAN BUS tools (cantools) [10] bieten weitere sehr nützliche Funktionen, welche zum Entwickeln von CAN-Applikationen verwendet werden können. Mittels *decode* dekodiert man eintreffende Nachrichten mit Hilfe eines CAN-Description-Files (CAN-Database) und sieht anschließend statt der Bytes sofort die beschriebenen physikalischen Werte. Der Befehl *dump* dient der graphischen Darstellung von Bitpositionen, welche in einem CAN-Description-File beschrieben werden. Der für diese Arbeit relevanteste Subbefehl jedoch ist *generate_c_source*. Dieser generiert aus einer CAN-Beschreibungsdatei C-Code, welcher structs, `pack()`- und `unpack()`-Funktionen zum Befüllen und Auslesen von CAN-Nachrichten bereitstellt. Außerdem werden `encode()`- und `decode()`-Funktionen zum Umwandeln verschiedener Signalwerte, beispielsweise bei der Geschwindigkeit mit Offset und Auflösung (Skalierung der Werte auf dem Bus), sowie Definitionen von ID, Längen und Typ der beschriebenen Nachrichten definiert. Dies ist ein großer Vorteil, da die Implementierung, wie bereits erwähnt, komplett in C++ geschrieben werden soll und daher problemlos genutzt werden kann.

4.3 CAN-Description-File (CAN-Datenbank)

Ein CAN-Description-File stellt eine einfache Textdatei dar, welche Informationen über den Bus, Nachrichten und deren Signale enthält. Damit können dann die am Bus ankommenden Nachrichten dekodiert und aus den Bytes physikalische Daten gewonnen werden. Ein bekanntes Format für solch ein CAN-Data-Description-Format ist DBC. Es ist von der Firma *Vector Informatik GmbH* eingeführt worden, welche Softwarewerkzeuge und -Komponenten sowie Hardware für die Entwicklung, das Testen und das Analysieren von elektronischen und eingebetteten Systemen, sowie deren Vernetzung, entwickelt [22].
Da dieses Format jedoch proprietär ist, viele Dateien und CAN-Datenbanken nicht öffentlich und kostenlos zur Verfügung stehen und meist schlecht oder nicht dokumentiert sind, hat das open-source CAN-Analysis-Tool Kayak [30] ein neues Format eingeführt, welches gut dokumentiert ist und auf XML basiert. Dieses ist das KCD-Format mit der Dateiendung .kcd, was die Kurzform für „Kayak CAN definition“ ist [31].
Es bietet die gleiche Funktionalität und Beschreibungsumfang wie eine .dbc Beschreibungsdatei. Durch das auf Java basierte Konvertierungsprogramm *CANBabel* lassen sich CAN-Datebases des DBC-Formats in ein KCD-Dateiformat umwandeln. Dieses und weitere Programme ermöglichen das Konvertieren der Formate untereinander, womit es vorerst zweitrangig ist, welches der beiden Beschreibungsformate verwendet wird. Für meinen Verwendungszweck, für das Auslesen der physikalischen Werte, wurde daher das KCD-Format verwendet, da es durch die XML-Syntax besser strukturiert und leichter

zum Lesen und Schreiben ist, als das DBC-Format.
Nachdem KCD als Beschreibungsformat feststand, wird nun eine CAN-Datenbank für das SR73 Radar erstellt. Alle Informationen über die ankommenden und ausgehenden CAN-Nachrichten, sowie deren IDs werden dabei aus dem offiziellen Communication-Protocol-Datenblatt des Radars von der Website des Radarherstellers entnommen. Dieses ist als Quelle im Literaturverzeichnis unter [52] aufgeführt.
Nachdem diese Datei fertiggestellt war, muss verifiziert werden, dass die Bitpositionen aus dem Datenblatt mit denen im .kcd-File übereinstimmen, damit die Daten später richtig dekodiert werden können. Für diesen Zweck und auch zum allgemeinen Testen von verschiedenen CAN-Funktionen werden die bereits genannten CAN BUS tools (cantools) verwendet. Dadurch, dass diese unter der MIT Lizenz stehen und einfach mittels „`python3 -m pip install cantools`" installiert werden können, werden diese zum Experimentieren mit dem CAN-Bus benutzt.

An dieser Stelle ist es wichtig zu nennen, dass das zugrundeliegende Betriebssystem, auf dem die komplette Software für das Radar entwickelt und getestet wird, die auf Debian basierte Linux-Distribution Ubuntu ist. Genauer noch wird die LTS (long term support) Version 22.04.1 verwendet. Das erleichtert den Software-Entwicklungsprozess deutlich, da es hier bzw. generell auf Linux sehr einfach möglich ist, virtuelle CAN-Netzwerke zu erstellen. Damit können, auch ohne physikalische CAN-Hardware, Funktionen und Programme zum Senden und Empfangen von CAN-Nachrichten getestet werden. Außerdem kann darauf die ROS2-Version *Humble Hawksbill* [47] problemlos installiert werden, welche speziell für diese Ubuntu-Version empfohlen wird.

Um schließlich verifizieren zu können, ob die im Description-File angegebenen Bitpositionen denen des Communication Protocols entsprechen, wird in den cantools der Subbefehl *dump* auf die Datei angewendet, um ein visuelles, besser lesbares Format zu erhalten. Damit ist der Verifizierungsschritt sehr einfach, da daran die einzelnen Bitpositionen, inklusive der Signale, anschaulich dargestellt werden und bei Kollisionen oder Überschneidungen von Signalen ein Hinweis gegeben wird.

4.4 Erste Inbetriebnahme des Radars

Mithilfe des CAN-Description-Files kann dann zum ersten Mal das Radar und dessen Daten analysiert und ausgelesen werden. Das Radar wird dafür mithilfe eines PCAN-USB-Adapters an den Computer angeschlossen. Zum Einrichten des SocketCAN-Interfaces muss die Bus-Bitrate angegeben und das Interface schließlich aktiviert werden. Dann ist es mittels des Befehls `candump can0` möglich, die vom Radar gesendeten Daten in der Konsole anzeigen zu lassen. Hierbei können dann die sekündlich geschickten Versions- und Konfigurationsnachrichten sowie die Nachrichten zu den Objekten anhand derer

IDs zugeordnet werden. Zum ersten praktischen Test der Beschreibungsdatei für die CAN-Signale wird daraufhin ein *log* mittels `candump can0 -l` erstellt und anschließend werden mit dem Befehl „`cat candump.log | cantools decode sr73.kcd`" die tatsächlichen physikalischen Informationen angezeigt. Dabei ist „sr73.kcd“ der Dateiname des CAN-Description-Files. Daraufhin wird die Möglichkeit des Radars getestet, sich durch eine Konfigurationsnachricht parametrieren zu lassen. Dafür wird eine Nachricht über den CAN mit der ID *0x200* zum Radar gesendet. Darin werden dann einzelne Bits, wie das Senden von weiteren Informationen, gesetzt und können in der nächsten vom Radar gesendeten Status-Message (ID *0x201*) als erfolgreich gesetzt gesehen werden. Anschließend wird der Cluster-Modus des Radars getestet. Dabei fällt jedoch auf, dass die ankommenden Radardaten nicht mit den Informationen aus dem Datenblatt übereinstimmen. Nach erneutem Überprüfen der CAN-Datenbank und des Datenblattes kann kein Fehler in der Implementierung gefunden werden. Daher wird die Firma *Nanoradar* kontaktiert, um die Aktualität des Datenblattes und der Software des Radars zu bestätigen. Nach mehreren Nachrichten mit einer Angestellten der Firma stellt sich heraus, dass das Radar den Cluster-Modus gar nicht unterstützt. Außerdem sind neue Funktionen hinzugekommen und es haben sich vereinzelt Bitpositonen verändert. Dies lässt sich aus einem neueren Datenblatt [53] entnehmen. Nachdem dieses jedoch bei manchen Nachrichten und deren Signale widersprüchliche Informationen liefert, wird schlussendlich das aktuellste Datenblatt der Firma auf chinesisch [51] bereitgestellt, mit welchem dann das CAN-Description-File nochmals überarbeitet wird. Dabei werden die Beschreibungen der Cluster vorerst nicht entfernt und die der Kollisionserkennung in die CAN-Database eingefügt, trotz der Tatsache, dass diese zunächst nicht benötigt werden.

4.5 Erstes Arbeiten mit ROS2

Nachdem die Kommunikation mit dem Radar nun sichergestellt ist und auch die ankommenden Daten mittels des CAN-Description-Files entschlüsselt werden können, wird zum ersten Mal begonnen mit ROS2 zu arbeiten. Nach der erfolgreichen Installation der Distribution *Humble Hawksbill* wird das grundlegende Konzept von ROS2 mittels der sehr ausführlich beschriebenen Dokumentation [47] und des darin enthaltenen Tutorials erlernt. Dies ist dank erster Erfahrungen mit ROS Version 1 sehr verständlich und kann schnell durchgeführt werden, da die Grundkonzepte bereits bekannt sind. Daraufhin wird ein ROS2-Workspace für diese Arbeit, sowie ein eigenes Paket erstellt. Darin wird die im Folgenden beschriebene Software entwickelt.

4.6 Entwicklung der Software

4.6.1 Grundsätzlicher Aufbau und Struktur der Software

Da nun alle Grundlagen und Vorarbeiten durchgeführt wurden, muss nun begonnen werden, die Datenzuordnung und Umwandlung der Daten zu implementieren. Dafür wird vorerst eine mögliche Struktur der Software ausgearbeitet. Abbildung 4.1 zeigt den geplanten Aufbau und die Funktionalität sowie bereits erste Funktionen und deren Namen.

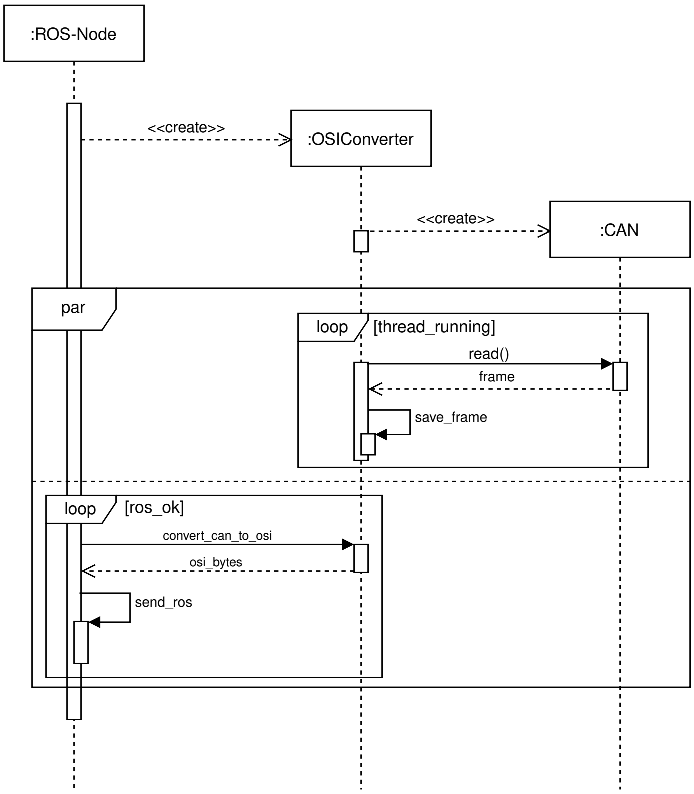

Abbildung 4.1: *Software-Struktur*

Zusammengefasst wird ein ROS-Knoten gestartet, welcher eine Konvertierungsklasse und eine CAN-Klasse instantiiert. Der ROS-Knoten wird ununterbrochen eine Funktion aufrufen oder in dieser verweilen, bis Daten vom CAN verfügbar sind. Falls dies der Fall ist, erhält er die serialisierten Bytes von der Converterklasse und verschickt diese über ROS2 an mögliche Subscriber. Gleichzeitig dazu speichert ein Thread alle über den CAN ankommenden frames ab. Beide Teile greifen dabei auf einen kompletten Datensatz an Objekten zu, welche dann bei der Durchführung noch gegen gleichzeitiges Benutzen oder Überschreiben der Informationen abgesichert werden müssen.
Ausgehend von diesem Konzept werden nun die einzelnen Programmteile implementiert. Im Folgenden werden die jeweiligen Schritte erläutert sowie auf Schwierigkeiten bei der Umsetzung hingewiesen.

4.6.2 CAN-Klasse

Zuerst wird eine Möglichkeit gesucht, die zunächst nur im candump angezeigten CAN-Frames in C++ auszulesen, damit man diese und deren Inhalt dort verwenden kann. Für diesen Zweck wird mithilfe von SocketCAN eine CAN-Klasse geschrieben, die genau dies ermöglicht. In den Konstruktoren dieser Klasse wird ein Socket, wie in der offiziellen Dokumentation von SocketCAN [32] beschrieben, erstellt und konfiguriert. Anschließend werden noch zwei Funktionen geschrieben, um die Daten vom CAN zu lesen bzw. darauf zu schreiben. Diese beruhen auf den beiden C-Funktionen `read()` und `write()`. Code 3 zeigt die Benutzung jener Funktionen .

```cpp
int bytes = read(skt, buf, sizeof(struct can_frame)); // Read a frame from CAN
write(skt, &frame, sizeof(struct can_frame) // Write a frame to the CAN
```

Code 3: *Beispiel der Methoden zum Lesen und Schreiben auf den CAN*

Es werden zwei Konstruktoren geschrieben. Einer, der den Namen des CAN-Interfaces als Parameter, die Sensor-ID und den ROS-Node-Typ bekommt und noch ein zweiter, welcher statt dem Node-Typ einen C++ Vektor an `canid_t` bekommt. Mithilfe von diesem werden dann im zweiten Konstruktor noch Filter für den Socket gesetzt. Dies hat den Vorteil, dass die Nachrichten mit anderen IDs nicht durch den Filter hindurchkommen und folglich nicht durch die Software verarbeitet werden müssen. Dies spart Rechenzeit ein, vor allem wenn neben dem Radar noch weitere Sensoren oder Geräte am gleichen CAN-Netzwerk angeschlossen sind. Der erste Konstruktor wird anschließend für das allgemeine Testen und das der Filter benutzt, um deren Funktionalität zu verifizieren. Schließlich wird dieser Klasse noch ein Destruktor hinzugefügt, welcher den CAN-Socket beim Beenden der Software oder beim Löschen einer Instanz dieser Klasse schließt.

Ein Destruktor ist allgemein eine spezielle Methode, wie auch der Konstruktor. Während ein Konstruktor bei der Erzeugung eines Objektes oder Variablen aufgerufen wird und alle benötigten Parameter für die Funktionalität der Klasse setzt, wie in diesem Fall den CAN-Socket oder die Filter, wird der Destruktor bei der Auflösung eines Objektes aufgerufen. Dabei wird dann in der Klasse zuvor allokierter Speicher oder andere Ressourcen freigegeben. In dem Fall der CAN-Klasse ist dies das Schließen des im Konstruktor konfigurierten Sockets.

Die Klasse wird anschließend noch um eine zweite `read_can()`-Funktion mit Zeiterfassung mittels eines `ioctl()`-Aufrufs erweitert. Damit ist es möglich, den Zeitstempel der zuletzt eingetroffenen CAN-Nachricht, welche zuvor durch das `read()` eingelesen wurde, in einem *struct timeval* abzuspeichern, wenn der `ioctl()`-Aufruf direkt nach der `read()`-Funktion ausgeführt wird. Dies wird exakt wie in [32] beschrieben implementiert. Die Funktion besitzt den Rückgabewert null, wenn das Lesen des frames und Abspeichern des Zeitstempels fehlerfrei abliefen, ansonsten wird der Rückgabewert der jeweiligen Funktion zurückgegeben, welche den Fehler auslöste. Code 4 zeigt die eben beschriebene Funktion, welche im Folgenden zum Lesen der CAN-Frames verwendet wird.

```
int CAN::read_can(void* __buf, struct timeval* tv) {
  nbytes = read(skt, __buf, sizeof(struct can_frame));  // read 1 frame
  int res = ioctl(skt, SIOCGSTAMP, tv); // get timestamp
  if (nbytes < 0) {
    std::cerr << "CAN read failed" << std::endl;
    return nbytes;
  }
  if (res < 0) {
    std::cerr << "CAN get timestamp failed" << std::endl;
    return res;
  }
  return 0;
}
```

Code 4: *Funktion `read_can()` zum Lesen der CAN-Nachrichten und der Zeitmessung*

4.6.3 OSI-Konvertierungsklasse

Nachdem die CAN-Klasse und somit das Lesen der Daten möglich ist, müssen diese Daten gespeichert und umgewandelt werden. Da das Abgreifen der Nachrichten vom CAN unabhängig und parallel zur restlichen Funktionalität der Software geschehen muss, wird hierfür ein C++ Thread erzeugt, welcher aus dem Konstruktor der Konvertierungsklasse gestartet wird und dann dauerhaft die Funktion zum Lesen der CAN-Nachrichten der CAN-Klasse aufruft. Nach erfolgreichem Empfangen wird der jeweilige CAN-frame dann abgespeichert. Code 5 zeigt einen Ausschnitt der while-Schleife, die eben Beschriebenes durchführt.

```
while(rx_thread_running)
{
        if((can->read_can(&frame, &tv)) < 0) break;
        saveFrame(frame);
}
```

Code 5: *Codeausschnitt der while-Schleife zum Auslesen der CAN-Nachrichten*

Die bool-Variable `rx_thread_running` wird dabei kurz vor Aufruf der Schleife auf *true* gesetzt und dann solange ausgeführt, bis sich der Wert auf *false* ändert. Diese Änderung geschieht nur im Destruktor dieser Klasse, damit der Thread das Abfragen der Daten und das anschließende Abspeichern beendet.

Da nun der linke loop-Block aus Abbildung 4.1 implementiert ist, folgt nun die wichtigste Funktionalität, das Umwandeln der Daten in das OSI-Format. Zunächst einmal muss dafür geklärt werden, wie die Daten vor gleichzeitiger Benutzung oder Veränderung gesichert werden können. Das ist wichtig, da der Thread, welcher die Daten vom CAN abgreift, seine CAN-Frames abspeichern muss und diese dann anschließend, wenn alle Objekte vorhanden sind, wieder überschrieben werden. Aus diesem Grund muss es jeweils mindestens zwei Variablen zum Speichern derselben Informationen geben, die einen, welche der Thread ständig verändert und die zweiten, welche einen kompletten Datensatz an erkannten Objekten darstellen.

Für diesen Zweck findet dann zunächst eine C++ std::mutex Verwendung. Diese wird gesperrt, wenn alle Objektinformationen am CAN angekommen sind. Im kritischen Bereich wird dann der komplette Datensatz an Objektinformationen, inklusive der generellen Infos und der Objekte abgespeichert und eine boolean Variable auf *true* gesetzt. Der „Hauptthread“ überprüft dann durch Polling, ob bereits Daten verfügbar sind (Variable ist *true*) oder nicht. Falls ja, können diese umgewandelt werden, falls nein, passiert nichts.

Da dieser Ansatz jedoch nicht sehr effizient ist und ROS ununterbrochen die Funktion aufruft, wird die Mutex durch eine Condition-Variable ersetzt. Im CAN-Thread wird dann, wenn alle Objektinformationen einer Erkennung vorhanden sind, mittels `notify_one()` dem „Hauptthread“ bekannt gemacht, dass neue Daten vorliegen und dieser kann direkt mit der Umwandlung derer beginnen. Dadurch wird das *Pollen* verhindert und der main-Thread wartet nur auf die Informationen und wird benachrichtigt, anstatt regelmäßig nachfragen zu müssen. Wenn die CPU ihre Ressourcen während des Wartens freigibt und der Thread somit nicht dauerhaft die CPU belegt, sondern in den Zustand *runnable* versetzt und durch das `notify_one()` wieder gestartet wird, stellt dies einen großen Vorteil dar und ist im Vergleich zur ersten Variante mit der Mutex sehr viel effizienter.

Die Umwandlung der Objektinformationen findet in der Funktion `convert_can_to_osi()` statt. Diese wartet mittels der eben erwähnen Condition-Variable auf den Erhalt von neuen Objektinformationen. Neue Informationen sind dann vollständig, wenn nach einer CAN-Nachricht mit der ID *0x60A* alle folgenden Nachrichten der ID *0x60B* eingetroffen sind oder eine weitere Nachricht mit der ID *0x60A* eintrifft. In diesem Fall wird dann kein Objekt vom Radar erkannt. Dann werden die neuen Informationen in lokale Variablen gespeichert, um das Überschreiben von neuen Objekterkennungen vom CAN zu verhindern. Mithilfe dieser lokal gespeicherten CAN-Frames wird dann zunächst der SensorDetectionHeader gefüllt und anschließend die einzelnen Detections in einer for-Schleife mittels `radarDetectionData.add_detecion()` hinzugefügt. „Anhang - Konvertierungsfunktion“ zeigt diese Funktion im Detail.

Im Folgenden werden die Zeilennummern der Codezeilen des Anhangs in runden Klammern angegeben.

Beim Header wird zunächst die *number_of_valid_detecions* gesetzt, welche aus der dem generellen Objektlistenstatus entnommen werden kann (69).

Da die empfangenen CAN-Frames in der Funktion, in der sie gespeichert werden, gleich in ein struct mittels der `unpack()`-Funktionen von der aus dem kcd-file generierten C-Datei umgewandelt werden, ist es sehr einfach, diese weiter zu verwenden. Dadurch kann diese mit Hilfe des Befehls `header->set_number_of_valid_detections (local_object_list_status.objects_nof_objects)` gesetzt werden (69).

Daran anschließend wird dasselbe mit dem Messzähler gemacht, welcher ebenfalls aus dem struct der generellen Objektinformation entnommen werden kann, um das Feld *cycle_counter* zu befüllen (71). Daraufhin wird die OSI Zeitmessungs-Variable gesetzt (73 und 74). Da das Radar keine Zeitstempel der Erkennungen mitschickt, wird hierfür der Zeitpunkt der ankommenden CAN-Nachricht mit der ID *0x60A* genutzt. Das hat den Grund, dass nach dieser Nachricht alle erkannten Objekte folgen und jene somit am nähesten am tatsächlichen Messzeitpunkt liegt, auch wenn zwischen der eigentlichen Messung durch das Radar und dem Erhalt der Nachricht über den CAN-Bus Zeit vergeht. Für die Zeitmessung wird, wie bereits erwähnt, nach dem `read()`-Aufruf zum Lesen der Nachricht in der CAN-Klasse der Befehl `ioctl(skt, SIOCGSTAMP, tv)` ausgeführt, welcher dann den Zeitpunkt der Ankunft des gelesenen frames in ein `struct timeval` einspeichert. Dieses wird der OSI-Konvertierungsklasse als Zeiger übergeben. Um nun die Messzeit in OSI zu setzen, müssen aus dem `struct timeval` nur noch die Sekunden und Mikrosekunden ausgelesen und in die OSI-Variable eingesetzt werden. Da OSI die Messzeit in Nanosekunden speichert, die Genauigkeit der Messung jedoch nur in Mikrosekunden vorliegt, müssen diese noch mit dem Faktor 1000 multipliziert und dann in die header-Variable gesetzt werden (73 und 74). Eine noch präzisere Zeitmessung kann durch das Setzen der *SO_TIMESTAMPNS* Option des CAN-Sockets erreicht werden. Hierbei wird dann ein Zeitstempel im Nanosekundenbereich mithilfe eines `struct timespec` erstellt und kann mittels `recvmsg()` und durch das Auslesen einer Kontrollnachricht verwendet werden. Diese Art der Zeitmessung wird jedoch nicht implementiert, da die Genauigkeit des Zeitstempels in Mikrosekunden für die Implementierung in der Fahrzeugarchitektur ausreicht.

Anschließend an das Setzen der Messzeit wird noch der data-qualifier auf *DATA_QUALIFIER_AVAILABLE* gesetzt (76). Daraufhin wird die Montageposition des Radars noch in OSI eingefügt, welche in der Konvertierungsklasse gespeichert ist und im Konstruktor mitgegeben wird (78 bis 83). Zuletzt wird noch der Wert der Sensor-ID im OSI-Header eingefügt (85).

Anschließend folgen die einzelnen Objekterkennungen. Dafür wird eine for-Schleife benutzt, um über alle Detections zu iterieren (90 bis 121). Dabei wird der RadarDetecionData-Variable im ersten Schritt eine Detektion hinzugefügt (93). Anschließend wird die Objekt-ID gesetzt (95 und 96). Diese wird mithilfe folgender Vorschrift in OSI eingefügt:

$$ObjektID_{OSI} = ObjektID_{Radar} + 1024 * sensorID \tag{4.1}$$

Dadurch, dass mehrere Radare des gleichen Typs ihre Objekterkennungen jeweils durchnummerieren, aber kein Austausch mit den anderen Radaren bezüglich der IDs existiert, kann somit eine Unterscheidung der einzelnen Erkennungen gewährleistet werden. Die

Klassifikation wird wie in Unterabschnitt 3.3.1 erläutert auf *OTHER* gesetzt (99), der rcs-Wert wird ebenfalls übernommen (101 und 102). Anschließend wird der radiale Abstand, sowie der Azimut gesetzt (109 und 112). Hierbei müssen die Werte des Radars noch umgerechnet werden, da von Seiten des Radars nur der longitudinale und laterale Abstand gegeben sind. Gleiches gilt für die Radialgeschwindigkeit (119 und 120). An dieser Stelle ist es wichtig zu erwähnen, dass OSI nicht genau spezifiziert, welche Geschwindigkeit (absolut oder relativ) hier benötigt wird. Da aber sowohl in der *GroundTruth*-Nachricht die Geschwindigkeit relativ ist, als auch in der Beschreibung der Radialgeschwindigkeit der *RadarDetection* erwähnt wird, dass die Geschwindigkeit „positiv in Richtung Sensor“ [44] angegeben wird, ist die Schlussfolgerung daraus, dass die Radialgeschwindigkeit in OSI relativ zum Sensor angegeben werden muss. Ansonsten wäre es nötig, eine aktuelle Geschwindigkeit des Fahrzeugs zu erhalten, woran das Radar installiert ist, um eine absolute Geschwindigkeitsinformation bereitstellen zu können. Dabei war es zusätzlich noch wichtig, das Vorzeichen der Geschwindigkeit zu vertauschen, da das Radar diese positiv angibt, falls sich ein Objekt von dem Radar entfernt [52]. In OSI jedoch ist die Geschwindigkeit, wie gerade erwähnt, positiv, wenn sich das Objekt dem Radar nähert. Im Anschluss daran wird noch überprüft, ob die Anzahl der Erkennungen in OSI mit der Variable aus dem Header übereinstimmt (125 bis 130). Zuletzt wird dann der Inhalt der RadarDetecionData-Variable in einen Byte-Stream serialisiert in eine Variable gespeichert (133). Da diese im Funktionsaufruf als Pointer übergeben wird, legt die von Protobuf generierte Funktion `SerializeToString()` den serialisierten Byte-Stream direkt an diese Adresse ab.

Bei dem Befüllen der Werte wird stets auf die Einheiten geachtet, damit die des Radars mit denen aus OSI übereinstimmen. Dies ist vor allem wichtig bei den Umrechnungen der lateralen und longitudinalen Werte in die radialen Werte. Da hier die trigonometrischen Funktionen der math-Bibliothek von C die Winkel in Radiant und nicht im Gradmaß berechnen, gibt es hier keine Schwierigkeiten. Das Speichern der CAN-Frames wird mittels der `unpack()`-Funktionen der generierten C-Datei ermöglicht. Dadurch werden die in den frames enthaltenen Bytes in structs umgewandelt. Die Informationen des CAN-Frames werden dann durch einen solchen `unpack()`-Funktionsaufruf in einem Schritt umgewandelt und den struct gespeichert.

Die einzelnen Objektinformationen der unterschiedlichen Objekterkennungen werden dabei in einem Array dieser structs auf dem Heap gespeichert. Der Speicher dafür wird vor dem Starten der while-Schleife mittels `calloc()` allokiert. Das Freigeben erfolgt dabei wieder im Destruktor der Converter-Klasse.

Ebenso wird mit der Implementierung einer Funktion begonnen, welche die Daten von ROS empfängt und abspeichert bzw. dem Radar sendet. Die wichtigen Informationen hierbei sind die Geschwindigkeit und Gierrate des Fahrzeugs. Im Laufe der Implementierung ist jedoch experimentell erkannt worden, dass das Radar nicht auf die gesendeten Informationen reagiert. Das war an dem *MotionRxState*-Signal ersichtlich, welches sich trotz des Sendens der Daten über den CAN nicht veränderte. Daraufhin wurde der Hersteller des Radars erneut kontaktiert, welcher die Tatsache bestätigte, dass das Radar den Empfang dieser Nachrichten nicht unterstützt. Aus der gleichen E-Mail ist auch hervorgegangen, dass das Datenblatt, welches offiziell für das SR73 Radar ist, nicht spezifisch für das SR73, sondern für alle 77 GHz-Radare der Firma ist. Somit kann nicht sichergestellt werden, dass die Umsetzung des Datenblattes passend für das vorliegende Radar ist.

Über diese Funktionen heraus wird in der Klasse noch eine Funktion implementiert, welche die initiale Konfiguration über den CAN zum Radar sendet. Diese wird der Klasse über den Konstruktor mitgegeben und dann in ein struct eingespeichert. Dessen Information wird dann - genau umgekehrt zum `unpack()` - mittels `pack()` in einen CAN-Frame gepackt und anschließend über die CAN-Klasse mithilfe der Funktion `write_can(frame)` gesendet.

Die aus dem CAN-Thread aufgerufene Funktion `saveFrame(can_frame fr)` entpackt und speichert die ankommenden CAN-Nachrichten je nach ID in die dazugehörigen structs in nur einem Funktionsaufruf. In dieser wird auch die condition-Variable freigegeben, wenn die letzte Objektinformationsnachricht eingetroffen ist und blockiert dann solange, bis die Daten von der Funktion `convert_can_to_osi()` dort lokal gespeichert werden. Gleiches passiert auch für den Fall, wenn keine Objekte erkannt werden. Dies könnte, wie später in Unterabschnitt 5.2.3 genauer erläutert wird, auch nur durch insgesamt zwei statt drei Variablen gelöst werden, da die Umwandlung der Objektinformationen und das Serialisieren selbst bei 255 Objekterkennungen im Mittel immer unter der Ankunftszeit eines neuen CAN-Frames liegen. Zum Testen und im weiteren Verlauf dieser Arbeit wird jedoch weiterhin mit der Version mit drei Variablen gearbeitet.

Da nun das Lesen und Konvertieren der Daten möglich ist, müssen diese jetzt noch über ROS2 versendet werden. Wie dies möglich ist, wird im folgenden Kapitel erläutert.

4.6.4 ROS2-Node

Der ROS-Node ist nun dafür zuständig, dass die serialisierten Daten von OSI verschickt werden. Dafür wir die C++ Bibliothek rclcpp benötigt, welche von ROS2 zur Verfügung gestellt wird und die Funktionalitäten, wie das Erstellen eines Knotens, eines Publishers oder Subscriber sowie weitere ROS2-typische Implementierungsmöglichkeiten für die Sprache C++ bereitstellt. Mit dem Wissen aus dem ROS2 Tutorial wurde dann zunächst ein Publisher implementiert, welcher die Daten nur zyklisch, mit einer festen Rate verschickt. Diese kann durch einen Timer eingestellt werden, der im Konstruktor des Knotens initialisiert wird und dann nach einer fest eingestellten Zeit eine Callback-Funktion ausführt. Im vorliegenden Fall ist das der Aufruf einer Funktion, welche die in der Konvertierungsklasse programmierte Funktion `convert_can_to_osi()` aufruft.

Bei einem ersten Ansatz bricht die Konvertierungs-Funktion gleich zu Beginn ab, falls

eine boolean-Variable, welche den Empfang neuer Objektinformationen bestätigt, noch auf `false` ist. Durch einen speziellen Rückgabewert wird der ROS-Callback-Funktion somit mitgeteilt, dass keine neuen Objekte eingetroffen sind. Daraufhin ist das Senden einer ROS-Nachricht nicht erforderlich. Dabei wird der Wert der boolean-Variable und das Abspeichern der Objektinformationen nur durch eine Mutex geschützt.
Diese Lösung hat jedoch die Schwäche, dass durch das zyklische Abfragen mit beispielsweise einer Rate von 100 Hz bedeutet, dass die Daten, welche gerade als aktuell angezeigt werden, bis zu 10 ms alt sein können. Das ist so nicht akzeptabel, weshalb nach einer anderen Lösung gesucht werden muss.
Ein nächster Ansatz ist, statt eines Timers eine Funktion aus dem Konstruktor aufzurufen, welche in einer while-Schleife mit der Bedingung `rclcpp::ok()` so lange eine Aktion ausführt, bis ROS2 einen Fehler meldet oder der Knoten zum Abbruch gezwungen wird. Dies schlägt zuerst fehl, da zu diesem Zeitpunkt noch versucht wird, einen Subscriber-Node gleichzeitig aus einem Programm heraus zu starten. Das wird mittels eines *MultiThreadedExecutors* versucht. Diesem werden beide Knoten, ein Subscriber und ein Publisher hinzugefügt. Bei dieser Lösung ist es jedoch für den Subscriber nicht möglich, Daten zu empfangen, da der Publisher in der while-Schleife ununterbrochen die Funktion `convert_can_to_osi()` aufruft. Daraufhin wird wieder kurzfristig auf die Timer-basierte Lösung gewechselt. Nachdem jedoch bekannt wurde, dass das Radar keine Geschwindigkeitsinformationen oder Daten bzgl. der Gierrate empfangen kann, wird die komplette Funktionalität des Subscribers vorerst verworfen und der Fokus wird weiter auf den Publisher gesetzt. Dabei kommt dann wieder die Lösung mit der while-Schleife zum Einsatz ohne einen Timer. Da das Polling mit dem hochfrequentierten Aufrufen der Converter-Funktion viel Rechenleistung kostet, musste dafür eine Alternative geschaffen werden. Diese war dann die oben bereits genannte Condition-Varible. Jene ermöglicht es, den Haupt-Thread so lange „idle“ zu legen, bis der CAN-Thread diesen benachrichtigt. Dann kann er die Informationen verarbeiten und wird anschließend wieder in einen wartenden Zustand gesetzt. In der beschriebenen while-Schleife wird dann der Rückgabewert der Konvertierungsfunktion überprüft und bei Korrektheit (null) werden die Daten über ROS versendet. Falls nicht wird eine Fehlermeldung ausgegeben. Fehler können hierbei beim Konvertieren eingetreten oder durch den Empfang von falschen Informationen über den CAN verursacht werden. Dabei kommt es durch die Konvertierungsklasse zu einer weiteren Ausgabe einer detaillierten Fehlermeldung. Code 6 zeigt diese Funktion, welche im ROS2-Node aus dem Konstruktor heraus aufgerufen wird.
Die serialisierten Daten werden dabei in eine eigens entworfene ROS2-Nachricht mittels eines `uint8_t`-Vektors eingefügt. Der Aufbau dieses Nachrichtentyps ist in Code 7 zu sehen
Er besteht nur aus einem Byte-Array, welches wie eben beschrieben mit den OSI-Bytes befüllt wird und einer ROS-Header-Nachricht. Diese beinhaltet einen Zeitstempel und eine FrameID. Jene werden, wie nachfolgend dargestellt, gesetzt und mit verschickt.

```cpp
void PubSR73OSI::publishOSI()
{
        auto msg = sr73_interface::msg::SrRadar(); // SR73 message type for ROS2
        msg.header.frame_id = frameID; // Set the frame ID of the Header message
        buffer = new std::string(); // Initialize the string buffer

        // Publish the ROS2 messages while the node is running and ROS2 is ok
        while(rclcpp::ok())
        {
                if(!osi->convert_can_to_osi(buffer)) { // If the return value is 0 (no error)
                        // Convert the string buffer to a vector of uint8_t and
                        // set it as the data of the ROS2 message
                        msg.data = std::vector<uint8_t>(buffer->begin(), buffer->end());
                        RCLCPP_INFO(this->get_logger(), "Byte-Size: %ld, Byte-Length: %ld",
                                buffer->size(), buffer->length());
                        // Set the current ROS2 timestamp to the ROS2 message header
                        msg.header.stamp = this->now();
                        // Publish the ROS2 message
                        publisher_->publish(msg);
                }
                // If the return value is not 0 (error) -> print an error message.
                else RCLCPP_ERROR(this->get_logger(), "Error in converting CAN to OSI");
        }
}
```

Code 6: *Funktion zum Versenden der ROS2-Nachrichten im Publisher-Knoten*

```
std_msgs/Header header
byte[] data
```

Code 7: *ROS2-Nachrichtentyp für die Radardatenübertragung des SR73*

Somit kann die in Abbildung 4.1 anfangs konzipierte Struktur eingehalten werden.

4.6.5 Weitere Implementierungen und gesamte Projektstruktur

Nachdem jetzt die einzelnen Klassen geschrieben sind, muss nun festgestellt werden, ob der Publisher auch tatsächlich Daten versendet. Außerdem muss es möglich sein, den Publisher, und die von diesem erstellten Klassen, parametrisieren zu können, um einen Wert für die Sensor-ID oder den Namen des CAN-Interfaces ändern zu können. Im ersten Schritt wird für den Empfang der Daten ein ROS2 Subscriber Knoten in der Sprache Python programmiert. Dieser wird im selben Projekt abgelegt und kann mittels des Pakets *ament_cmake_python* neben dem C++ Knoten mithilfe derselben CMakeLists.txt in den ROS2 Installationsordner mit installiert werden. Dieser Knoten dient zur Überprüfung des Empfangens der ROS-Nachrichten und kann zukünftig um eine Visualisierung der Radardaten erweitert werden. Sollte eine solche grafische Darstellung der Objekterkennungen benötigt werden, können die vom C++ Publisher über ROS2 versendeten Daten vom Python-Node so verarbeitet werden, dass dieser daraufhin PointCloud2-Nachrichten über ROS veröffentlicht. Jede einzelne dieser Nachrichten repräsentiert dabei eine Objekterkennung. RViz, ein 3D-Visualisierungsprogramm für das ROS-Framework, kann die Objekte mittels dieser Nachrichten graphisch durch Punkte darstellen.

Nach diesem Schritt ist es möglich, die Daten in einem Python Knoten zu empfangen und eine Konsolenausgabe zu erstellen, welche den Erhalt der Nachrichten bestätigt. Für diesen Zweck wird anfangs die Frame-ID in der Headernachricht benutzt, welche hochgezählt wird und somit beobachtet werden kann, dass Nachrichten versendet und empfangen werden können. Das Hochzählen einer Nummer in der Frame-ID wird im Laufe der Implementierung dann wieder gelöscht und durch einen festen, parametrisierbaren Wert ausgetauscht. Dieser kann mittels eines ROS2-Launchfiles dem Knoten mitgegeben werden.

Ein Launchfile ist eine Datei, welche es ermöglicht, mehrere Knoten mit einem Befehl zu starten. Daher ist dies für den „gleichzeitigen“ Start des Publishers und des Python Subscribers sinnvoll. Ein weiterer großer Vorteil liegt darin, dass ein Launch-file die Parameter den Knoten beim Start sehr einfach mitgegeben kann. Dies erleichtert das Starten, da sonst jeder parametrierbare Knoten mittels „ros2 run *package executable* --ros-args -p <*param*>:=<*value*> ...“ oder mittels einer Parameterdatei aufgerufen werden muss. So ist es einfach möglich, die Parameter einmal zu setzen und diese gleichzeitig mehreren Nodes durch Starten dieses Launchfiles mitzugeben. Ein solches Launchfile kann in ROS2 entweder in Python, YAML oder XML geschrieben werden. Da es hierfür keine Vorgaben oder Präferenzen seitens des Forschungsinstituts gibt, wurde XML verwendet, da es gut dokumentiert, übersichtlich und kompakter ist als Python und YAML. Einen Ausschnitt aus dem Launchfile zeigt Code 8, worin zwei Nodes gestartet werden und jeweils Parameter mitgegeben wird.

```xml
<launch>
    <!-- Radar 0 -->
    <group>
        <!-- Parameters for Radar with ID 0 -->
        <let name="sensorID" value="0" />
        <let name="frameID" value="SR73_OSI_Radar" />
        <!--  let ... -->

        <!-- CAN to OSI publisher here with parameters -->
        <node pkg="sr73" exec="pub_sr73_OSI" name="pub_sr73_OSI" output="screen" >
            <!-- Setting Sensor ID here -->
            <param name="sensorID" value="$(var sensorID)" />
            <!-- Setting ... -->
        </node>

        <!-- Visualization of RadarData with Pointcloud2 -->
        <node pkg="sr73" exec="RViz_sr73.py" name="RViz_SR73" output="screen" >
            <!-- Input FrameID for Error Detection -->
            <param name="frameID" value="$(var frameID)" />
        </node>
    </group>

    <!-- Radar 1 -->
    <!-- ... -->
</launch>
```

Code 8: *Auschnitt aus dem angefertigten Launchfile zum Starten zweier Nodes*

Diese Starterdatei wird dann in einem eigenen Unterordner *launch* im Paket *sr73* abgelegt und mithilfe der CMakeLists.txt mit in den install-Ordner von ROS2 installiert, damit diese aus dem ROS2-Workspace heraus gestartet werden kann, wenn die *setup.bash* aufgerufen wurde. Das Ausführen der *setup.bash* Datei, welche im ROS2-Workspace im *install* Ordner erstellt wird, ist notwendig, um in einem Terminal die mittels colcon erstellten Programme ausführen zu können.

```
ros2_ws
├── build
├── install
├── log
└── src
    ├── sr73
    │   ├── include
    │   │   ├── open_simultion_interface
    │   │   └── sr73
    │   ├── kcd
    │   ├── launch
    │   ├── scripts
    │   ├── sr73
    │   ├── src
    │   ├── CMakeLists.txt
    │   └── package.xml
    └── sr73_interface
```

Abbildung 4.2: *Verzeichnisstruktur des ROS2-Workspaces*

Abbildung 4.2 veranschaulicht graphisch diese Ordnerstruktur mit den wichtigsten Ordnern und Dateien ausgehend vom ROS2-Workspace.

Im sr73 Paket, welches im src-Unterordner des Workspaces liegt, befinden sich alle C++ Source-Dateien im src-Unterordner. Im *include*-Ordner wird OSI hinzugefügt, welches in der CMakeLists.txt auch in den ROS2-Workspace installiert wird. Die übrigen benötigten Header-Dateien sind dabei im *sr73*-Unterordner des *include*-Ordners abgelegt. Die Python Nodes und Skripte findet man in den Ordnern *scripts* und *sr73*.

Zu sehen sind außerdem zwei Dateien, die *CMakeLists.txt* sowie die *package.xml*. Diese werden automatisch beim Erstellen eines ROS2-Pakets generiert. In package.xml befinden sich Meta-Informationen über das Paket, wie der Name, eine Version und eine Beschreibung. Wichtig darin sind die Abhängigkeiten zu anderen Paketen oder Bibliotheken. So benötigt das sr73 Paket eine Abhängigkeit zum sr73_interface, welches den Nachrichtentyp aus Code 7 beinhaltet, um diesen im sr73 Paket benutzen zu können. Die CMakeLists.txt ist die wichtigere der beiden Dateien und beschreibt, wie der Build des gesamten Pakets ablaufen soll. Wichtige Punkte sind hierbei das Kompilieren der C++ Source-Files, das Linken sowie das anschließende Installieren dieser und weiterer Dateien, wie den eben genannten Launch-files oder Python-Nodes in den ROS2 install-Ordner des ROS-Workspaces.

Da jetzt sowohl der Aufbau des Workspaces als auch die Implementierung erläutert wurden und die Software nun fertiggestellt ist, muss jetzt noch die Funktionalität der Software sichergestellt werden. Das wird im anschließenden Kapitel aufgezeigt.

5 Analyse der Ergebnisse

Das Testen von Anwendungen oder Programmen ist in der Informatik einer der wichtigsten Bestandteile beim Entwickeln von Software. Dabei ist das Testen notwendig, um nachweisen zu können, dass das Gesamtsystem oder einzelne Komponenten entsprechend ihrer Spezifikationen funktionieren [25].
Edsger W. Dijkstra, ein niederländischer Informatiker, machte bezüglich des Softwaretestens bereits 1972 folgendes Zitat:

> „program testing can be a very effective way to show the presence of bugs,
> but it is hopelessly inadequate for showing their absence“ [16, S. 864]

Dies bedeutet so viel, als dass durch das Testen zwar Fehler (Bugs) gefunden werden können, es aber nicht ausgeschlossen ist, dass nicht noch weitere Fehler in der Software vorhanden sein können. Somit wird eine möglichst hohe Testabdeckung der Software benötigt, damit viele Fehler ausgeschlossen werden können.
Softwaretests lassen sich dabei in die beiden Kategorien *funktionales* und *nichfunktionales* Testen unterscheiden. Funktionales Testen beschreibt, wie der Name bereits andeutet, die Überprüfung der vorgesehenen Funktion einer Software. Nicht funktionale Tests hingegen untersuchen die Anwendung oder das Programm als Ganzes. Darunter zählt die Effizienz, Wartbarkeit, Übertragbarkeit, Zuverlässigkeit sowie die Benutzbarkeit, wie in der ISO9126 definiert ist. Diese beschreibt grundsätzlich die Qualitätsmerkmale einer Software [27].
Um die im vorherigen Kapitel implementierten Klassen und Funktionen zu testen, wird demnach im Folgenden beschrieben, welche Tests durchgeführt werden. Dabei wird zwischen funktionalen und nichtfunktionalen Tests unterschieden.

5.1 Funktionale Tests

Beim funktionalen Testen geht es, wie im vorherigen Kapitel beschrieben, um die funktionalen Spezifikationen. Es wird dabei untersucht, ob Funktionen, Klassen und schlussendlich das ganze Programm richtig ausgeführt werden.
Dabei gibt es unterschiedliche Arten von funktionalen Tests, die während eines Entwicklungsprozesses eingesetzt werden, um die Qualität der Software zu überprüfen. Diese sind Unit-Tests, Regressionstests, Integrationstests und Abnahmetests. Unit-Tests werden in der Regel von Softwareentwicklern selbst durchgeführt und testen den Code und dessen

Methoden darauf, ob gegebene Eingabeparameter vordefinierte Ausgaben liefern. Bei diesen Tests ist auch die Codeabdeckung ein wichtiger Punkt, um die wichtigsten Fälle abzudecken. Dabei werden die Codeabdeckung (code coverage), die Codepfade (code path coverage) sowie Abdeckung der Methoden (method coverage) unterschieden. Ein guter Unit-Test maximiert dabei alle drei Abdeckungsmöglichkeiten. Regressionstests prüfen, ob Änderungen oder das Hinzufügen neuer Features eine bereits getestete, funktionale Komponente disfunktional machen. Somit ist der Zweck dieser Tests, Fehler zu finden, welche versehentlich in die Software mit eingeführt wurden, wenn diese verändert wird. Integrationstests werden verwendet, um das Zusammenspiel mehrerer Komponenten zu testen. Da einzelne Module meist von unterschiedlichen Entwicklern geschrieben werden, stellen diese Art von Tests sicher, dass die Module richtig miteinander kommunizieren und funktionieren. Zuletzt werden dann noch Systemtests durchgeführt. Diese sind dafür verantwortlich, dass das System, welches aus all seinen Komponenten zusammengebaut wird, festgelegte Anforderungen erfüllt [20].

Es sind für diese Arbeit nicht alle Testarten durchführbar. Das liegt zum Einen daran, dass es keine vollständig formulierten Anforderungen gibt, zum Anderen aber, dass es neben den hier implementierten Komponenten bzw. Klassen noch keine weitere Software, wie ein bereits existentes ROS2-Netzwerk, gibt, worin beispielsweise ein Knoten Daten wie Fahrinformationen sendet. Aus diesen Gründen werden jetzt hauptsächlich Unit-Tests durchgeführt. Zudem wird das Zusammenspiel jener Softwarekomponenten und auch der Gesamtverbund lokal getestet, welcher dann ROS, die Converter-Klasse sowie die CAN-Klasse umfasst. Begonnen wird zunächst mit dem Test der generierten Source-Dateien aus der CAN-Datenbank.

5.1.1 KCD-generierte Dateien

Da das Packen und Entpacken der CAN-Frames zwei generierte Dateien, ein C-Source- und ein C-Header-File, bereitstellen, werden diese zuerst getestet. Für das Testen dieser Dateien, der CAN-Klasse und der Converter-Klasse wird *Google Test* [23], oder kurz *gtest*, verwendet. Google Test ist ein Testing-Framework, welches C++ Code plattformunabhängig testen kann. Dies bedeutet, dass es sowohl auf Linux, als auch auf Windows und MacOS, verwendet werden kann. Es unterstützt dabei auch neben Unit-Tests noch weitere Testarten, welche für diese Arbeit aber nicht von Bedeutung sind.

Da alle Klassen bereits fertig im ROS-Workspace platziert wurden, wird nach einer Möglichkeit gesucht, gtest mit ROS2 zu kombinieren, um die Tests direkt aus ROS heraus ausführen zu können. Da hierfür in ROS2 bereits das Paket *ament_cmake_gtest* existiert, kann dieses sehr einfach in das bestehende sr73-Package mit aufgenommen und anschließend für Tests verwendet werden.

Daraufhin wird dann eine Datei geschrieben, welches das Hauptprogramm aller Google Tests darstellt und außer dem Aufruf der Initialisierung des Frameworks nur noch den

Befehl `RUN_All_TESTS()` ausführt. Dieser führt somit alle Tests, welche in verschiedenen Dateien platziert und mit dieser Hauptdatei zu einer Test-executable zusammengelinkt werden, aus. Die Verteilung auf mehrere Dateien hat den Grund, dass die verschiedenen Tests zu den unterschiedlichen Klassen besser strukturiert sind und es dadurch möglich ist, jene getrennt voneinander ausführen zu können. Alle Testdateien werden dabei in einen Unterordner namens *tests/gtest* im sr73-Package platziert.

Somit wird nach dem Erstellen des Haupt-Testprogramms die erste Datei zum funktionalen Testen des generierten C-files erstellt, welche die Umwandlung der Nachrichten vom CAN in physikalische Werte und andersherum gewährleistet. Der hauptsächliche Grund, warum diese Klasse getestet wird, liegt nicht daran, dass Unsicherheiten bezüglich der Richtigkeit des generierten Codes bestehen, sondern es Unklarheiten im Bezug auf Datentypen, Wertebereiche und Vorzeichen gab, welche im Communication Protocol [52] des Radars beschrieben sind. Daher wird das in Kapitel 8.4 des aktuellen Datenblatts der Version 1.4 [53] angegebene Beispiel zuerst betrachtet. In diesem ist ein Beispiel-Frame mit der ID *0x60B* und daher eine Objektinformation dargestellt. Mithilfe der angegebenen Berechnungen der Werte wurde daher ein Test mittels `TEST()` erzeugt.

Ein Google Test benutzt für alle Tests dieselbe Syntax. Der erste Parameter einer „Test-Funktion“ ist ein TestSuite-Name, welcher eine Übergruppe darstellt, und der zweite beinhaltet den genauen Namen des Tests. Als Suitename wird für den ersten Test *GenKCDFilesTest* benutzt, daraufhin werden dann alle darin enthaltenen Tests unterschiedlich benannt.

Für den ersten Test werden, wie bereits erwähnt, die Beispielbytes des Datenblattes benutzt. Diese werden in meinem Testfall manuell in eine CAN-Nachricht eingefügt. Daraufhin entpackt die generierte Funktion `sr73_objects_general_information_unpack()` diesen Frame wieder und überprüft alle Werte auf Gleichheit mit den angegebenen Werten im Datenblatt des Radars. Der Vergleich auf Gleichheit zweier Werte kann in gtest mittels `ASSERT_EQ(par1, par2)` durchgeführt werden. Eine weitere Möglichkeit wäre die Benutzung von `EXPECT_EQ(par1, par2)`. Der Unterschied der beiden Funktionen liegt darin, dass nach einem fehlgeschlagenen `ASSERT_EQ` der restliche Code des Testes inklusive weiterer Überprüfungen nicht mehr ausgeführt wird. Bei `EXPECT_EQ` ist jedoch dies nicht der Fall. Dort wird der restliche Testcode hinter dem `EXPECT_EQ` noch ausgeführt. Bei diesem Test wird auch bereits ein erster Fehler gefunden. Ein Datentyp der CAN-Datenbank war statt *unsinged* auf *signed* gesetzt und bewirkte daher einen arithmetischen Überlauf und somit ein falsches Testergebnis. Ebenfalls wurde der im Datenblatt beschriebene Offset bei einem Wert nicht eingetragen, was zusätzlich noch zu einer Verschiebung des Wertebereichs führt. Ein Offset wird dann benötigt, wenn es auch negative Werte laut des Datenblattes gibt. Dann muss von umgewandelten Informationen noch der kleinste Wert abgezogen werden, um auf den finalen korrekten physikalischen Wert zu kommen. Da nicht bekannt war, dass alle Werte nur ohne Vorzeichen gespeichert werden, müssen hier im Nachgang noch einige Parameter der CAN-Datenbank angepasst werden. Dies wird dann in der kcd-Datei verändert und anschließend war dieser Test auch erfolgreich.

Die zwei nächsten durchgeführten Tests beschäftigen sich dann erneut mit den Objektinformationen. Dabei werden die angegebenen Minimal- und Maximalwerte überprüft, indem ein frame - nur aus Null-Bytes bestehend - umgewandelt wird und in dem weiteren Test acht 0xFF-Bytes, da hier alle Bits auf eins gesetzt sind. Hierbei werden die umgewandelten Werte jeweils mit den errechneten Werten laut der Vorschrift im Datenblatt verglichen. Dabei wird noch einmal ein Vorzeichenfehler in der Geschwindigkeit erkannt, welcher sofort korrigiert wird. Auch kommt es zu Schwierigkeiten beim Vergleich zweier double Werte. Trotz der Verwendung von `ASSERT_DOUBLE_EQ()` wird der Wert hier nicht als gleich angesehen. Mit dem Blick auf die beiden verglichenen Werte des fehlgeschlagenen Tests kann dann jedoch verifiziert werden, dass diese tatsächlich sehr nahe beieinander liegen und sich nur wegen der trigonometrischen Funktionen zum Umwandeln der longitudinalen und lateralen Entfernungen, sowie Geschwindigkeiten, in die radialen Werte minimal unterscheiden. Daher wird zum erfolgreichen Bestehen des Test die Testfunktion `ASSERT_NEAR()` verwendet, welche neben den beiden zu vergleichenden Werten noch einen dritten Parameter erwartet, auf dessen Genauigkeit die anderen beiden verglichen werden. In meinem Fall wird dafür der Wert 0,0000000000001 verwendet. Das bedeutet, dass sich die beiden Vergleichsparameter um weniger als diesen Wert unterscheiden müssen, was sie schließlich auch tun.

Beim Testen der minimalen Werte kann dann noch ein Fehler im Communication Protocol festgestellt werden. Hier stimmt der Minimalwert der lateralen Geschwindigkeit nicht. Da die Verschiebung des physikalischen Wertes hier schon durch vorherige Tests auf das Datenblatt angepasst wurde, war nicht ersichtlich, wo der Fehler liegt. Durch einen Blick in die aktuellste Version auf chinesisch [51] wird dann der Fehler schnell erkannt. Der Minimalwert wird in der englischen Version 1.4 [53] falsch angegeben und ist nur im neuesten Datenblatt der Firma, welches das Chinesische ist, korrekt.

Anschließend an den Test dieser Nachricht werden die restlichen Nachrichten ebenfalls noch getestet. Zunächst die Geschwindigkeits- und Gierratenwerte. Dort werden gleichermaßen die Minimal- und Maximalwerte manuell errechnet und dann mit den Ergebnissen der `unpack()`-Funktionen verglichen. Diese zeigen, dass das Datenblatt erneut einen Fehler haben muss, wenn man die Werte nach der Rechenvorschrift berechnet. Dabei wird die Genauigkeit nicht korrekt angegeben bzw. bei der Gierrate nicht richtig errechnet. Bei der Geschwindigkeit sollten es laut Datenblatt 163,8 $\frac{\text{m}}{\text{s}}$ maximal geben und die Gierrate sollte nur 327,67 $\frac{\text{deg}}{\text{s}}$ als Maximalwert besitzen. Im Folgenden die dafür getätigte Berechnung (nach dem $\neq$ Zeichen sind die im Datenblatt angegebenen Werte zum Vergleich nochmals aufgenommen):

$$Geschwindigkeit_{max} = (2^{13} - 1) * 0,02 = 163,82 \neq 163,8\,[\tfrac{m}{s}] \tag{5.1}$$

$$Gierrate_{max} = (2^{16} - 1) * 0,01 - 327,68 = 327.67 \neq 327,68\,[\tfrac{deg}{s}] \tag{5.2}$$

Die meiner Ansicht nach falschen Werte liegen auch so in der aktuellsten Version des Datenblattes vor. Daher kann hierbei nicht zu 100 % verifiziert werden, ob die CAN-Beschreibungsdatei und somit meine Ergebnisse oder die des Datenblattes korrekt sind. Daraufhin werden noch die Konfigurationsnachricht des Radars mit maximalen und minimalen Werten, sowie der Radarstatus überprüft. Dabei kommt es zu keinen fehlgeschlagenen Tests. Bei der Versionsinformation werden dann zusätzlich zur Kontrolle der Werte noch gezielt falsche Werte eingesetzt, um einen bewussten arithmetischen Überlauf zu erzwingen. Dies ist ebenfalls erfolgreich, womit bewiesen ist, dass die Werte somit korrekt sind. Die Cluster sowie die Nachrichten über eine Kollisionserkennung werden, wie beschrieben, nicht verwendet und daher auch nicht weiter getestet.

Da nun dieses Modul getestet wurde, folgt anschließend das Testen der Funktionalität der CAN-Klasse.

5.1.2 CAN-Klasse

Der erste Test der CAN-Klasse bestand darin, dass eine Instanz der Klasse erstellt wird, welche diese Daten des virtuellen CANs empfangen kann. Das Senden der exemplarischen CAN-Nachrichten über den virtuellen CAN wird mithilfe eines Threads ermöglicht, welcher am Anfang der Testfunktion gestartet wird und der fünf frames im 10 ms Rhythmus sendet. Nachdem diese empfangen wurden, wird auf die Beendigung des Threads mittels `t.join();` gewartet und anschließend die versendeten frames mit den empfangenen hinsichtlich der Gleichheit der Payload verglichen.

Da dies erfolgreich war, wird in einem nächsten Test die Funktionalität der zu setzenden CAN-Filter untersucht. Dabei wird in einem Thread eine bestimmte Zeitspanne lang eine Nachricht mit einer CAN-ID und danach mit einer anderen gesendet. Der Filter wird nun so gesetzt, dass erst die zweite gesendete Nachricht durch den Filter hindurch zur Software gelangt. Dies wird dann mithilfe von Zeitmessungen verifiziert, damit die ersten frames nicht empfangen werden, sondern erst die zuletzt gesendeten. Dadurch kann die Funktionalität des Filters nachgewiesen und verifiziert werden.

Im nächsten Test wird dann mittels zweier Zeitmessungen die Funktionalität des Zeitstempels einer CAN-Nachricht getestet. Dies wird durch das Mitgeben einer Zeitmess-Variable an die Lesefunktion der CAN-Klasse und durch das Messen der aktuellen Zeit direkt nach dem Aufruf der `read()`-Funktion erreicht. Daran anschließend kann dann zunächst festgestellt werden, dass in der `read()`-Funktion tatsächlich eine Zeit gesetzt wird und dass diese minimal kleiner sein muss, als die der Messung nach dem Aufruf von `read()`. Mit dieser Methode wird die Funktionalität der Zeitmessung verifiziert.

Anschließend wird das Werfen einer Programmausnahme (Exception) der CAN-Klasse getestet, falls versucht wird, einen Socket auf einem CAN-Interface zu erstellen, welches nicht existiert. Dabei wird in der CAN-Klasse beim Öffnen des Sockets und beim `bind()` bei einem fehlerhaften Rückgabewert jeweils eine Exception mittels `throw errno;` geworfen. Das Fangen und Auswerten dieser Exception wird dann in der Testfunktion versucht. Dabei wird festgestellt, dass es einen Unterschied macht, ob man eine CAN-Klasse mittels `CAN can = new CAN("vcan1", 0, filters)` oder `CAN can = CAN("vcan1", 0, filters)` erstellt. Der Unterschied dabei liegt nur bei dem Speicherort des Objektes. Während es im ersten Fall eine Speicherallokation auf dem Heap ist, wird das zweite auf dem Stack gespeichert. Das Werfen einer Exception sollte dabei jedoch in beiden Fällen funktionieren. Dies wird auch experimentell durch Einfügen von `throw` Anweisungen erzielt. Der Grund liegt daher nicht am Werfen einer solchen Fehlermeldung, sondern bei der `bind()`-Funktion. Diese hat, wenn sie durch `new` am Heap erzeugt wurde, niemals einen Fehlerwert zurückgegeben, weshalb auch keine Exception geworfen wurde. Auf dem Stack ist dies schon der Fall. Um das Problem zu lösen wird daher am Anfang des Konstruktors mittels `if(if_nametoindex(can_interface.c_str()) == (unsigned int)NULL)` festgestellt, ob das Interface am Computer verfügbar ist. Falls dies nicht der Fall ist, wird hier eine Exception geworfen. Diese Lösung funktioniert schließlich, sowohl mit der Objektvariable am Stack, als auch am Heap.

In einem letzten Testfall für diese Klasse wird dann noch überprüft, ob neben dem Lesen auch ein Schreiben von Nachrichten auf den CAN möglich ist, da dies beim Starten der OSI-Konvertierungsklasse benötigt wird. Darin wird die initiale Konfiguration des Radars versendet. Um zu verifizieren, ob das Senden nun grundsätzlich möglich ist, wird hierfür ein Testfall erstellt, wobei in einem extra erstellten Thread frames mittels der CAN-Klasse gesendet und im Hauptthread des Testes empfangen werden, um diese auf Gleichheit zu überprüfen.

Das Senden in den vorherigen Tests wurde durch Aufrufe der `system()`-Funktion erreicht, worin dann der Terminalbefehl `cansend` verwendet werden konnte, was das Senden generell einfacher machte, da diese nicht extra durch einen zusätzlichen Socket und der Zuordnung aller Parameter verschickt werden müssen.

5.1.3 OSI-Converter-Klasse

Da die CAN-Klasse nun auch funktional getestet wurde, folgen nun Testfälle zur Verifikation der Funktionalität der Konvertierungsklasse. Ein erster Testfall überprüft das Erstellen und den Erhalt eines OSI-Bytestreams. Dafür wird eine Instanz der Klasse erstellt und parallel dazu wieder in einem Thread Nachrichten über den virtuellen CAN versendet. Daraufhin folgt die Erstellung einer String-Variable, welche zunächst darauf überprüft wird, dass diese keinen Inhalt besitzt. Anschließend daran kommt es zum Aufruf der Funktion zum Konvertieren der CAN-Daten und es wird im Anschluss verifiziert, dass der String nun einen Inhalt besitzt. Dies stellt somit einen Test zum reinen Erstellen und dem Aufruf der Converter-Funktion dar.

Ein zweiter Testfall, welcher ähnlich dem Ersten ist, beinhaltet die Analyse des Strings, in welchem die serialisierten OSI-Daten enthalten sind. Diese werden somit deserialisiert und die darin enthaltenen Variablen werden mit denjenigen, welche der OSI-Konvertierungs-Klasse im Konstruktor mitgegebenen wurden, verglichen. Dies stellt die Überprüfung der korrekten Werte der Sensor-ID der Montageposition dar.

Beim nächsten Test wird dann statt der Sensor-ID und der mounting-position die Werte des Messzählers der Nachricht, die Anzahl an Objekterkennungen sowie der data-qualifier analysiert, ob diese mit den über den vcan0 verschickten Daten übereinstimmen.

Der nächste Testfall überprüft, ob eine initiale Konfiguration über den vcan0 versendet wird. Dies ist später zum Parametrieren eines realen Radars relevant. Dafür wird zuerst eine CAN-Klasse und anschließend eine Konvertierungsklasse erstellt. Direkt nach dem Erstellen der OSI-Klasse wird dann mittels `can.can_read(&frame)` auf die Nachricht mit der ID *0x200* gewartet. Diese wird dann hinsichtlich der enthaltenen Bytes analysiert, ob die gesendeten, korrekt umgewandelten, Bytes den empfangenen entsprechen. Im gleichen Testfall wird dann noch anschließend getestet, ob das Setzen einer anderen Sensor-ID und das Senden anderer frames bewirkt, dass die OSI-Klasse auch diese Daten erfolgreich empfangen kann. Somit ist auch das Setzen der richtigen Filter für eine andere ID und das anschließende Empfangen und Auslesen möglich.

Der nächste Test überprüft die Grenzen der Objektdaten-Speicherarrays und ob es möglich ist, eine solch große Anzahl an Elementen zu speichern, zu serialisieren und anschließend wieder auszulesen. Dafür wird im Thread, welcher zu Beginn des Tests gestartet wird, zunächst eine Nachricht mit der Identifikation *0x60A* verschickt, welche als Objektzähler den höchsten Wert, also 255 enthält. Dann werden anschließend 255 CAN-Nachrichten, alle mit demselben Inhalt, nur mit aufsteigender Objekt-ID, versendet. Dabei werden die IDs von 0 bis 254 verschickt. Daraufhin wird, parallel dazu, eine Konvertierungsklasse erstellt und auf den Erhalt aller CAN-Nachrichten gewartet. Anschließend wird, wie auch schon in den vorherigen Tests, wieder ein `osi3::RadarDetectionData` Objekt namens *erg* erzeugt und mittels des Aufrufs `erg.ParseFromString(*buffer);` dann aus dem serialisierten Byte-Stream aus dem `convert_can_to_osi(buffer)`-Funktionsaufruf die Daten extrahiert. Daraufhin werden verschiedene Indizes der erkannten Objekte hinsichtlich deren Daten überprüft. Dabei enthalten ist das erste und zweite Objekt, eines exemplarisch beim Index 121 und eins bei dem Index 254, was der letzte in diesem Fall ist. Dabei werden jeweils alle empfangenen Daten, die des SensorDetecionHeader's und die der Erkennungen, überprüft. Dabei fällt auf, dass mit dem Index 254 noch alle Werte korrekt sind, ein Index von 255 jedoch falsche Werte liefert. Auf Daten dieses Indizes dürfte kein Zugriff möglich sein, da es diesen gar nicht gibt bzw. an diesem keine Objektinformationen vorhanden sind. Ein Index von 256 liefert einen Segmentaion fault. Daher wäre es theoretisch möglich, eine Objekterkennung aus dem Index 255, dessen Daten jedoch falsch sind, zu verwenden. Dies wird mithilfe der for-Schleife im Programm verhindert.

Als nächstes werden noch drei Testfälle geschrieben, um falsch gesendete Werte in der Versionsnummer des Radars zu überprüfen. Diese Funktionalität wird erst spät in den

Code mit eingefügt. Beim Aufruf der Konvertierungsfunktion wird zu Beginn kurz überprüft, ob die empfangene Version mit einer fest einprogrammierten übereinstimmt, da alle Radare des Forschungsinstituts die gleiche Version verwenden. Trotzdem ist eine Abänderung dieser Version jederzeit möglich. Dabei werden in den drei Tests jeweils ein Teil der Version verändert und beobachtet, ob die Funktion einen fehlerhaften Rückgabewert liefert. Dies geschieht in allen drei Tests erfolgreich.

Die letzten beiden Tests der Klasse überprüfen die Konvertierungsfunktion, ähnlich der vorherigen drei Tests, darauf, ob eine falsche Sensor-ID oder ein Sortierungsindex erkannt wird. Dies bestätigt sich in beiden Fällen.

Somit war die Funktionalität der OSI-Konvertierungsklasse verifiziert und es fehlte noch zu zeigen, dass auch der ROS-Knoten ordnungsgemäß funktioniert.

5.1.4 ROS-Publisher-Node

Um dies zu verifizieren, wird nicht die zuvor genannte Methode mit Google Tests verwendet. Es wird zwar vorerst nach einer Möglichkeit gesucht, den Knoten in einem gemeinsamen Test mit den anderen Komponenten mit zu testen, jedoch ist dies mit meinem Ansatz nicht möglich. Das Problem dabei ist nämlich, dass zwei Threads gestartet werden, der erste zum Senden von CAN-Nachrichten, und einen zweiten, welcher den Publisher ausführt und die Daten über ROS versendet. Im „Haupt-Thread“ des Tests sollten dann die Daten empfangen und verglichen werden. Da dies bei mir jedoch aus unerklärlichen Gründen nicht korrekt funktioniert, wird nach einer anderen Möglichkeit gesucht, um den Publisher zu testen.

Daraufhin wird eine Möglichkeit implementiert, welches einen Test durch ein extra Launch-File ausführt. In dieser Datei werden nacheinander mehrere Knoten gestartet. Der Erste, welche nur Nachrichten über den virtuellen CAN versendet. Anschließend dann der eigentliche Publisher, so wie er auch aus dem „normalen“ Launch-File heraus gestartet wird. Zuletzt noch ein spezieller Subscriber-Knoten. Dieser bekommt alle Parameter und erwartete Werte durch Auslesen einer Datei und anschließendes Abspeichern der Werte in Variablen. Das ermöglicht ein schnelles Testen von richtigen und auch falschen Werten, da das gesamte Projekt nicht neu kompiliert werden muss. Stattdessen werden nur Parameter in der Datei verändert, welche dann vom Subscriber eingelesen werden. Ein weiterer Vorteil neben der exakt gleichen Startmethode des Publisher-Knotens und des schnellen Austauschs mehrerer Parameter ist, dass dieser Test auch unabhängig von gtest gestartet und auch über einen längeren Zeitraum beobachtet werden kann. Dadurch ist es möglich, neben der Funktionalität des Knotens auch gleich die Fehleranfälligkeit von ROS und die Datenrate sowie Frequenz zu überprüfen, was nicht zum funktionalen Testen zählt, jedoch trotzdem einen Vorteil bietet.

Durch Vergleich aller Parameter der umgewandelten CAN-Bytes mit denen in der Parameterdatei und eines sofortigen Abbruches bei Ungleichheit kann somit die Funktionalität des Knotens, also das korrekte Senden der richtigen Daten nachgewiesen werden.

5.2 Nichtfunktionale Tests

Was unter nichtfunktionalen Tests verstanden werden kann, wurde bereits im Kapitel 5 dargelegt. Zusammengefasst sind das alle Tests, die nicht direkt die vorgesehene Funktion einer Software testen. Da dies ebenso relevant ist wie die eigentliche Funktion, werden im Folgenden noch die nichtfunktionalen Tests beschrieben und wie diese durchgeführt werden.

5.2.1 Überprüfung auf Memory Leaks

Zuerst wird die Software auf Speicherlücken, sogenannte *memory leaks*, überprüft. Diese entstehen dann, wenn Speicher auf dem Heap allokiert wird, beispielsweise für ein Objekt oder eine Variable, dieser jedoch beim Beenden eines Programms nicht mehr freigegeben wird. Dafür muss sichergestellt werden, dass alle erstellen Objekte zum Programmende auch wieder gelöscht werden. Viele andere Programmiersprachen besitzen zum Umgehen dieses Problems einen Garbage-Collector, welcher durch eine automatische Speicherverwaltung sicherstellt, dass es zu keinen Speicherproblemen oder -lücken kommt. Da C++ und C jedoch sehr hardwarenah sind und auch keine solche automatische Speicherbereinigung besitzen, muss dies durch den Programmierer selbst sichergestellt werden.

Memory leaks werden hierbei zu den nichtfunktionalen Tests gezählt, da sie nicht direkt die Funktionalität des Programms betreffen, sondern vielmehr die Speicherbelegung und Fehler beim langfristigen oder häufigen Ausführen des Programms.

In C++ geschieht eine Speicherallokation mittels des Operators `new`. Damit kann ein Objekt oder eine Klasse erstellt werden und dessen Speicher und die Parameter der Klasse werden dann auf dem Heap gespeichert. Mittels `delete` kann dieses dann wieder gelöscht werden. In der Programmiersprache C geschieht dies mittels `malloc()` oder `calloc()`. Dabei muss dieser Speicher dann wieder mithilfe von `free()` freigegeben werden.

Eine Möglichkeit, um eine Speicherlücke zu erkennen, ist zu überprüfen, ob alle `new`-Aufrufe ein passendes `delete` besitzen und, im Falle einer Allokation in C die beiden Äquivalenten, jeweils in Paaren im Code vorkommen oder aufgerufen werden. Bei Klassenvariablen ist das Freigeben von Speicher im Destruktor möglich, der beim Löschen eines Objektes aufgerufen wird. Da die Methode der einfachen Überprüfung unzuverlässig ist und nicht sichergestellt werden kann, ob der Speicher tatsächlich freigegeben wird und die speicherlöschenden Funktonen aufgerufen werden, ist eine andere Methode gefragt.

Valgrind [57] ist ein genau ein solches Tool, welches verwendet werden kann, um Spei-

cherlücken festzustellen. Allgemein kann es neben automatischen Speichermanagement auch noch Profiling des Codes und Probleme bei parallelen Anwendungen feststellen.
Daher ist Valgrind für diesen Zweck geeignet. Zuerst wird nach der Installation des Tools und eines Tests anhand einer speziell konzipierten Beispielanwendung dann mit der CAN-Klasse als Executable ausgeführt. Valgrind kann dann eine Executable mittels `valgrind -tool=memcheck <executable>` auf Speicherlücken überprüfen. Daraufhin wird bereits Speicher entdeckt, welcher nicht freigegeben wird. Es war das fehlende Löschen des Vektors der can_id's für die Filterung des Nachrichtenempfangs. Dies wird anschließend mittels `delete [] rfilter` korrigiert und Folglich sind in dieser Klasse keine Speicherprobleme mehr vorhanden.
Anschließend wird Valgrind noch auf die OSI-Konvertierungsklasse und zuletzt während der Ausführung des ROS-Knotens ausgeführt. Dabei sind jeweils keine direkten Speicherlücken zu erkennen, nur Speichersegmente, welche eventuell noch erreichbar sind.
Ebenfalls wird beim Testen auf Speicherlücken festgestellt, dass ein *log*-Aufruf in ROS2 mittels `RCLCPP_INFO(...)` im Destruktor des Publisher-Knotens nicht mehr möglich ist. Da zur Beobachtung der Destruktor-Aufrufe hier zunächst ROS2-Logger-Konsolenausgaben benutzt werden, kommt es zu einem Fehlercode beim Beenden des Knotens. Das kann den Grund besitzen, dass die rclcpp-Bibliothek zum Zeitpunkt des Destruktoraufrufs bereits heruntergefahren wurde bzw. einige Funktionen nicht mehr bereitstellte. Das Wechseln der Ausgabe auf den Standard-Output `std::cout` kann dieses Problem beheben.
Somit kann den vorherigen Tests geschlussfolgert werden, dass keine dauerhaften Speicherlücken durch die Ausführung des Programms entstehen.

5.2.2 Schließen der Sockets

Als nächstes wird noch getestet, ob der in der CAN-Klasse geöffnete Socket wieder geschlossen wird. Das hat den Grund, dass nicht nach mehrmaligem Ausführen des Programms noch unbenutzbare, geöffnete Sockets im System vorhanden sind.
Zu diesem Zweck wird nur im Destructor der CAN-Klasse der Rückgabewert der Funktion `close()` überprüft und anschließend durch erneutes Schließen des Sockets beobachtet, ob er bereits beim ersten Mal geschlossen wurde. Das konnte mithilfe des Rückgabewertes -1 statt null verifiziert werden. Somit wird der Socket jeweils korrekt geschlossen.
Da jener zusätzlich eine Klassenvariable ist und nicht auf dem Heap erstellt wird, kann daraus geschlussfolgert werden, dass er korrekt geschlossen und gelöscht wird.

5.2.3 Datenmenge und Performance bei unterschiedlicher Anzahl an Objekterkennungen

Die letzten nichtfunktionalen Tests werden die Performance und Auslastung des Systems bei unterschiedlicher Anzahl an Objekterkennungen messen. Der Computer, auf dem diese Messungen durchgeführt werden, ist ein Dell Latitude 5480 Laptop mit einem i5-6300U Prozessor. Dieser besitzt 2 physische Prozessorkerne und kann dank Multithreading-Technologie bis zu 4 Threads gleichzeitig auf den zwei Kernen ausführen. Die Grundtaktfrequenz des Prozessors beträgt 2,40 GHz, die Turbo-Taktfrequenz 3,00 GHz [26]. Außerdem sind 16 GB RAM verbaut. Wie bereits erwähnt, wird das Betriebssystem Ubuntu in der LTS-Version 22.04.1 für diese Messungen verwendet. Für die Tests werden unterschiedliche Tools benutzt. Pidstat [45] dient zum Auslesen der Prozessorauslastung, canbusload [9], ein Befehl der SocketCAN Utilities, ermöglicht das Messen der Auslastung eines CAN-Interfaces. Gprof [19], ein Profiling-Programm, macht es möglich, die Software dahingehend zu analysieren, wann und wie oft welche Funktion aufgerufen wird und welchen Anteil diese an der gesamten Programmlaufzeit besitzt. Zeitmessungen von und innerhalb von Funktionen werden mittels `timespec_get()`-Aufrufen durchgeführt, wobei die Zeitstempel mit einer Genauigkeit im Nanosekundenbereich in einem `struct timespec` gespeichert werden und dadurch die Laufzeiten durch die Differenz der Zeitpunkte errechnet werden können. Weitere Informationen folgen in diesem Kapitel.

Zunächst wird der Daten-Overhead gemessen, welcher durch das Umwandeln der CAN-Frames über OSI und schließlich noch durch den Header in der ROS-Nachricht entsteht. Dabei werden im Allgemeinen mehrere Messungen mit realen Daten des Radars und Andere mit selbsterstellten Daten über den virtuellen CAN benutzt. Das hat den Vorteil, dass stets gleiche Daten verwendet werden können, damit eine maximale Auslastung erreicht werden kann. Bei den Messungen mit dem Radar sind die Entfernungen und Geschwindigkeiten jeweils unbekannt bzw. variieren, bei den synthetischen Daten werden jeweils die höchst möglichen Werte gesetzt, um eine maximale Auslastung zu erzielen.
Zum Messen der Bytesmenge werden somit eine verschiedene Anzahl an Objektdaten über den virtuellen CAN verschickt und anschließend umgewandelt. Dabei wird stets die Menge an Bytes gemessen, welche die OSI-Nachricht und welche die ROS-Nachricht besitzt. Die realen Messergebnisse des Radars werden hier zunächst nicht verwendet, im nächsten Kapitel sind diese dann aber relevant.
Die Anzahl der Bytes, welche über den CAN verschickt werden, berechnen sich dabei mit der Vorschrift $Byteanzahl = (111Bit + 111Bit * Erkennungen)/8$. 111 Bit deshalb, da ein gesamter Standard CAN-frame genau aus 111 Bits besteht, wenn man die Annahme trifft, dass die drei Bits zwischen Nachrichten (IFS, *Inter-Frame Space*) regulär bei Datennachrichten versendet werden und die Payload aus vollen acht Bytes besteht. Die ersten 111 Bit der Berechnung stellen die der Nachricht mit der ID *0x60A* dar, welche die Liste an Objekterkennungen liefert. Diese wird trotz der Angabe im Datenblatt, dass die Nachricht nur vier Byte lang ist, mit allen acht berechnet, da am realen CAN auch acht vom Radar verschickte Bytes empfangen werden. Die anderen Bits entsprechen denen der Nachricht mit der ID *0x60B*. Es wird die Gesamtanzahl an Bytes inklusive der Kontrollbits von CAN bei dieser Messung betrachtet, da dies bei OSI und ROS genauso

geschieht.

Objektanzahl	CAN-Byteanzahl	OSI-Byteanzahl	ROS-Byteanzahl
0	13,875	88	120
1	27,75	144	180
2	41,625	200	240
5	83,25	368	400
10	152,625	648	680
20	291,375	1207	1240
50	707,625	2888	2920
100	1401,375	5687	5720
255	3552	14369	14410

Tabelle 5.1: *Vergleich der Anzahl an Bytes der Nachrichten bei unterschiedlicher Menge an Objekterkennungen*

Tabelle 5.1 zeigt die Anzahl der Bytes bei unterschiedlicher Anzahl von Objekterkennungen. Zu sehen ist, dass die Byte-Menge mit steigender Anzahl an Erkennungen zunimmt, was trivial erscheint. Außerdem zu erkennen ist, dass sich die Bytes von OSI mit denen von ROS immer nur um einen gewissen Betrag von ca. 30-40 Bytes unterscheiden, was somit Größe des ROS-Nachrichten-Headers sein muss. Da die ROS2-Byteangabe nur in Kilobyte gegeben ist und nur eine Genauigkeit von 0,01 KByte durch das Kommando *ros2 topic bw /SR73_OSI_Radar_0* bereitgestellt wird, kann hier keine genauere Angabe zur Bytemenge des ROS2-Headers offeriert werden.

Der Faktor, wie viel mehr Daten nun durch die Verwendung von OSI und ROS benötigt werden, kann somit errechnet werden. Bei keiner Erkennung ist das der Faktor 6,354 durch die Verwendung von OSI und 8,849 durch das anschließende Versenden über ROS. Somit werden bei keinem erkannten Objekt 8,849 mal so viele Bytes über ROS2 verschickt als die CAN-Nachricht des Radars lang ist.

Bei mehreren Erkennungen sinkt dieser Faktor dann. Bei einer maximal möglichen Anzahl von 255 Objekten ist es bei OSI der Faktor 4,045 und 4,057 bei ROS. Dies veranschaulicht, dass der Overhead an Daten durch die Verwendung von OSI bzw. ROS2 immer mindestens Faktor vier im Vergleich zu den Radardaten beträgt, wenn die größten Werte am CAN anliegt. Der Fall, dass alle Werte (Geschwindigkeit, Position etc.) genau den Wert null besitzen, wurde hier nicht explizit getestet. Hierbei wäre jedoch der eben genannte Faktor kleiner und könnte auch einen geringeren Wert als vier annehmen.

Abbildung 5.1 veranschaulicht die Ergebnisse aus Tabelle 5.1 nochmals graphisch. Zu erkennen ist, dass die ROS-Bytes immer sehr nahe an denen von OSI liegen, was dem Unterschied von 30-40 Bytes entspricht. Sehr deutlich ist ebenfalls zu sehen, dass durch Verwendung von OSI und der Serialisierung der Informationen ein großer Overhead entsteht, welcher zuvor ja bereits errechnet wurde.

Da nun die Anzahl der Daten veranschaulicht wurde, ist es jetzt noch wichtig zu erfahren, wie bei den genannten Situationen die Auslastung des CAN-Busses und die der CPU ist

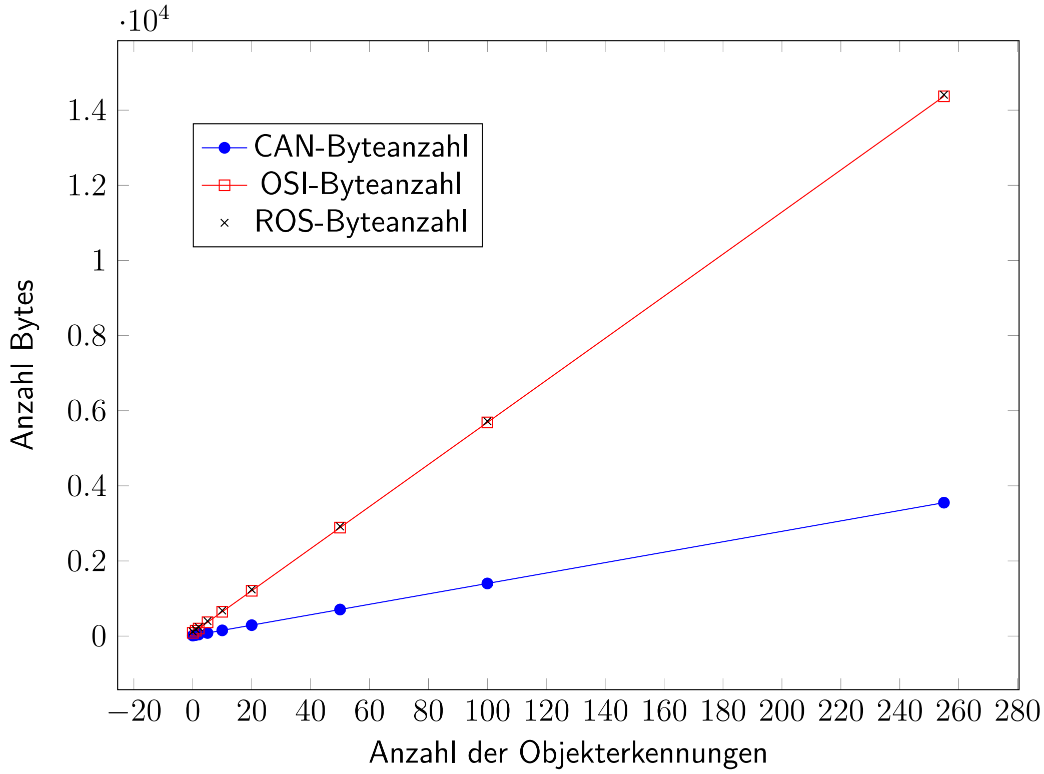

Abbildung 5.1: *Anzahl an Bytes der verschiedenen Nachrichten*

und welche Funktionen dabei wie viel Rechenzeit in Anspruch nehmen.
Die verwendeten Tools sind bei diesem Test canbusload, pidstat und gprof. Eine nanosekundengenaue Zeitmessung stellte `timespec_get()` bereit.
Mittels gprof wird der Anteil einzelner Funktionen an der gesamten Programmdauer gemessen. Dabei kommt es bei unterschiedlichen Messungen zu verschiedenen Ergebnissen. Da diese jedoch grundsätzlich die gleichen Aussagen tätigen, es durch unterschiedliche Laufzeiten aber zu leicht unterschiedlichen Ergebnissen bei den Prozentangaben kommt, folgen hier die wichtigsten Erkenntnisse, welche von dem Großteil der Messungen hervorgehen:

1. In `convert_can_to_osi()` wird mit über 30 %, aber dennoch wenigeren Aufrufen bei mehr Objekten, viel Zeit im Vergleich zur Gesamtlaufzeit verbracht.

2. In `saveFrame()` wird mit durchschnittlich 20 % viel Rechenzeit benötigt, vor allem bei den Messungen mit mehreren Objekterkennungen steigt dieser Anteil.

3. `read_can()` wird mit ca. 10 % Anteil der Rechenzeit am Gesamtprogramm, und bei einer höherer Anzahl von Objektwahrnehmungen noch öfter aufgerufen.

4. In `publishOSI()` wird trotz nur eines Aufrufs dieser Funktion ungefähr 6 % Zeit gerechnet.

Viele der eben genannten Punkte sind direkt nachvollziehbar.

Punkt 1 ist logisch, da die Funktion auf den Erhalt der Objekte wartet und zusätzlich dazu dann noch die Radardaten in OSI konvertiert. Auch wenige Aufrufe bei mehr Objekten ist stimmig, da diese, unabhängig von der Objektanzahl, nur durchlaufen wird, wenn alle Objekte vorhanden sind.

Punkt 2 ist ebenfalls nachvollziehbar. Bei mehreren Objekten steigt auch die Anzahl der Aufrufe zum Einspeichern der Informationen. Außerdem muss diese Funktion auf das Abspeichern der Werte warten.

Außerdem wird die Funktion `read_can()` sehr oft aufgerufen, was ebenfalls stimmig ist, da bei steigender Objektanzahl mehrere CAN-Nachrichten vom Radar verschickt werden.

Auch Punkt 4 ist nachvollziehbar, da diese Funktion aus dem Konstruktor des ROS-Knotens nur einmal aufgerufen wird und dann in einer while-Schleife die Informationen verschickt, solange bis der Knoten beendet wird.

Daraus ist erkennbar, dass sehr viel Zeit in den Funktionen zum Umwandeln der Daten und Speichern von CAN-Nachrichten gerechnet wird. Deshalb ist es anschließend von Interesse, wie viel Zeit genau das Umwandeln der Objekte in den serialisierten OSI-Bytestream benötigt.

Objektanzahl	Minimal	Maximal	Durchschnittlich
0	9,06	87,69	14,45
1	12,51	152,46	20,40
2	13,10	159,95	20,64
5	16,39	104,00	23,25
10	18,11	484,15	26,38
20	20,54	125,16	35,19
50	31,88	149,99	53,36
100	43,68	589,09	77,32
255	82,53	238,12	130,97

Tabelle 5.2: *Zeitmessungen der Umwandlungen und Serialisierungen von Objekten abhängig von der Objektanzahl in Mikrosekunden*

Tabelle 5.2 und Abbildung 5.2 zeigen beide, dass sich jeweils die minimale Dauer der Konvertierung, sowie der Durchschnitt mit zunehmenden Objekten nahezu proportional erhöhen. Der Grund, warum es nicht direkt proportional ist, könnte die Serialisierung durch OSI sein. Die Maximalwerte schwanken dabei sehr und sind unregelmäßig verteilt. Aus diesen Ergebnissen kann gefolgert werden, dass die Umwandlung inklusive Serialisierung der Objektdaten im Durchschnitt immer weniger als 222 Mikrosekunden dauert. Das ist genau die Zeit, die ein kompletter CAN-frame auf einem 500.000 $\frac{\text{kBit}}{\text{s}}$ CAN-Bus benötigt, um verschickt oder empfangen zu werden. Daher ist im Mittel das Serialisieren immer schneller, als die Zeitspanne zwischen zwei direkt nacheinander gesendeten CAN-Nachrichten. Somit könnte bei der Implementierung auch auf die jeweils dritten Variablen zum lokalen Speichern verzichtet werden und das `notify()` der Bedingungsvariable auch erst am Ende der Funktion aufgerufen werden. Der Socket würde dann im

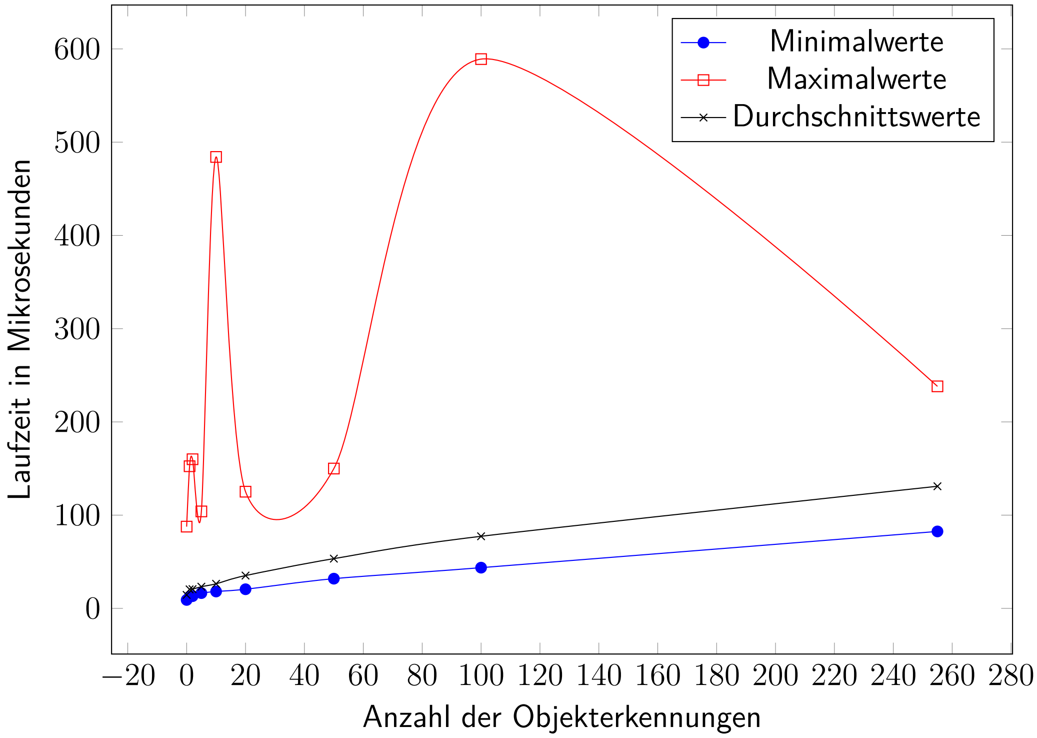

Abbildung 5.2: *Minimal-, Maximal- und Durchschnittsdauer der Konvertierung und Serialisierung*

Falle eines längeren Umwandelns eingehende Nachrichten wie ein Puffer speichern. Dies ist durch ein Beispielprogramm verifiziert worden. Da in diesem Fall auch die Maximalwerte über den CAN verschickt, sowie auch in OSI umgewandelt werden, ist es möglich, diese Aussage zu treffen.

Des Weiteren wird noch die Frequenz gemessen, mit welcher die Nachrichten in ROS verschickt werden. Hierbei ist ein Unterschied zwischen den realen Radardaten und den synthetischen Daten über den vcan0 zu sehen. Während der Durchschnitt aller Nachrichten des Radars exakt mit 30,303 Herz verschickt werden, liegen die Werte beim virtuellen CAN bei ungefähr 26 Hz. Der Grund dafür liegt bei der Implementierung des Programms, welches diese Daten schickt. Trotz eingebauter Timing- und Wartefunktionen ist es nicht möglich, eine konstante Frequenz von 30,303 Hz, wie beim Radar, erreichen zu können. Die Herz-Angabe der über den „realen“ CAN verschicken Objektinformationen entspricht genau der Angabe der Frequenz im Communication Protocol, dass die Objekte alle 33 ms versendet werden.

Abschließend werden noch die Prozessor- und CAN-Bus-Auslastung gemessen. Bei der Prozessorauslastung werden Durchschnittswerte zweier Messungen betrachtet und davon nochmals der Mittelwert gebildet. Die Auslastung des CAN-Busses wird mittels „`canbusload vcan0@500000 > busload.txt`" in unterschiedliche Dateien gespeichert und dann der Durchschnitt ausgelesen.

Tabelle 5.3 zeigt die Ergebnisse der Messungen. Während die Prozessorauslastung unerwartete Werte zeigt, ist die CAN-Auslastung nachvollziehbar mit der steigenden Objek-

Objektanzahl	CPU-Auslastung in %	CAN-Bus-Auslastung in %
0	1,80	0
1	1,84	1
2	1,85	2
5	1,88	4
10	1,90	8
20	2,09	14
50	2,10	35
100	2,07	70
255	2,05	179

Tabelle 5.3: *CPU- und CAN-Bus-Auslastungen des ROS2-Knotens abhängig von der Anzahl an Objekterkennungen*

tanzahl gestiegen. Hier ist beim letzten Eintrag zu sehen, dass es sogar mehr als 100 % Buslast wären. Dies kann rechnerisch gezeigt werden:

$$Bitanzahl = (30 * 256 + 2) * 111 Bit = 852702 Bit > 500000 Bit \tag{5.3}$$

Dabei beschreibt *30*256* die Nachrichtenanzahl der Objektinformationen bei 255 Objekterkennungen und *2* die der übrigen Nachrichten, den Radarstatus und die Version des Radars, welche einmal pro Sekunde versendet werden. In Summe macht das 852702 Bits, welche in einer Sekunde über den CAN verschickt werden müssen. Dies ist fast doppelt so viel, wie ein Bus dieser Bitrate übertragen kann. Das zeigt, dass das Radar somit entweder weniger als 255 Objekte erkennen kann oder die 33 ms zum Senden der Informationen nicht eingehalten werden können. Durch einen Versuch wird daher getestet, eine maximale Anzahl der Erkennungen des Radars zu erreichen, indem das Radar nahe an eine Metallplatte gelegt wird und somit alle ausgesendeten, elektromagnetischen Wellen direkt wieder zurück reflektiert werden. Dabei ergibt sich ein maximaler Wert von 63. Keine eigens durchgeführte Messung mit den Realdaten des Radars lieferte einen größeren Wert als 63. Eventuell ist dies der maximale Wert an Objekterkennungen, welcher das Radar tatsächlich unterstützt. Dabei würde der Bus auch nicht überlastet werden, was an der Tabelle 5.3 ersichtlich ist und die 33 ms zwischen den Objektdetektionen werden eingehalten.

Die Prozessorauslastung hingegen steigt bis zu einer Objektanzahl von 50 an, geht anschließend aber wieder zurück. Dieses Ergebnis ist trotz mehrerer Messungen regelmäßig erreicht worden.

Ein Betrieb von mehreren Radaren in einem CAN-Netzwerk wurde hier im Allgemeinen nicht explizit betrachtet. Dies kann jedoch abgeschätzt und berechnet werden. Zu beachten wären dann allerdings noch die Nachrichten, welche aufgrund der Busarbitrierung zunächst nicht versendet werden und anschließend das Radar einen neuen Sendeversuch jener starten muss. Ob dies durch das Radar geschieht, wurde ebenfalls nicht überprüft. Somit sind damit alle Tests abgeschlossen und die praktische Arbeit hiermit fertiggestellt.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung der Ergebnisse

Um die Ergebnisse der vorherigen drei Kapitel nochmals zusammenzufassen:
Das Radar stellt Informationen zu Objekten, sowie Statusnachrichten bereit. Die Objektinformationen des Radars können dabei alle in OSI abgebildet werden, bis auf die dynamischen Eigenschaften des Objektes und das verwendete CAN-Interface. Es ist nicht möglich, die Statusnachrichten, sowie die Funktion der Kollisionserkennung durch OSI-Nachrichten darzustellen. In OSI können bei den Objekten die wichtigsten Attribute, wie Entfernung, Geschwindigkeit und ID gesetzt werden, manche Werte, wie das Signal-Rausch-Verhältnis oder die Werte des mittleren quadratischen Fehlers jedoch, werden vom Radar nicht bereitgestellt.
Durch die Implementierung war es möglich, diese Zuordnung programmiertechnisch umzusetzen. Dafür wurde zunächst eine CAN-Klasse zum Senden und Empfangen der Nachrichten entwickelt, anschließend daran noch eine Konvertierungsklasse, welche die empfangenen CAN-Frames auswertete und abspeicherte. Diese stellte ebenfalls eine Funktion zur Verfügung, welche die aktuellen Objektdaten in eine *RadarDetectionData*-Nachricht umwandelte und diese im Anschluss mittels Protobuf serialisierte. Zuletzt wurde noch ein ROS2-Knoten implementiert, welcher ununterbrochen bis zum Herunterfahren oder Stoppen von ROS2 die eben beschriebene Funktion zum Umwandeln aufrief und auf neue Objekterkennungen seitens des Radars wartete. Eine praktische Umsetzung eines Subscribers-Knoten wurde nicht mehr durchgeführt, da das Radar keine Informationen über den Fahrzeugzustand erhalten kann und er somit zunächst nicht benötigt wurde.
Zuletzt noch wurde die gesamte Implementierung getestet. Für die funktionalen Tests diente die Bibliothek *gtest*. Es wurden zuerst die aus der CAN-Datenbank generierten Dateien getestet, ob die Datenbank korrekt war und die Nachrichten richtig umwandelte. Dafür wurden jeweils verschiedene CAN-Frames erzeugt und dann mittels der entsprechenden Funktionen verpackt, entpackt und dekodiert. Die Werte wurden jeweils so gewählt, dass sich die minimalen und maximalen Werte verifizieren lassen. Dabei wurden noch einige Fehler, vor allem in Hinblick auf die verwendeten Vorzeichen, in der Datenbank erkannt. Anschließend wurde die CAN-Klasse getestet. Besonders wichtig war es zu sehen, ob das Setzen der Filter, der Erhalt eines korrekten Zeitstempels und das Senden sowie Empfangen möglich waren und fehlerfrei abliefen. Daraufhin wurde die Konvertierungsklasse getestet. Bedeutend dabei war, ob die richtigen Daten vom CAN

korrekt in OSI umgewandelt wurden und alle Funktionen ordnungsgemäß funktionierten. Schließlich fand eine Überprüfung des ROS2-Knotens statt, ob dieser die Daten korrekt sendete.

Abschließend wurden noch nichtfunktionale Tests ausgeführt, um festzustellen, ob es Speicherprobleme oder Sockets gab, welche nicht geschlossen wurden und welche Performance die Implementierung in Abhängigkeit von der Anzahl an Objekterkennungen aufweist. Hierbei konnte gezeigt werden, dass keine Probleme existierten und dass die Software korrekt arbeitete.

6.2 Fazit

Wie im vorherigen Kapitel bereits zusammengefasst, ist es also generell möglich, reale Sensordaten eines Short-Range Radars in OSI darzustellen. Dabei gelang das nicht bei Konfigurationsnachrichten oder anderen Funktionen, wie einer Kollisionserkennung. Eine Abbildung hierfür ist in OSI noch nicht möglich. Die Objekte hingegen ließen sich, bis auf deren dynamischen Eigenschaften, problemlos im OSI-Format darstellen. Jedoch konnten dabei nicht alle in OSI verfügbaren Felder, wie zum Beispiel die Fehlerabweichungen oder auch andere Werte wie das Signal-Rausch-Verhältnis, gesetzt werden. Auch Wahrscheinlichkeiten einer Erkennung oder bestimmte Kennzeichner konnten nicht aus den Radardaten abgeleitet werden.

Für einen Normalbetrieb des Radars, bei welchem nur die Entfernungen, Geschwindigkeiten und Winkel sowie RCS-Werte und IDs zur Unterscheidung der Objekte relevant sind, ist es jedoch möglich, alle Daten in OSI darzustellen.

Die Annahme, welche zu Beginn der Arbeit getroffen wurde, dass es nicht möglich sei, alle Radardaten in OSI darzustellen, konnte somit bestätigt werden.

6.3 Reflexion

Die Bachelorarbeit hat zur Sammlung vieler neuer praxisbezogener Erfahrungen geführt. Aufgrund des erstmaligen Arbeitens mit einem Radar konnte durch diese praktische Vorgehensweise noch einmal ein ganz anderer Zugang - weit über die theoretische Funktionsweise und den Verwendungszweck hinaus - zu diesem Thema erlangt werden.

Auch das praktische Arbeiten mit einem CAN-Netzwerk war sehr lehrreich. Es wurde zwar in Praktika bereits kurz angeschnitten, jedoch hatte ich zuvor noch nie die Möglichkeit, reelle Daten eines Sensors auszulesen, auszuwerten und damit zu arbeiten.

Ebenfalls habe ich viel durch die Verwendung des Open Simulation Interfaces über den Nutzen von gemeinsamen Softwareschnittstellen erfahren. Auch durch die Verwendung von ROS2 konnte bereits vorhandenes Wissen über ROS Version 1 nochmals erweitert werden. Die genauen Konzepte und Abläufe in ROS wurden durch diese Arbeit nochmals vertieft.

Durch die Implementierung konnte viel Programmiererfahrung gesammelt werden. Vor allem die Tatsache, dass ein größeres Projekt von mir noch nie in C++ geschrieben wurde, war letztendlich eine große herausfordernde Erfahrung und bereitete mir sehr viel Spaß. Durch diese Arbeit wurde ebenfalls eine Vorliebe für das Linux Betriebssystem entwickelt, da es auf Windows auf sehr viele Arten schwieriger gewesen wäre, diese Arbeit zu erstellen.

Zusammenfassend würde ich diese Arbeit als sehr instruktiv betrachten, da nicht nur plausible und wichtige Forschungsergebnisse dabei herauskamen, sondern ich dabei auch noch viel zu den verschiedenen Themen gelernt und mitgenommen habe.

6.4 Nachwort und Ausblick

Zuletzt wird noch kurz angesprochen, was in der Arbeit nicht behandelt wurde, beziehungsweise was es anschließend an diese Abschlussarbeit auszuführen gilt.

Zunächst wurde ja bereits angesprochen, dass es in OSI keine Konfigurationsmöglichkeiten für reale Sensoren gibt. Diese wären zum Parametrieren wichtig. Außerdem fehlen noch bestimmte Funktionen, wie eine Möglichkeit, eine Kollisionserkennung in OSI darzustellen. Hierfür müssen bereits bestehende Nachrichtentypen erweitert werden.

Des Weiteren wurden in dieser Arbeit die Daten des Radars verwendet, ohne tatsächlich zu überprüfen, ob die Realdaten des Radars korrekt sind. Denn nur dann kann auch eine korrekte Funktionsweise der Software gewährleistet werden.

Außerdem muss sichergestellt werden, ob die Radialgeschwindigkeit in OSI tatsächlich relativ zum Radar angegeben wird. Wenn das nicht der Fall ist, muss an dieser Stelle noch ein Subscriber-Knoten implementiert werden, welcher dann die aktuelle, absolute Geschwindigkeit des Fahrzeugs dem Publisher-Knoten bereitstellt, damit dieser mithilfe der relativen Angaben des Radars eine absolute Geschwindigkeit der Objekterkennungen errechnen kann.

Hier wäre eine bessere Dokumentation der *RadarDetection*-Nachricht hilfreich gewesen.

Abschließend kann nur noch genannt werden, dass die Abbildung weiterer Sensoren in OSI erforscht werden sollte, um in naher Zukunft ein autonomes Fahrzeug entwickeln zu können, welches auf eine ähnliche Architektur mit dem standardisiertem Format setzt und somit alle Daten untereinander ausgetauscht werden können. Dann ist es nur noch notwendig, einen auf *Machine Learning* basierten Algorithmus zu entwickeln, um ein Fahrzeug schließlich komplett autonom im Straßenverkehr fahren lassen zu können.

Literatur

[1] O. M. G. (OMG®), *Interface Definition Language*, Zuletzt zugegriffen am 10.02.2023 um 22:00 Uhr. Adresse: `https://www.omg.org/spec/IDL`.

[2] O. Abdulaaty, G. Schroeder, A. Hussein, F. Albers und T. Bertram, “Real-time Depth Completion using Radar and Camera”, in *2022 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*, 2022, S. 1–6. DOI: `10.1109/ICVES56941.2022.9986598`.

[3] D. R. Adler, *Autonom oder vielleicht doch nur hochautomatisiert?*, Zuletzt zugegriffen am 27.01.2023 um 18:01 Uhr, Sep. 2019. Adresse: `https://www.iese.fraunhofer.de/blog/autonom-oder-vielleicht-doch-nur-hochautomatisiert-was-ist-eigentlich-der-unterschied/`.

[4] C. AG, *Mobilitätsstudie 2022*, Zuletzt zugegriffen am 27.01.2023 um 18:01 Uhr, 2022. Adresse: `https://cdn.continental.com/fileadmin/__imported/sites/corporate/_international/german/hubpages/10_20presse/studien_und_publikationen/mobiliteatsstudien/2022/20220630-continental-mobilitaetsstudie_2022-ergebnisse-de.pdf`.

[5] Aral, *Aral Studie. Trends beim Autokauf 2021*, Zuletzt zugegriffen am 27.01.2023 um 17:40 Uhr, 2021. Adresse: `https://www.aral.de/content/dam/aral/business-sites/de/global/retail/presse/pressemeldungen/2021/broschuere-aral-studie-trends-beim-autokauf.pdf`.

[6] ASAM, *ASAM OSI® (Open Simulation Interface)*, Zuletzt zugegriffen am 09.02.2023 um 20:00 Uhr, Feb. 2023. Adresse: `https://opensimulationinterface.github.io/osi-documentation/`.

[7] R. Bartolozzi, V. Landersheim, G. Stoll, H. Holzmann, R. Möller und H. Atzrodt, “Vehicle simulation model chain for virtual testing of automated driving functions and systems”, in *2022 IEEE Intelligent Vehicles Symposium (IV)*, 2022, S. 1054–1059. DOI: `10.1109/IV51971.2022.9827074`.

[8] D. Caivano, M. De Vincentiis, F. Nitti und A. Pal, “Quantum Optimization for Fast CAN Bus Intrusion Detection”, in *Proceedings of the 1st International Workshop on Quantum Programming for Software Engineering*, Ser. QP4SE 2022, Singapore, Singapore: Association for Computing Machinery, 2022, S. 15–18, ISBN: 9781450394581. DOI: `10.1145/3549036.3562058`. Adresse: `https://doi-org.thi.idm.oclc.org/10.1145/3549036.3562058`.

[9] *canbusload - manual page for canbusload 2018.02.0-1*, Zuletzt zugegriffen am 12.02.2023 um 12:24 Uhr. Adresse: `https://manpages.ubuntu.com/manpages/bionic/man1/canbusload.1.html`.

[10] *cantools*, Zuletzt zugegriffen am 10.02.2023 um 23:40 Uhr. Adresse: `https://github.com/cantools/cantools`.

[11] *Challenges for cameras in automotive applications*, Zuletzt zugegriffen am 01.12.2022 um 22:35 Uhr, Feb. 2022. Adresse: `https://www.image-engineering.de/library/blog/articles/1157-challenges-for-cameras-in-automotive-applications`.

[12] *Collaboration diagram for osi3::RadarDetection*, Zuletzt zugegriffen am 06.02.2023 um 20:11 Uhr. Adresse: `https://opensimulationinterface.github.io/open-simulation-interface/structosi3_1_1RadarDetectionData__coll__graph.png`.

[13] W. Commons, *File:NetzwerkTopologien.png — Wikimedia Commons, the free media repository*, Zuletzt zugegriffen am 31.01.2023 um 13:26 Uhr, 2020. Adresse: `https://commons.wikimedia.org/w/index.php?title=File:NetzwerkTopologien.png&oldid=458729865`.

[14] P. David Cen Cheng, M. Indri, F. Sibona, M. De Rose und G. Prato, “Dynamic Path Planning of a mobile robot adopting a costmap layer approach in ROS2”, in *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2022, S. 1–8. DOI: `10.1109/ETFA52439.2022.9921458`.

[15] J. Detlefsen, *Radartechnik*. Springer Berlin Heidelberg, 1989. DOI: `10.1007/978-3-642-83600-8`. Adresse: `https://doi.org/10.1007/978-3-642-83600-8`.

[16] E. W. Dijkstra, “The Humble Programmer”, in *ACM Turing Award Lectures*. New York, NY, USA: Association for Computing Machinery, Okt. 1972, ISBN: 9781450310499. Adresse: `https://doi-org.thi.idm.oclc.org/10.1145/1283920.1283927`.

[17] A. e.V., *About ASAM*, Zuletzt zugegriffen am 06.02.2023 um 14:15 Uhr. Adresse: `https://www.asam.net/about-asam/our-vision/`.

[18] A. e.V., *The History of ASAM*, Zuletzt zugegriffen am 06.02.2023 um 14:15 Uhr. Adresse: `https://www.asam.net/about-asam/history/`.

[19] J. Fenlason und R. Stallman, *GNU gprof - The GNU Profiler*, Zuletzt zugegriffen am 12.02.2023 um 12:09 Uhr. Adresse: `https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html`.

[20] *Funktionale Tests*, Zuletzt zugegriffen am 24.01.2023 um 16:31 Uhr. Adresse: `https://ditcraft.io/de/services/functional-testing.html`.

[21] D.-I. B. Giesler und D.-I. M. Reichel, *Roboterarchitekturen im Fahrzeug*, Zuletzt zugegriffen am 27.01.2023 um 18:25 Uhr, Mai 2016. Adresse: `https://www.all-electronics.de/automotive-transportation/roboterarchitekturen-im-fahrzeug.html`.

[22] V. I. GmbH, *Firmengeschichte*, Zuletzt zugegriffen am 08.02.2023 um 17:00 Uhr. Adresse: `https://www.vector.com/de/de/unternehmen/ueber-vector/firmengeschichte/`.

[23] Google, *GoogleTest*, Zuletzt zugegriffen am 12.02.2023 um 12:09 Uhr. Adresse: `https://github.com/google/googletest`.

[24] J. Hertzberg, K. Lingemann und A. Nüchter, *Mobile Roboter*. Springer Berlin Heidelberg, 2012. DOI: `10.1007/978-3-642-01726-1`. Adresse: `https://doi.org/10.1007/978-3-642-01726-1`.

[25] informatik-verstehen, *Testverfahren*, Zuletzt zugegriffen am 31.01.2023 um 15:18 Uhr. Adresse: `https://www.informatik-verstehen.de/lexikon/testverfahren/`.

[26] Intel, *Intel® Core™ Prozessor i5-6300U*, Zuletzt zugegriffen am 26.01.2023 um 17:11 Uhr. Adresse: `https://www.intel.de/content/www/de/de/products/sku/88190/intel-core-i56300u-processor-3m-cache-up-to-3-00-ghz/specifications.html`.

[27] P. D. C. Johner, *ISO 25010 und ISO 9126*, Zuletzt zugegriffen am 26.01.2023 um 17:11 Uhr. Adresse: `https://www.johner-institut.de/blog/iec-62304-%20medizinische-software/iso-9126-und-iso-25010/`.

[28] L. Joseph und A. Johny, *Robot Operating System (ROS) for Absolute Beginners*. Apress, 2022. DOI: `10.1007/978-1-4842-7750-8`. Adresse: `https://doi.org/10.1007/978-1-4842-7750-8`.

[29] L. Kang und H. Shen, "Detection and Mitigation of Sensor and CAN Bus Attacks in Vehicle Anti-Lock Braking Systems", *ACM Trans. Cyber-Phys. Syst.*, Jg. 6, Nr. 1, Jan. 2022, ISSN: 2378-962X. DOI: `10.1145/3495534`. Adresse: `https://doi-org.thi.idm.oclc.org/10.1145/3495534`.

[30] *Kayak*, Zuletzt zugegriffen am 13.02.2023 um 18:17 Uhr. Adresse: `https://github.com/dschanoeh/Kayak/`.

[31] *kcd*, Zuletzt zugegriffen am 08.02.2023 um 17:10 Uhr. Adresse: `https://github.com/julietkilo/kcd`.

[32] T. L. Kernel, *SocketCAN - Controller Area Network*, Zuletzt zugegriffen am 11.02.2023 um 00:40 Uhr. Adresse: `https://www.kernel.org/doc/html/latest/networking/can.html`.

[33] G. Knuepffer, *Asam entwickelt OSI-Format-Spezifikation von BMW weiter*, Zuletzt zugegriffen am 12.12.2022 um 19:28 Uhr, Okt. 2019. Adresse: `https://www.all-electronics.de/automotive-transportation/asam-entwickelt-osi-format-spezifikation-von-bmw-weiter.html`.

[34] R. Kumar und S. Jayashankar, "Radar and Camera Sensor Fusion with ROS for Autonomous Driving", in *2019 Fifth International Conference on Image Information Processing (ICIIP)*, 2019, S. 568–573. DOI: `10.1109/ICIIP47207.2019.8985782`.

[35] H. Li, T. Kanuric und A. Eichberger, "Automotive Radar Modeling for Virtual Simulation Based on Mixture Density Network", *IEEE Sensors Journal*, S. 1–1, 2022. DOI: `10.1109/JSEN.2022.3223765`.

[36] Y. Liao, *Roboter-Betriebssysteme: ROS2 bügelt Schwächen aus*, `https://safe-intelligence.fraunhofer.de/artikel/autonome-systeme-ros2`, Zuletzt zugegriffen am 10.12.2022 um 22:40 Uhr, Juli 2020. Adresse: `https://safe-intelligence.fraunhofer.de/artikel/autonome-systeme-ros2`.

[37] R. Liu, Z. Guan, B. Li, G. Wen und B. Liu, "Research on real-time positioning and map construction technology of intelligent car based on ROS", in *2022 IEEE International Conference on Mechatronics and Automation (ICMA)*, 2022, S. 1587–1592. DOI: `10.1109/ICMA54519.2022.9856339`.

[38] MaikolDrechsler, *Update host vehicle data with kinematics data #679*, Zuletzt zugegriffen am 10.02.2023 um 23:20 Uhr. Adresse: `https://github.com/OpenSimulationInterface/open-simulation-interface/pull/679`.

[39] N. Marko, J. Ruebsam, A. Biehn und H. Schneider, "Scenario-based Testing of ADAS - Integration of the Open Simulation Interface into Co-simulation for Function Validation", in *Proceedings of the 9th International Conference on Simulation and Modeling Methodologies, Technologies and Applications - Volume 1: SIMULTECH,*, INSTICC, SciTePress, 2019, S. 255–262, ISBN: 978-989-758-381-0. DOI: `10.5220/0007838302550262`.

[40] S. Mones, *Autonomes Fahren: Robo-Autos dürfen bis zu 130 km/h fahren*, Zuletzt zugegriffen am 27.01.2023 um 18:10 Uhr, Jan. 2023. Adresse: `https://www.kreiszeitung.de/leben/auto/verordnung-autonomes-fahren-robo-autos-130-kmh-kraftfahrtbundesamt-mercedes-bmw-un-zr-92015525.html`.

[41] S. Muckenhuber, H. Holzer, J. Rübsam und G. Stettinger, "Object-based sensor model for virtual testing of ADAS/AD functions", in *2019 IEEE International Conference on Connected Vehicles and Expo (ICCVE)*, 2019, S. 1–6. DOI: `10.1109/ICCVE45908.2019.8965071`.

[42] D. St-Onge und D. Herath, "The Robot Operating System (ROS1 &2): Programming Paradigms and Deployment", in *Foundations of Robotics: A Multidisciplinary Approach with Python and ROS*, D. Herath und D. St-Onge, Hrsg. Singapore: Springer Nature Singapore, 2022, S. 105–126, ISBN: 978-981-19-1983-1. DOI: `10.1007/978-981-19-1983-1_5`. Adresse: `https://doi.org/10.1007/978-981-19-1983-1_5`.

[43] *osi_featuredata.proto*, Zuletzt zugegriffen am 08.02.2023 um 12:52 Uhr. Adresse: `https://github.com/OpenSimulationInterface/open-simulation-interface/blob/master/osi_featuredata.proto`.

[44] *osi3::RadarDetectionData Struct Reference*, Zuletzt zugegriffen am 06.02.2023 um 20:11 Uhr. Adresse: `https://opensimulationinterface.github.io/open-simulation-interface/structosi3_1_1RadarDetection.html`.

[45] *pidstat(1) — Linux manual page*, Zuletzt zugegriffen am 12.02.2023 um 12:19 Uhr. Adresse: `https://man7.org/linux/man-pages/man1/pidstat.1.html`.

[46] M. Reidel, *Welche Vorteile bietet autonomes Fahren? (Studie)*, durchgeführt an der DHMW Ravensburg, veröffentlicht im Horizont am 11.10.2018 in der Ausgabe 41/2018 auf Seite 69, visualisiert durch Statista, Jan. 2017. Adresse: `https://de.statista.com/statistik/daten/studie/270606/umfrage/vorteile-von-autonomen-fahrzeugen/`.

[47] O. Robotics, *ROS 2 Documentation*, Zuletzt zugegriffen am 12.02.2023 um 12:09 Uhr, 2023. Adresse: `https://docs.ros.org/en/humble/index.html`.

[48] *ROS2: Was ändert sich gegenüber ROS?*, `https://www.generationrobots.com/blog/de/ros2-was-andert-sich-gegenuber-ros/`, Zuletzt zugegriffen am 10.12.2022 um 23:20 Uhr. Adresse: `https://www.generationrobots.com/blog/de/ros2-was-andert-sich-gegenuber-ros/`.

[49] C. M. Schwarzer, *Hochautomatisiertes Fahren: Level 3 bis 130 km/h zuerst bei Mercedes*, Zuletzt zugegriffen am 27.01.2023 um 18:05 Uhr, Juli 2022. Adresse: `https://www.heise.de/hintergrund/Hochautomatisiertes-Fahren-Level-3-bis-130-km-h-zuerst-bei-Mercedes-7162660.html`.

[50] *SocketCAN userspace utilities and tools*, Zuletzt zugegriffen am 10.02.2023 um 23:35 Uhr. Adresse: `https://github.com/linux-can/can-utils`.

[51] *SR73 77GHz Millimeter Wave Radar Communication Protocol (Titel übersetzt aus dem chinesischen)*, 1.4, Hunan Nanoradar Science und Technology Co., Ltd., Nov. 2021.

[52] *SR73 Millimeter Wave Radar Communication Protocol*, 1.0, Zuletzt zugegriffen am 12.02.2022 um 11:50 Uhr, Hunan Nanoradar Science und Technology Co., Ltd., Apr. 2019. Adresse: `http://en.nanoradar.cn/File/view/id/494.html`.

[53] *SR73 Millimeter Wave Radar Communication Protocol V1.3*, 1.4, Hunan Nanoradar Science und Technology Co., Ltd., März 2020.

[54] G. Stavrinos, "ROS2 For ROS1 Users", in *Robot Operating System (ROS): The Complete Reference (Volume 5)*, A. Koubaa, Hrsg. Cham: Springer International Publishing, 2021, S. 31–42, ISBN: 978-3-030-45956-7. DOI: `10.1007/978-3-030-45956-7_2`. Adresse: `https://doi.org/10.1007/978-3-030-45956-7_2`.

[55] H. Teper, M. Günzel, N. Ueter, G. von der Brüggen und J.-J. Chen, "End-To-End Timing Analysis in ROS2", in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022, S. 53–65. DOI: `10.1109/RTSS55097.2022.00015`.

[56] *Ultraschallsensor*, Zuletzt zugegriffen am 06.02.2023 um 13:50 Uhr. Adresse: `https://www.mein-autolexikon.de/fahrerassistenzsysteme/ultraschallsensor.html`.

[57] *Valgrind*, Zuletzt zugegriffen am 12.02.2023 um 12:15 Uhr. Adresse: `https://valgrind.org/`.

[58] *Vor- und Nachteile der linearen Bustopologie*, Zuletzt zugegriffen am 04.12.2022 um 20:56 Uhr. Adresse: `https://kfz-aufgaben.de/can/Vor-und-Nachteile-der-lineare.442.0.html`.

[59] R. Yadav, P. K. Dahiya und R. Mishra, "Comparative analysis of automotive radar sensor for collision detection and warning system", *International Journal of Information Technology*, Jg. 12, Nr. 1, S. 289–294, März 2020, ISSN: 2511-2112. DOI: `10.1007/s41870-018-0167-3`. Adresse: `https://doi.org/10.1007/s41870-018-0167-3`.

[60] W. Zimmermann und R. Schmidgall, *Bussysteme in der Fahrzeugtechnik*. Springer Fachmedien Wiesbaden, 2014. DOI: `10.1007/978-3-658-02419-2`. Adresse: `https://doi.org/10.1007/978-3-658-02419-2`.

Anhang - Konvertierungsfunktion

```
// Converter function for the CAN data to OSI message
int SR73OSI::convert_can_to_osi(std::string* buffer)
{
    // Time messured: ros2 launch sr73 sr73_launch.xml |
    //                grep Conversation | grep -o '........$' > converter_funktion_durations.txt
    // Average calculated: 26947.8717 ns
    bool messure = false;
    if(messure) {
        clock_gettime(CLOCK_REALTIME, &start); // Start time measurement
    }

    // check if data is received correctly
    // check version information
    // if one version is received, check if all version-parts are correct
    if(version_information.major_release) {
        // If any version-part is not correct, print an error message and return -2
        if(version_information.major_release != expected_version_information.major_release ||
           version_information.minor_release != expected_version_information.minor_release ||
           version_information.patch_level != expected_version_information.patch_level) {
            RCLCPP_ERROR(rclcpp::get_logger(__FILENAME__),
                    "Wrong version information received: Expected: %d.%d.%d, Received: %d.%d.%d",
                        expected_version_information.major_release,
                        expected_version_information.minor_release,
                        expected_version_information.patch_level,
                        version_information.major_release, version_information.minor_release,
                        version_information.patch_level);
            return -2; // when version information is not correct
        }
    }
    // check radar status
    // if one radar status is received, check if the current radar status is correct
    if(sr73_radar_statue_sensor_id_is_in_range(radar_status.sensor_id)) {
        // if the sensorID or sortIndex is not correct, print an error message and return -3
        if(radar_status.sensor_id != sensorID || radar_status.sort_index != sortIndex) {
            RCLCPP_ERROR(rclcpp::get_logger(__FILENAME__),"Wrong radar status received");
            return -3; // when sensor ID is not correct
        }
    }

    // critical section -> save frames locally ----------------------------
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, [this]{return ready;}); // Wait until the data is ready
    tlocal = tcurr; // save the timestamp of the newest object list status frame
    local_object_list_status = current_object_list_status; // save the object list status
    // copy the object general information from the heap to the local variables here on the stack
    std::memcpy(local_objects_general_information, //dest
        current_objects_general_information, //src
        sizeof(struct sr73_objects_general_information_t)*
                local_object_list_status.objects_nof_objects //size
    );
```

```
    ready = false;
    // notify the "CAN"-Thread that the data is saved, so it can continue saving new frames
    cv.notify_one();
    // end of critical section ----------------------------

    if(messure)clock_gettime(CLOCK_REALTIME, &after_mutex);

    // create the OSI message
    osi3::RadarDetectionData radarDetectionData;
    osi3::SensorDetectionHeader* header = radarDetectionData.mutable_header();

    // local variables for distance and velocity calculation
    double distLong = 0.0, distLat = 0.0;
    double vRelLong = 0.0, vRelLat = 0.0;
    double objectAngle = 0.0;

    // HEADER: Set all header fields
    // num of objects
    header->set_number_of_valid_detections(local_object_list_status.objects_nof_objects);
    // cycle counter
    header->set_cycle_counter(local_object_list_status.objects_meas_count);
    // measurement timestamp (can) of object list status message
    header->mutable_measurement_time()->set_nanos(tlocal.tv_usec * 1000);
    header->mutable_measurement_time()->set_seconds(tlocal.tv_sec);
    // Data qualifier
    header->set_data_qualifier(osi3::SensorDetectionHeader_DataQualifier_DATA_QUALIFIER_AVAILABLE);
    // mount pos
    header->mutable_mounting_position()->mutable_position()->set_x(mount_pos.pos_x);
    header->mutable_mounting_position()->mutable_position()->set_y(mount_pos.pos_y);
    header->mutable_mounting_position()->mutable_position()->set_z(mount_pos.pos_z);
    header->mutable_mounting_position()->mutable_orientation()->set_roll(mount_pos.ori_roll);
    header->mutable_mounting_position()->mutable_orientation()->set_pitch(mount_pos.ori_pitch);
    header->mutable_mounting_position()->mutable_orientation()->set_yaw(mount_pos.ori_yaw);
    // sensor id
    header->mutable_sensor_id()->set_value(sensorID);

    if(messure)clock_gettime(CLOCK_REALTIME, &after_header);

    // DETECTIONS: Set for every detection its fields
    for(i = 0; i < local_object_list_status.objects_nof_objects; i++)
    {
        // add a new detection to the RadarDetectionData message
        osi3::RadarDetection* detection = radarDetectionData.add_detection();
        // identifier
        detection->mutable_object_id()->set_value(local_objects_general_information[i].objects_id +
                (1<<10) * sensorID); // OSI ID = RadarDetectionID + 1024 * sensorID (0-7)
        // classification
        // Dyn. Props always OTHER (not supported by SR73 -> always 0)
        detection->set_classification(osi3::DETECTION_CLASSIFICATION_OTHER);
        // rcs
        detection->set_rcs(sr73_objects_general_information_objects_rcs_decode(
                local_objects_general_information[i].objects_rcs));
        // distance + azimuth
        distLong = sr73_objects_general_information_objects_dist_long_decode(
                local_objects_general_information[i].objects_dist_long);
        distLat = sr73_objects_general_information_objects_dist_lat_decode(
                local_objects_general_information[i].objects_dist_lat);
        // pythagoras to get distance
        detection->mutable_position()->set_distance(sqrt(distLong*distLong + distLat*distLat));
        detection->mutable_position()->set_elevation(0.0); // 2D -> elevation = 0
        // angle in radian. alpha: between x-axis (DistLat) and object
```

```
        detection->mutable_position()->set_azimuth(atan2(distLong, distLat));
        // velocity absolute
        objectAngle = atan2(distLat,distLong); // angle in radian. gamma
        vRelLong = sr73_objects_general_information_objects_vrel_long_decode(
                local_objects_general_information[i].objects_vrel_long);
        vRelLat = sr73_objects_general_information_objects_vrel_lat_decode(
                local_objects_general_information[i].objects_vrel_lat);
        detection->set_radial_velocity((vRelLong*cos(objectAngle) + vRelLat*sin(objectAngle))
                * (-1) ); // sign changes (radar: speed negative towards the sensor, osi: positive)
    }
    if(messure)clock_gettime(CLOCK_REALTIME, &after_detections);

    // check if number of objects is correct
    if(local_object_list_status.objects_nof_objects != radarDetectionData.detection_size()) {
        // if something went wrong while adding the detections, return -1 and print an error message
        RCLCPP_ERROR(rclcpp::get_logger(__FILENAME__),
                "wrong number of detections detected in OSI");
        return -1;
    }

    // serialize the message and write it to the buffer
    radarDetectionData.SerializeToString(buffer);

    if(messure) { // print the messured time differences
        clock_gettime(CLOCK_REALTIME, &end);
        if(end.tv_nsec - start.tv_nsec > 0) RCLCPP_INFO(rclcpp::get_logger(__FILENAME__),
                "Conversation took %ld ns", end.tv_nsec - start.tv_nsec);
        else RCLCPP_INFO(rclcpp::get_logger(__FILENAME__),
                "Conversation took %ld ns", 1000000000 + end.tv_nsec - start.tv_nsec);
        if(end.tv_nsec - after_mutex.tv_nsec > 0) RCLCPP_INFO(rclcpp::get_logger(__FILENAME__),
                "Since Mutex unlock took %ld ns", end.tv_nsec - after_mutex.tv_nsec);
        else RCLCPP_INFO(rclcpp::get_logger(__FILENAME__),
                "Since Mutex unlock took %ld ns", 1000000000 + end.tv_nsec - after_mutex.tv_nsec);
        if(after_header.tv_nsec - after_mutex.tv_nsec > 0)
                RCLCPP_INFO(rclcpp::get_logger(__FILENAME__),
                        "Header fill took %ld ns", after_header.tv_nsec - after_mutex.tv_nsec);
        else RCLCPP_INFO(rclcpp::get_logger(__FILENAME__),
                "Header fill took %ld ns", 1000000000 + after_header.tv_nsec -
                        after_mutex.tv_nsec);
        if(after_detections.tv_nsec - after_header.tv_nsec > 0)
                RCLCPP_INFO(rclcpp::get_logger(__FILENAME__),
                        "Detections fill took %ld ns", after_detections.tv_nsec -
                                after_header.tv_nsec);
        else RCLCPP_INFO(rclcpp::get_logger(__FILENAME__),
                "Detections fill took %ld ns", 1000000000 + after_detections.tv_nsec -
                        after_header.tv_nsec);
        if(after_detections.tv_nsec - after_mutex.tv_nsec > 0)
                RCLCPP_INFO(rclcpp::get_logger(__FILENAME__),
                        "Fill took %ld ns", after_detections.tv_nsec - after_mutex.tv_nsec);
        else RCLCPP_INFO(rclcpp::get_logger(__FILENAME__),
                "Fill took %ld ns", 1000000000 + after_detections.tv_nsec - after_mutex.tv_nsec);
        if(end.tv_nsec - after_detections.tv_nsec > 0)
                RCLCPP_INFO(rclcpp::get_logger(__FILENAME__),
                        "Serialize took %ld ns", end.tv_nsec - after_detections.tv_nsec);
        else RCLCPP_INFO(rclcpp::get_logger(__FILENAME__),
                "Serialize took %ld ns", 1000000000 + end.tv_nsec - after_detections.tv_nsec);
    }

    return 0; // return 0 if everything went fine
}
```