technische universität
dortmund

Diplomarbeit

Exploring the Performance of Hardware Message Filtering in Controller Area Network

Iryna Denysenko

08. April 2019

Gutachter:
Prof. Dr. Jian-Jia Chen
M.Sc. Lea Schönberger

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 12
Design Automation for Embedded Systems Group

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Ziele der Arbeit 3

1.2 Aufbau der Arbeit 5

2 Grundlagen **7**

2.1 Controller Area Network (CAN) 7

2.1.1 Grundlegende Eigenschaften 8

2.1.2 Aufbau einer CAN-Nachricht 10

2.1.3 Lowspeed- und Highspeed-CAN 12

2.2 Hardware-Nachrichtenfilter 13

2.2.1 Funktionsweise 15

2.2.2 Filterkonfigurationen 15

2.3 Integer Linear Programming (ILP) 16

2.3.1 Geschichte der Entstehung und Entwicklung 17

2.3.2 Grundlegende Definitionen 18

2.3.3 ILP-Beispiel 20

2.3.4 Big-M-Methode 22

3 Design der Filterkonfiguration **25**

3.1 Problembeschreibung 25

3.2 Formulierung des Problems als Heuristik 26

3.3 Beispiel 27

4 Design der Nachrichten **29**

4.1 Problembeschreibung 29

4.2 Formulierung des Problems als ILP 29

5 Implementierung **31**

5.1 Verwendete Werkzeuge (Tools) 31

5.1.1 Programmiersprache Python 31

5.1.2 Installierte Bibliotheken und Packages 33

5.1.3 Entwicklungsumgebung 34
5.1.4 Verwendete Solver: Gurobi 36
5.2 Implementierungsdetails zum Design der Filterkonfiguration 38
5.3 Implementierungsdetails zum Design der Nachrichten 42

6 Evaluation **47**
6.1 Design der Filterkonfiguration 47
6.1.1 Experimenteller Aufbau 48
6.1.2 Ergebnisse 50
6.1.3 Diskussion und Schlussfolgerungen 56
6.2 Design der Nachrichten 56
6.2.1 Experimenteller Aufbau 56
6.2.2 Ergebnisse 57
6.2.3 Diskussion und Schlussfolgerungen 58

7 Fazit und Ausblick **59**

Abbildungsverzeichnis **61**

Tabellenverzeichnis **63**

Literaturverzeichnis **69**

Eidesstattliche Versicherung **72**

Kapitel 1

Einleitung

Heutzutage integrieren sich elektronische Technologien immer mehr in den Alltag. Angefangen bei einer Smartwatch, der Begleitung durch die Wohnung mittels einer Sprachsteuerung, dem Verlassen von intelligenten Gebäuden, über Fahrkartenautomaten mit einer smarten Fahrplananzeige bis hin zu Echtzeitnavigationssystemen. Im Büro wird der moderne Berufstätige von Digitalisierung, Big Data und Industrie 4.0 erwartet, und die ganze Zeit lassen sich die Smartphone-Besitzer vom Internet in der Tasche begleiten. Sie übernehmen immer mehr Funktionen und Aufgaben. Hinter jeder dieser Technologien versteckt sich ein Verbund aus einzelnen unabhängigen, aber miteinander kommunizierenden elektronischen Steuergeräten (engl. Electronic Control Unit, ECU) oder Kontrollern, sogenannten eingebetteten Systemen (engl. Embedded Systems, ES) mit verteilter Intelligenz. Solche Steuergeräte sind elektronische Kontrollmodule, die zur Überwachung, Steuerung oder Regelung in einem technischen System eingesetzt werden. „Eingebettete Systeme sind informationsverarbeitende Systeme, die in ein größeres Produkt integriert sind, und die normalerweise nicht direkt vom Benutzer wahrgenommen werden“ [43]. Typische Anwendungsbereiche sind Kraftfahrzeuge, Produktionsmaschinen, Telekommunikationsgeräte, Steuerung von Maschinen und industriellen Anlagen und reichen hin bis zu medizinischen Systemen aller Art.

In der modernen Autoindustrie sind Steuergeräte unverzichtbar. Immer mehr Autos werden mit Klimaanlagen, Navigationssystemen, automatisch gesteuerten Türen und Fenstern, Airbag-Steuerungssystemen, automatischen Einparkhilfen und Ähnlichem ausgestattet. Diese Systeme werden von den jeweiligen elektronischen Kontrollern gesteuert. Prozessoren in solchen Kontrollern sind aufgrund der Herstellungskosten ressourcenschwach, sie haben eine relativ niedrige Taktfrequenzen und einen begrenzten Speicher. Aufgrund dieser Limitierungen gilt es, eine zu hohe Belastung der Prozessoren der ECUs zu vermeiden.

Um die verschiedenen Steuergeräte miteinander zu vernetzen, wird ein Kommunikationsmedium benötigt. Für die Bedürfnisse in der Autoindustrie wurden der CAN-Bus (engl. Controller Area Network), der LIN-Bus (engl. Local Interconnect Network), Media Oriented Systems Transport (MOST) and FlexRay entwickelt [41]. Der Einsatz serieller Bussysteme als Datenkommunikationssysteme bietet eine hohe Flexibilität von Systemen in Bezug auf Änderungen oder Erweiterungen sowie die Möglichkeit der Reduktion des Aufwandes für die Projektierung und die Installation [27]. LIN ist ein serieller Feldbus, der auf einem Master/Slave-Prinzip mit Fehlererkennung basiert, wobei die Anzahl der Slaves auf 16 begrenzt ist. Der LIN-Bus ist für eine kostengünstige Kommunikationsinfrastruktur bei Anwendungen der langsamen Steuerung in Fahrzeugen geeignet, wie z.B. Türmodul- und Klimasystemen [38]. Wenn eine größere Bandbreite benötigt wird, kommt der CAN-Bus (engl. Controller Area Network) zum Einsatz.

Seit 1994 ist CAN der am weitesten verbreitete Bus für Automotive-Anwendungen [41] und wird in der International Organization for Standardization (ISO) in der ISO11898 beschrieben [31]. CAN ist ein serieller Feldbus, der auf dem Multi-Master-Prinzip basiert, wobei alle Geräte gleichberechtigt beim Senden und Empfangen sind. Die Kommunikation findet mittels Broadcast-Nachrichten statt, also jedes Gerät empfängt jede Nachricht, unabhängig davon, ob sie für dieses Gerät relevant ist [22]. Das kann zu einer höheren Verarbeitungslast für die einzelnen Empfangsgeräte führen.

Der Einsatz von Hardware- oder Softwarefiltern kann diese unnötige Belastung reduzieren oder sogar ganz vermeiden, indem die Nachrichten herausgefiltert und blockiert werden, die für das entsprechende Steuergerät nicht interessant bzw. irrelevant sind. Wenn z. B. ein Fenster geöffnet wird, interessiert sich das Navigationssystem für dieses Ereignis nicht. Die damit verbundenen Nachrichten sind für das Navigationssystem irrelevant. Die auf der Hardwareebene eingesetzten Filter fangen die für ein Gerät irrelevanten Nachrichten gleich zu Beginn ab, sodass der Rechenaufwand für die später eingesetzten Softwarefilter geringer wird. Ein Softwarefilter wird dann eingesetzt, wenn nicht alle irrelevanten Nachrichten vom Hardwarefilter erkannt werden. Hardwarefilter können konfiguriert werden und bestimmte Bit-Muster erkennen. Eine korrekte Filterkonfiguration akzeptiert alle relevanten Nachrichten, darin besteht die Mindestanforderung an einen Filter. Eine perfekte Filterkonfiguration akzeptiert alle interessanten und blockiert alle irrelevanten Nachrichten, diese Konfiguration liefert die beste Lösung und wird bevorzugt. Eine unvollkommene Filterkonfiguration akzeptiert alle interessanten Nachrichten und akzeptiert einige irrelevante Nachrichten. Diese Konfiguration ist weniger favorisiert jedoch, aufgrund der nicht ausreichenden Ressourcen in der Praxis unvermeidbar.

Das Problem ist nicht nur für die Industrie relevant, sondern auch WissenschaftlerInnen beschäftigen sich damit, wie Hardwarefilter designet werden können. Einige Arbeiten konstruieren solche Filterkonfigurationen basierend auf Prioritäten [28] oder Planbarkeitsanalysen (engl. Schedulability Analysis) [46]. Diese Prioritäten sind in den Nachrichten-IDs enthalten und die IDs sind immer eindeutig. In [28] wird ein Maß eingeführt, das die Güte solcher Filterkonfigurationen messen und somit vergleichen kann. Für jede Nachricht, die von der Filterkonfiguration akzeptiert wird, aber für das jeweilige Steuergerät irrelevant ist, gibt es „Strafpunkte“, deren Summe die Strafpunkte das negative Ausmaß dieser Güte darstellt.

Eine (perfekte) Filterkonfiguration zu finden eröffnet neue Perspektiven auf dem Forschungsfeld. Das Problem kann im Kontext von ganzzahligen linearen Programmen erforscht werden und neue Erkenntnisse für die Welt der Wissenschaft und der Industrie mit sich bringen.

1.1 Ziele der Arbeit

Diese Arbeit ist in zwei Teile aufgeteilt. Der erste Teil untersucht den Fall, dass in dem System alle Nachrichten bekannt sind sowie bekannt ist, welche Nachrichten davon für welches Steuergerät interessant und welche irrelevant sind. Für dieses Szenario soll eine Heuristik entwickelt werden, die eine Filterkonfiguration für das jeweilige Steuergerät designet. In dieser Arbeit wird die Qualität des Filters anhand der in [28] angebotenen Metrik gemessen.
Das erste Ziel besteht darin zu erforschen, unter welchen Bedingungen diese Heuristik welche Ergebnisse liefert und unter welchen Voraussetzungen die gefundene Filterkonfiguration perfekt ist.
Der zweite Teil der Arbeit befasst sich mit dem Design von Nachrichten-IDs, also mit dem umgekehrten Fall. Es wird davon ausgegangen, dass die Filterkonfigurationen für die jeweiligen Steuergeräte bereits bekannt sind. Hier werden Nachrichten-IDs designet, die den Filter passieren dürfen. Das Problem ist als ein ganzzahliges lineares Programm (engl. Integer Linear Programming, ILP) formuliert, die Variablenbelegung des ILPs designet Nachrichten-IDs, die von dem Filter akzeptiert werden. Unter dem Begriff lineare Programmierung (engl. Linear Programming, LP) ist die lineare Optimierung zu verstehen, im Sinne von Planung und nicht im Sinne eines Computerprogramms. Dabei handelt es sich um die Optimierung einer linearen Zielfunktion in einem Bereich, der durch bestimmte Nebenbedingungen (engl. Constraints) eingeschränkt ist. Die Nebenbedingungen werden in der Form von linearen Gleichungen und Ungleichungen ausgedrückt. Wenn einige oder alle Variablen nur ganzzahlige Werte annehmen dürfen, und nicht beliebige reelle Werte, dann handelt es sich um eine ganzzahlige lineare Programmierung (engl. Integer Linear

Programming, ILP) oder einfach eine ganzzahlige Programmierung.

Viele Probleme aus Technik, Wirtschaft oder Logistik lassen sich als lineare Programme darstellen. Diese mathematischen Modelle unterstützen die Entscheidungsprozesse der großen Unternehmen. Es existiert eine Reihe von Algorithmen, wie z. B. der Simplex-Algorithmus, die Branch-and-Bound-Methode oder heuristische Verfahren, die lineare Programme lösen. Da es sich dabei um ein Optimierungsproblem handelt, existieren viele zulässige Lösungen, die das Problem lösen können. Jede Lösung liefert einen bestimmten Wert. Die Algorithmen der linearen Programmierung suchen in der Menge aller zulässigen Lösungen einen Wert, der den besten, optimalen Wert liefert.

Das beschriebene System im zweiten Teil dieser Arbeit ist dynamisch. Zu einem späteren Zeitpunkt werden im System neue Nachrichten definiert, wenn z.B. ein neues Steuergerät mit einer neuen Funktion in das Gesamtsystem integriert wird.

Daraus ergibt sich eine weitere Fragestellung, nämlich wie die Nachrichten-IDs im veränderten System aussehen könnten, damit sie von der bestehenden Filterkonfiguration akzeptiert bzw. blockiert werden.

Um in dieser Arbeit entworfene Algorithmen zu testen und die Ergebnisse zu evaluieren, wird die Programmiersprache Python eingesetzt. Python ist eine beliebte und wirkungsvolle Skriptsprache, die einfach zu erlernen ist. Sie verfügt über eine einfache Syntax und eine große Standardbibliothek sowie über Erweiterungen von Drittanbietern. Dank der Einfachheit der Syntax werden die Programme kurz und verständlich geschrieben. Die Syntax erlaubt trotzdem komplexere Programme für zahlreiche Anwendungsgebiete zu schreiben. Die Sprache wird sowohl von WissenschaftlerInnen als auch von EntwicklerInnen gleichermaßen verwendet. Python ist zuverlässig, flexibel und ziemlich stabil und wird als reine Skriptsprache und auch als Erweiterungssprache benutzt. Python unterstützt alle Arten von Entwicklung: Webanwendungen, Desktop-Apps, Skripts und wissenschaftliche Berechnungen für Forschungs- und Industriezwecke. Python ist nicht kommerziell, die Bibliotheken und Erweiterungen sind kostenlos und können beliebig angepasst werden. Eine starke Entwicklercommunity, die die Sprache regelmäßig pflegt und weiterentwickelt sowie Dokumentationen sind im Internet präsent.

Das zweite Problem, das in dieser Arbeit betrachtet wird (siehe oben), wird als ganzzahlige lineares Programm formuliert. Um das ILP zu lösen, wird die Bibliothek Gurobi verwendet. Der Gurobi Optimizer ist ein hochmoderner Löser (engl. Solver) für lineare Programme und ermöglicht es, komplexe Probleme mathematisch zu formulieren. Der Gurobi Optimizer wird von über 1600 Firmen in knapp zwei Dutzend Branchen eingesetzt, wie z. B.

in der Autoindustrie, Telekommunikation, industriellen Automatisierung, Biotechnologie, Medizin und Pharmazeutik, und hat zu messbaren Verbesserungen beigetragen [5]. Die Software ist kostenpflichtig, eine kostenlose akademische Lizenz sowie eine kostenlose Testversion stehen zur Verfügung. Gurobi bietet für Python eine Schnittstelle. Die mit Hilfe von Gurobi gefundene Lösung eines ganzzahligen linearen Programms löst effizient das Problem. Die Größe der Probleme darf enorm groß sein, denn Gurobi ist in der Lage mit Millionen von Entscheidungsvariablen umzugehen und dabei betrachtet dabei Milliarden von verschiedenen möglichen Lösungen, um die Beste zu finden [7].

1.2 Aufbau der Arbeit

Das Kapitel 2 beginnt mit der Definition grundlegender Begriffe, beginnend mit Controller Area Networks (CAN), gefolgt von dem Konzept des Hardware-Nachrichtenfilters. Zuletzt wird Integer Linear Programming (ILP) erklärt.

Im Kapitel 3 wird das erste Problem, das Design der Hardwarefilterkonfiguration für Broadcast-Busse, vorgestellt, eine Problemdefinition und eine nähere Erklärung präsentiert. Dann folgt der heuristische Entwurf einer Filterkonfiguration. Das Kapitel schließt mit einem Beispiel ab, das die Vorgehensweise verdeutlicht.

Das nächste Kapitel 4 beschäftigt sich mit dem Design der Nachrichten für eine bestehende Hardwarefilterkonfiguration. Zuerst wird das Problem genauer betrachtet. Eine Formulierung als Integer Linear Programming (ILP) wird dargelegt und detailliert erklärt.

Im Kapitel 5 werden zuerst die Implementierungswerkzeuge näher betrachtet. Dazu gehören die Programmiersprache Python, die notwendigen Bibliotheken sowie die Entwicklungsumgebung PyCharm. Dann folgen die Einzelheiten der Implementierung der beiden Fragestellungen dieser Arbeit. Die zentralen Ausschnitte des Programmiercodes werden angegeben, um die Realisierung der Ansätze zu veranschaulichen.

Im Kapitel 6 werden die Ergebnisse der implementierten Algorithmen aus den Kapiteln 3 und 4 vorgestellt. Zuerst wird die Konfiguration der Testplattform beschrieben, dann werden die Eingabedaten für die Experimente definiert. Die Ergebnisse dieser Forschung werden zusammengetragen und abschließend diskutiert.

Diese Arbeit schließt mit dem Kapitel 7, welches kurz die Resultate dieser Arbeit zusammenträgt und einen Ausblick über weitere mögliche Forschungsansätze bietet.

Kapitel 2

Grundlagen

Im Folgenden wird ein Überblick über die Hintergründe und Technologien gegeben, die für das Verständnis dieser Arbeit notwendig sind. Zunächst wird Controller Area Network (CAN), dann werden Hardware-Nachrichtenfilter und dann schließlich Integer Linear Programming (ILP) genauer betrachtet.

2.1 Controller Area Network (CAN)

Wie bereits im Kapitel 2 erwähnt, befasst sich diese Arbeit mit Broadcast-Bussen. Einer der beliebtesten im Kontext der Kommunikation in Kraftfahrzeugen sowie für den Einsatz in industriellen Systemen, ist der CAN-Bus, der im Jahre 1983 von Bosch und Intel entwickelt wurde [22]. CAN wurde konzipiert um kurze Informationen Just-in-Time auszutauschen. Der CAN-Bus kommt insbesondere im Automobilbereich häufig vor, weil damit eine große Anzahl von Leitungen durch einen einzigen Bus ersetzt werden kann. Die Verkabelung eines Autos war bis dahin durch Direktverbindungen (Punkt-zu-Punkt-Verbindung) geprägt, welche zusammen durchaus bis zu 2 km lang und bis zu 100 kg schwer wurden [41]. Es ist verständlich, dass solche Direktverbindungen schwierig zu installieren sind. Der Einsatz des CAN-Buses hat dazu geführt, die Kabellänge deutlich zu reduzieren und damit Kosten und Gewicht zu sparen. Der CAN-Bus ist stabil und günstig und bietet Flexibilität bei der Verkabelung.

Die Kommunikation des CAN-Bus ist international standardisiert und in der ISO11898-1 bis ISO11898-6 ([36], [37], [33], [32], [34], [35]) beschrieben. Diese Standardisierung vereinfacht die Kompatibilität von Geräten verschiedener Hersteller. Wegen der Größe des Automobilmarktes sind CAN-Komponenten verhältnismäßig preiswert und werden aus dem Grund auch in anderen Bereichen, wie z. B. zur Heim-Automatisierung und in Fabriksteuerungen, eingesetzt [43].

Seit Anfang der 1990er Jahre wurde CAN erstmalig in Industriesteuerungen und Serienautomobilen eingesetzt und im Jahr 1992 erstmals in der Mercedes S-Klasse verwendet, wo er als Hochgeschwindigkeitsnetzwerk für die Kommunikation zwischen elektronischen Steuergeräte diente. Kurze Zeit später setzten weitere Automarken, wie z. B. BMW, Porsche und Jaguar, den CAN-Bus in Serienautomobilen ein. Danach Volkswagen, Fiat, Renault und andere ebenfalls dazu entschieden, auf den CAN-Bus umzusteigen [41].

Heutzutage wird CAN als Basistechnologie für die Vernetzung mikrokontroller-gesteuerter Systeme betrachtet und kommt zunehmend in allen Bereichen der Automobilelektronik sowie der Automatisierungstechnik zum Einsatz. Im Sektor der Personen- und Nutzfahrzeuge ist der CAN-Bus weltweit der de-Facto-Standard für die Datenkommunikation zwischen Steuergeräten bzw. für den Anschluss von Sensoren und Aktoren an Steuergeräten (Karosserieelektronik). CAN-basierte Kommunikationssysteme sind in praktisch allen Ausführungsformen mobiler Systeme, von Alarm-, Monitoring und Steuerungssystemen auf Schiffen, über Steuerungs- und Überwachungssystem in Güterwaggons bis hin zur Steuerung und Überwachung von Aufzügen zu finden (vgl. [41], [27]). Jährlich werden über 400 Millionen CAN-fähige Mikrokontroller hergestellt [46].

Die Bedeutung von CAN für alle Bereiche kommt in eindrucksvoller Weise auch dadurch zum Ausdruck, dass heute viele Microkontroller mit mindestens einer integrierten CAN-Schnittstellen zur Verfügung stehen. Dies stellt für die Praxis einen sehr wichtigen Aspekt da, weil der die Möglichkeiten der physikalischen Ankopplung von Busteilnehmern betrifft [27].

2.1.1 Grundlegende Eigenschaften

Der CAN-Bus ist ein serielles Bussystem und gehört zu den Feldbussen [43]. In der industriellen Automatisierung werden Feldbussysteme für die Datenkommunikation eingesetzt. Auch Kommunikationssysteme für die Realisierung verteilter Geräte- und Elektroniksysteme werden als Feldbusse bezeichnet. Die seriellen Bussysteme haben sich in fahrzeuginternen Elektroniksystemen durchgesetzt [41].

Wie im Kapitel 1 bereits erwähnt, basiert der CAN-Bus auf dem seriellen Multi-Master-Prinzip. Die Gründe für diese Kommunikationsart sind Sicherheit und Verfügbarkeit: Sollte ein Steuergerät ausfallen, bricht dank dieser Ausstattung das ganze System nicht zusammen. das ganze System nicht zusammen. Logischerweise wird die Systemleistung entsprechend beeinträchtigt. Bei dem Multi-Master-Prinzip geht es um die Art der Netzwerkzugriffsinitiative zwischen den einzelnen Steuergeräten. Das bedeutet, dass jede Steuergerät grundsätzlich zu jedem beliebigen Zeitpunkt auf das Netzwerk zugreifen kann. Immer,

wenn ein Steuergerät eine Nachricht senden möchte, wird es versuchen, auf das Netzwerk zuzugreifen, um zu übertragen. Beim gleichzeitigem Zugriff sind Kollisionen nicht auszuschließen [41].

Aufgrund des Multi-Master-Prinzips setzt CAN eine CSMA/CD+CR (Carrier Sense Multiple Access/Collision Detection + Collision Resolution)- Mediumzugriffstechnik ein. Das CSMA/CD+CR-Verfahren löst Kollisionen und regelt das Problem des gleichzeitigen Buszugriffs. Ein Steuergerät, das eine Information übertragen möchte, wartet bis der Bus frei ist und erst dann startet die Übertragung [41]. Falls mehrere ECUs gleichzeitig auf den Bus zugreifen, wird die Kollision durch die Arbitrierung anhand der Prioritäten aufgelöst. Diese Prioritäten sind bereits in den zu sendenden Nachrichten enthalten. Wie genau eine Nachricht aufgebaut ist, wird im Abschnitt 2.1.2 detailliert beschrieben. Die genaue Funktionsweise der Arbitrierung ist in [41] und in [27] näher erläutert.

Die CAN-basierte Kommunikation findet grundsätzlich mittels des Broadcast-Prinzips statt. Jede Nachricht im Netz wird an alle anderen Netzwerkteilnehmer gesendet. Der Sender muss die Empfängeradressen nicht kennen. Aus dem Grund können Steuergeräte einfach hinzugefügt werden, ohne aufwändige physikalische Adressierungen durchführen zu müssen. Jedes Steuergerät empfängt jede Nachricht, unabhängig davon, ob sie für dieses Gerät relevant ist. Das kann zu einer höheren Verarbeitungslast für die einzelnen Empfangsgeräte führen, mehr dazu im Abschnitt 2.2. Eine Empfangsbestätigung der Nachricht ist nicht erforderlich.

Eine entscheidende Eigenschaft von CAN ist die Medienzugriffstechnik, die in die Zuständigkeit der Sicherungsschicht fällt. Diese Eigenschaft stellt einen bedeutenden Unterschied zu vielen anderen Bussen dar. Die Zugriffstechnik ist stark von der Topologie des Netzwerkes abhängig [41]. CAN basiert auf einer linienförmigen Topologie. Der Einsatz von Router und Repeater ermöglicht stern- oder baumförmige Typologien und hierarchische Netzstrukturen. Die Anzahl der Teilnehmer im Netz ist nicht durch CAN begrenzt, sondern ist von der Leistungsfähigkeit der eingesetzten Treiberbausteine abhängig [27]. Das Vertiefen in dieses Themas ist nicht für diese Arbeit relevant, deswegen wird an dieser Stelle für mehr Informationen über Medienzugriffstechniken auf [41] verwiesen.

Das CAN-Protokoll ist in den zwei untersten Schichten 1 und 2 des OSI (Open System Interconnection)- Referenzmodells spezifiziert, der Bitübertragungsschicht (engl. Physical Layer) und der Sicherungsschicht (engl. Data Link Layer) [36]. Dies ermöglicht den anderen Anwendungsbereichen basierend auf dem CAN-Protokoll ergänzende Protokolle in einfacher Weise zu entwickeln. Über die Bitübertragungsschicht werden alle für die physikalische Bitstromübertragung notwendigen Regelungen getroffen. Die Sicherungsschicht legt

fest, wie genau die Datenübertragung geregelt wird, es geht um Fehlererkennung, Fehlerbehebung und Flusskontrolle sowie Buszugriff. Die zu übertragende Information, genannt Nachricht, wird in Frames unterteilt. Der Nachricht wird ein Fehlersicherungscode hinzugefügt, der die Fehlererkennung ermöglicht. Die Fehlerbehebung erfolgt durch die erneute Nachrichtenübertragung [27].

2.1.2 Aufbau einer CAN-Nachricht

Die in CAN spezifizierten Nachrichtenformate erfüllen jeweils einen eigenen Zweck und sind in unterschiedlich lange Bit-Sequenzen, sogenannte Felder (engl. Field), unterteilt. Diese Felder tragen Informationen wie Ziel- oder Absenderadresse und werden im Folgenden näher betrachtet.

Die CAN-Kommunikation basiert auf vier Frames (vgl. [27]):

- ein Data Frame dient der Datenübertragung
- ein Remote Frame dient der Anforderung eines bestimmten Daten-Frames von einem anderen elektronischen Steuergerät
- ein Error Frame signalisiert allen Teilnehmern einen erkannten Fehler
- ein Overload Frame dient zur Verzögerung zwischen Daten- bzw. Datenanforderungstelegramm.

CAN spezifiziert zwei Nachrichtenformate, das Standard-Format CAN 2.0A (Base Frame Format) mit 11-Bit-Identifier und das Extended-Format CAN 2.0B (Extended Frame Format) mit 29-Bit-Identifier. CAN 2.0A lässt $2^{11} = 2048$ Adressierungen zu, das bedeutet, dass die Anzahl von verschiedenen Teilnehmern auf 2048 beschränkt ist. Da dies für manche Systeme nicht ausreichend ist, wurde die CAN-Spezifikation um CAN 2.0B erweitert, sodass $2^{29} = 536.870.912$ Adressierungen möglich sind. Diese Nachrichten-Identifier sind immer eindeutig [46]. Darin besteht der einzige Unterschied zwischen den beiden Standards.

Dieser Bit-Identifier (Nachrichten-ID) dient zwei Zwecken über die reine Identifizierung der Nachricht hinaus. Zunächst wird der Identifier als Priorität verwendet, um zu bestimmen, welche Nachricht unter denjenigen, die um den Bus konkurrieren, als Nächstes übertragen wird. Zweitens kann der Identifier von Empfängern verwendet werden, um Nachrichten herauszufiltern, die sie nicht interessieren, und so die Belastung des Mikroprozessors des Empfängers verringern [46].

Die Abbildung 2.1 zeigt den Aufbau einer Nachricht im Standard-Format mit 11-Bit-Identifier. In der Abbildung 2.2 ist der Aufbau einer Nachricht im Extended-Format mit 29-Bit-Identifier zu sehen.

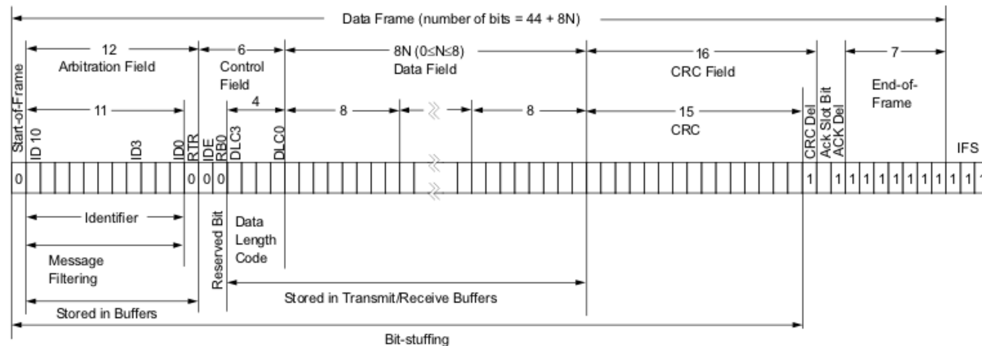

Abbildung 2.1: CAN 2.0A: Base Frame Format [18]

Im Folgenden wird die Aufbau eines Data Frames kurz erläutert (vgl. [27], [41]).

Frame-Anfangskennung (Start of Frame, SOF) Dieses Bit ist dominant und kennzeichnet den Beginn einer Nachricht.

Arbitrierungsfeld (Arbitration Field) Das Feld besteht aus einem Identifier (11 Bit oder 29+2 Bit) und einem RTR-Bit (Remote Transmission Request).

Steuerfeld (Controll Field, CTRL) Das Feld besteht aus sechs Bits, einschließlich vier Bits für die Darstellung der Datenlänge (Data Length Code, DLC). Das erste Bit Identifier Extension (IDE) zeigt an, ob ein Standard- oder ein erweitertes Nachrichtenformat vorliegt. Das zweite Bit Reserved Bit (RBO) ist reserviert für eventuelle Erweiterungen des CAN-Protokolls.

Datenfeld (Data Field) Dieses Feld enthält die eigentlichen Nutzdaten einer CAN-Nachricht und kann 0 bis 8 mal 8 Bit umfassen.

Datensicherungsfeld (CRC Field) Dieses Feld besteht aus 15 Bit (CRC) gefolgt von einem rezessiven CRC-Delimiter-Bit.

Bestätigungsfeld (Acknowledge Field, ACK) Das 2-Bit lange Bestätigungsfeld besteht aus einem Acknowledge Slot und einem rezessiven Begrenzungsbit (ACK-Delimiter).

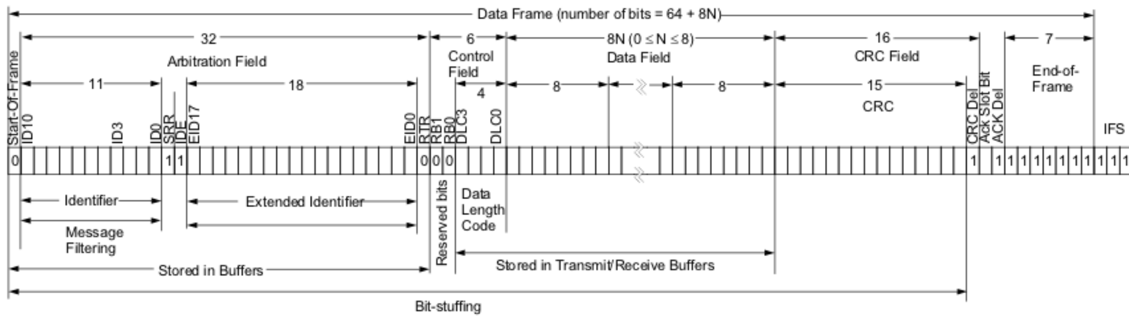

Abbildung 2.2: CAN 2.0B: Extended Frame Format [18]

Frame-Enderkennung (End of Framem, EOF) Der Abschluss der Nachricht wird mit einer Folge von 7 rezessiven Bits signalisiert.

Für diese Arbeit sind lediglich die Nachrichten-IDs relevant und die Berechnungen gelten sowohl für Standard CAN 2.0A mit 11-Bit-Identifier als auch für Extended CAN 2.0B mit 29-Bit-Identifier. Der Einfachheit halber werden in dieser Arbeit als Standard die Nachrichten mit 11-Bit-Identifier betrachtet, andernfalls wird auf die Unterschiede hingewiesen.

2.1.3 Lowspeed- und Highspeed-CAN

Lowspeed-CAN (CAN-B)

Für simplere Aufgaben wird der Lowspeed-CAN eingesetzt, um die Herstellungskosten gering zu halten. Er wird besonders für Bitraten von bis zu 125 kbit/s wird Lowspeed-CAN verwendet [33]. Die niedrigen Bitraten sorgen dafür, dass die Leitungen am Bus viel länger sein können als beim Highspeed-CAN. Lowspeed-CAN wird in PKWs in der Karosserieelektronik und für den Komfortbereich eingesetzt [41], [47].

Highspeed-CAN (CAN-C)

Bei Systemen, welche hohe Bitraten bis zu 250 kbit/s voraussetzen, wird das Highspeed-CAN eingesetzt [37]. Damit wird die Datenübertragung den Echtzeitanforderungen des Antriebsstrang gerecht. Aufgrund der hohen Bitraten ist die mögliche Buslänge jedoch begrenzt. Generell gilt: Je größer die Bitraten sind, desto kürzer muss die Buslänge sein. Umgesetzt wird der Bus als "Hauptleitung" mit "Stichleitungen" von maximal 30 cm Länge. Highspeed-CAN wird in PKWs im Antriebsbereich verwendet [41], [47].

Die Regelung und Steuerung von Motor, Getriebe, Bremsen und Fahrwerk ist nur möglich, wenn die Steuergeräte Informationen schnell austauschen und Stelleingriffe über mehrere Systeme hinweg koordinieren. Die benötigten Informationen sind nur wenige Bytes lang, müssen aber periodisch mit hoher Frequenz, kurzer Verzögerungszeit (Latenz) und großer

Zuverlässigkeit übertragen werden. Durch moderne Fahrwerkssteuerungen steigen die Anforderungen an die Technologien. Daher wurde der CAN-Bus zu CAN FD (CAN with Flexible Data-Rate) weiterentwickelt und neue Bussysteme wie FlexRay konzipiert [60].

Es existieren weitere CAN-basierte höhere Protokolle. Wichtige Vertreter sind DeviceNet, CANKingdom, SAE-J1939 sowie CANopen. Die Protokolle werden in [58] näher erläutert.

2.2 Hardware-Nachrichtenfilter

In Automobilanwendungen wird CAN typischerweise verwendet, um Hochgeschwindigkeitsnetzwerke bis zu 500 kBit/s bereitzustellen, z.B. für Motorregelung und Getriebesteuerung. CAN wird auch für langsame Netzwerke von 100 bis zu 125 kbBts/s eingesetzt (vgl. Kap. 2.1.3). Die über den CAN-Bus gesendeten Nachrichten übertragen Zustandsinformationen, sogenannte Signale, zwischen verschiedenen Steuergeräten. In einem High-End-Fahrzeug, wie z. B. dem Volkswagen Phaeton, kann es bis zu 2500 verschiedene Signale geben [46]. Beispiele für solche Signale sind Informationen über Raddrehzahlen, Temperatur, Klimaeinstellung oder Fehlercodes. Viele solche Signale haben Echtzeiteinschränkungen und die Reaktion soll innerhalb von wenigen Millisekunden erfolgen, wie z. B. bei den Bremsen. Größtenteils setzten Systeme enge Zeitangaben an die Signale, die exemplarisch alle 5 Millisekunden gesendet werden.

Die Datenübertragung über CAN-Netzwerke basiert auf dem Broadcast-Prinzip und alle auf dem Bus übertragenen Signale werden grundsätzlich von allen elektronischen Steuergeräten empfangen (vgl. Kap. 2.1.1). Viele On-Chip-CAN-Kontroller verfügen über mehrere Steckplätze, die entweder zum Senden oder Empfangen einer bestimmten Nachricht zugewiesen werden können. Beispielsweise besitzen einige Motorola, National Semi-conductor, Fujitsu und Hitachi On-Chip-CAN-Peripheriegeräte über 14, 15 oder 16 solcher Steckplätze. Diese Slots haben typischerweise nur einen einzigen Puffer und deshalb ist es notwendig sicherzustellen, dass die vorherigen Nutzdaten einer Nachricht übertragen wurden, bevor die neuen Nutzdaten in den Puffer geschrieben werden. Sonst wird die vorherige Nachricht überschrieben und geht verloren. Dieses Verhalten stellt eine zusätzliche Einschränkung dar [46].

Im Netzwerk werden demnach viele Nachrichten übertragen. In vielen Fällen wird sich ein bestimmtes elektronisches Steuergerät nur für einige wenige, für ihn bestimmte Nachrichten interessieren. Für ein elektronisches Steuergerät interessante Nachrichten werden in dieser Arbeit als als **Desired Messages** bezeichnet und die irrelevanten Nachrichten als **Undesired Messages**.
Aus den oben genannten Gründen ist es sinnvoll, bereits im Protokollkontroller einen zu-

sätzlichen Mechanismus zu implementieren, welcher dafür sorgt, dass der Host-Kontroller nur dann über das Eintreffen einer neuen Nachricht informiert wird, wenn es sich um eine Desired Message handelt (vgl. Kap. 1).

Diese Funktion der sogenannten Akzeptanzfilterung ist in den auf dem Markt verfügbaren Protokollkontroller-Bausteinen in unterschiedlicher Form realisiert. Die Undesired Messages können zu Beginn von Hard- oder Softwaretechniken gefiltert werden. Im Rahmen der Nachrichtenfilterung wird anhand der Nachrichten-ID einer empfangenen Nachricht entschieden, ob diese in den (bzw. einen der) Nachrichtenspeicher des Protokollkontrollers übernommen und der Host-Kontroller vom Eintreffen einer neuen Nachricht unterrichtet werden soll oder nicht [27]. Die Nachrichten-IDs sind immer eindeutig dienen hauptsächlich zur Priorisierung bei der Übertragung (vgl. Kap. 2.1.2). In [28] werden Nachrichten-IDs zum Filtern von Undesired Messages verwendet und somit wird die Belastung des Mikroprozessors vom Empfängers verringert. In [46] wird anhand der Planbarkeitsanalyse entschieden, ob eine Nachricht den Filter passieren darf.

Die Undesired Messages können von einem Filter auf der Hardware- oder Softwareebene erkannt werden. Die auf der Hardwareebene eingesetzten Filter fangen für ein Steuergerät nicht relevante Nachrichten ganz zu Beginn ab, so dass der Rechenaufwand für die später eingesetzten Softwarefilter geringer wird. Ein Softwarefilter wird dann eingesetzt, wenn nicht alle irrelevanten Nachrichten im Hardwarefilter erkannt werden.

Die Mindestanforderung an einen Filter besteht darin, alle Desired Messages zu akzeptieren. Eine solche Filterkonfiguration heißt *korrekt*. Eine *perfekte* Filterkonfiguration akzeptiert alle Desired Messages und blockiert alle Undesired Messages, wie der Name bereits signalisiert, ist diese Konfiguration perfekt und wird bevorzugt, da sie die beste Lösung liefert. Eine *unvollkommene* Filterkonfiguration akzeptiert alle Desired Messages und blockiert einige Undesired Messages. Diese Konfiguration ist wenig bevorzugt jedoch aufgrund der nicht ausreichenden Ressourcen ist in der Praxis unvermeidbar.

Die Herausforderung, Hardware-Nachrichtenfilter für Broadcast-Busse zu entwickeln, wird in der gesamten Forschungsgemeinschaft kaum wahrgenommen [40]. Wie bereits erwähnt, konzentrieren sich die ForscherInnen meist auf die Prioritätsvergabe und die Planbarkeitsanalyse. Das Problem des Hardware-Nachrichtenfilter-Designs ist weder auf CAN noch auf den Automobilsektor beschränkt und kann in jedem Anwendungsbereich eingesetzt werden, in dem Broadcast-Busse verwendet werden. In [40] wird eine Hardware-Filterkonfiguration angeboten, die anhand von Satisfiability Modulo Theories (SMT) formuliert wird. In [28] wird eine Metrik eingeführt, die die Qualität einer Hardware-Filterkonfiguration bewertet.

2.2.1 Funktionsweise

Ein Hardware-Nachrichtenfilter funktioniert wie folgt (vgl. [28]):

- Die eingehenden Nachrichten-ID wird mit dem Filtermuster verglichen.
- Wenn die Nachrichten-ID mit dem Filtermuster übereinstimmt, wird die Nachricht akzeptiert und die Nutzdaten werden in dem Empfangspuffer gespeichert.
- Wenn die Nachrichten-ID nicht mit dem Filtermuster übereinstimmt, wird die Nachricht blockiert.

Das Filtermuster besteht aus zwei Registern pro Filter, genannt Maske und Tag. Die Maske gibt an, welche Bits der ID berücksichtigt werden, und das Tag gibt die entsprechenden ID-Werte an, die den Filter passieren dürfen.

2.2.2 Filterkonfigurationen

Das Prinzip der Hardware-Nachrichtenfilter funktioniert für alle CAN-Kontroller gleich, jedoch gibt es Unterschiede bei der Implementierung und der Anzahl der verfügbaren Filter (vgl. [28], [40]).

AT90CAN128 stellt einen stromsparender 8-Bit-Mikrocontroller dar, sein CAN-Kontroller hat 15 Nachrichtenpuffer und jeder hat seine eigene Maske. Der Kontroller von MultiCAN verfügt ebenfalls über eine eigene Maske für jeden Nachrichtenpuffer [28].
Es existieren einige CAN-Kontroller, die sich eine globale Maske zwischen den Nachrichtenpuffern teilen, anstatt für jeden Puffer eine eigene Maske zu haben. Dies schränkt die Flexibilität und Effizienz ein. Zudem besitzt jeder Puffer seinen eigenen Tag. MPC555 ist ein 32-Bit-Mikrokontroller und hat zwei CAN-Module. M32R hat ebenfalls zwei CAN-Module. Diese CAN-Module haben 16 Nachrichtenpuffer, aber nur 3 Masken, eine globale und zwei lokale Masken. CR16 hat ein CAN-Modul mit 15 Nachrichtenpuffern mit zwei Masken, eine Maske ist lokal und eine global [28].

Die Nachrichtenpuffer werden so konfiguriert, dass sie die Nachrichten entweder senden oder empfangen können. Die fürs Senden konfigurierten Puffer können keine Nachrichten mehr empfangen und puffern [28].
Die Filterkonfigurationen lassen in Bezug auf die Anzahl der Masken und der Tags wie folgt unterscheiden (vgl. [40]):

- eine Maske mit kleiner Anzahl an Tags (1:n),
- mehreren Masken mit mehreren Tags (m:n),
- mehrere Masken, wobei jede Maske ihren eigenen Tag hat (1:1).

Zusammengefasst besitzt jedes Steuergerät eine eigene Filterkonfiguration, die extra für das Gerät konfiguriert ist. Eine Filterkonfiguration kann aus einem oder mehreren Filtern bestehen. Die Anzahl der Filter hängt von der Anzahl der Puffer des elektronischen Steuergerätes ab.

Im Fokus dieser Arbeit steht der Entwurf einer Filterkonfiguration (vgl. Kap. 3) auf der Hardwareebene. Um den Umfang dieser Arbeit nicht zu sprengen, beschränkt sich diese Arbeit auf solche Filterkonfigurationen, die aus einem einzigen Filter bestehen, der aus einer Maske und einem Tag besteht. Somit werden die Begriffe Filter und Filterkonfiguration gleich verwendet.

2.3 Integer Linear Programming (ILP)

In Wissenschaft, Politik, Sport, Industrie und Wirtschaft ist es natürlich, das Bestmögliche erreichen zu wollen. Ebenso ist es erwünscht, die beste Lösung für das vorliegende Problem zu erhalten. Um komplexe reale Probleme darzustellen, bieten sich mathematische Modelle an. Diese Modelle berücksichtigen nicht nur eine Zielsetzung, sondern beinhalten auch deren Rahmenbedingungen. Es gibt verschiedene Wege, ein Problem mathematisch zu beschreiben. Dabei lassen sich durch die Nebenbedingungen unzulässige Lösungen eliminieren. Die beste Lösung aus allen zulässigen Lösungen zu finden ist die Hauptaufgabe von Optimierung (vgl. [23]). Unter dem Begriff lineare Programmierung (engl. Linear Programming, LP) ist die lineare Optimierung zu verstehen, im Sinne von Planung und nicht im Sinne eines Computerprogramms. Lineare Programmierung ist ein Teilgebiet des Operations Research (OR). „Operations Research beschäftigt sich mit der Analyse von praxisnahen, komplexen Problemstellungen im Rahmen eines Planungsprozesses zum Zweck der Vorbereitung von möglichst guten Entscheidungen durch die Anwendung mathematischer Methoden“ [25]. Seit den 1950er Jahren beschäftigen sich Betriebswirtschaftler, Mathematiker, Ingenieure, Informatiker und Mediziner [59] intensiv mit dem Gebiet, wobei sich die OR in den letzten 20 Jahren stürmisch entwickelt hat [25].

Die Hauptaufgaben der Operations Research bestehen darin, ein reales Problem auf ein mathematisches Modell abzubilden und mit Hilfe von (Optimierungs-) Algorithmen eine Lösung für das Problem zu finden. Dabei spielt die Unterstützung durch Optimierungssoftware eine zentrale Rolle (vgl. [25]).

Wie bereits erwähnt, ist ein Teil von Operations Research die lineare Programmierung. Die Modelle der linearen Programmierung (LP) bestehen aus einer zu optimierenden linearen Funktion und Nebenbedingungen in der Form von linearen Gleichungen und Ungleichungen. Die Variablen der LP nehmen reelle Werte an, bei der ganzzahligen linearen Program-

mierung (ILP) nehmen die Variablen ganzzahlige oder sogar binäre (0 bzw. 1) Werte an. Viele Optimierungsprobleme lassen sich mathematisch als ganzzahlige oder binäre lineare Programme formulieren. Weitere Informationen zu dem Thema Operations Research werden in [25] und [59] näher erläutert.

2.3.1 Geschichte der Entstehung und Entwicklung

In den 1920er Jahren begann eine Gruppe von angesehenen Mathematikern und Ökonomen die Probleme von Volkswirtschaft in der Form von linearen Gleichungen bzw. Ungleichungen mit einer endlichen Anzahl von Variablen zu untersuchen. Später in den 1940ern unabhängig voneinander und ohne es zu wissen, haben WissenschaftlerInnen aus Europa, Amerika und ehemaligem Sowjetunion auf demselben Themengebiet geforscht. Die ersten Anwendungen betrafen kostenminimale ausreichende Ernährung, saisonal bedingte Lagerprobleme und optimale Strategien für den Kauf, die Lagerung und den Verkauf einer homogenen Ware, wie z.B. Getreide [44].

Der Begriff lineare Programmierung stammt aus dem Jahr 1948 als George B. Dantzig, damals mathematischer Berater bei der Air Force, im Kontext seiner Arbeit erfolgreich die Abläufe mathematisch formulierte und sie auch löste. Aus der Arbeit ist der bekannte Simplex-Algorithmus zur Lösung des daraus resultierenden Optimierungsproblems entstanden. Nachdem das Potenzial der linearen Programmierung als Entscheidungshilfe in großen Unternehmen erkannt worden, folgten industrielle, ingenieurtechnische und wirtschaftliche Anwendungen. Dies führte zu einem messbar höheren Bedarf und die Bereitschaft, die Entscheidungsprozesse großer Unternehmen analytisch zu modellieren. Die Methodik der linearen Programmierung (LP) wurde in praktisch allen Funktionsbereichen des Unternehmens vorgeschlagen und angewendet, wie z.B. Buchhaltung, Finanzen, Marketing, Produktion oder Betriebsführung. Die lineare Programmierung wird am häufigsten als das Problem der optimalen Zuweisung knapper Ressourcen für die Wirtschaftstätigkeit, u.a. Produktion, interpretiert [44].

Seit den 1950er Jahren hat sich die ganzzahlige Programmierung (ILP) zu einem Modellierungs- und Optimierungswerkzeug für viele praktische Probleme entwickelt, für die zu dem Zeitpunkt keine speziellen Algorithmen bekannt waren. Seit den 1980er Jahren hat sich diese Situation geändert und bis heute nutzen viele Anwendungen die ILP-Lösungsverfahren, z.B. Produktionsplanung oder Transportprobleme [25], [44].

Vor dem Hintergrund der begrenzten Rechenleistung der Computer von 1990er Jahre ist es amüsant zu lesen, was vor etwa vierzig Jahren machbar war und als groß angesehen wurde. Ein Transportproblem mit 25 Start- und 60 Zielorten konnte durch manuelle Be-

rechnungstechniken gelöst werden. Probleme dieser Größenordnung werden heute mit die richtigen Software innerhalb von Sekunden auf einem Personalcomputer bewältigt. Im Jahr 1951 konnte ein Optimierungsproblem mit 10 Gleichungen oder Ungleichungen des linearen Programms gelöst werden. Im Jahr 1975 lag die Grenze schon bei 16.000 und Ende 1990er bei 8.000 linearen Einschränkungen mit 3.000.000 Variablen [44].

Dank der technischen Entwicklung ist Software heutzutage in der Lage, Berechnungen mit Millionen von Entscheidungsvariablen durchzuführen und betrachtet dabei Milliarden von verschiedenen möglichen Lösungen, um die Beste zu finden [23], [7].

2.3.2 Grundlegende Definitionen

Im Folgenden werden die wichtigsten Definitionen der ganzzahligen linearen Programmierung (ILP) zusammengestellt.

Viele Optimierungsprobleme werden als Maximierung oder Minimierung einer Zielfunktion unter beschränkten Ressourcen und Nebenbedingungen formuliert. Falls die Zielfunktion durch eine lineare Funktion bestimmter Variablen und die Nebenbedingungen durch lineare Gleichungen oder Ungleichungen dieser Variablen spezifiziert werden können, dann liegt ein lineares Programm vor. Wenn es die zusätzliche Forderung gibt, dass alle Variablen ganzzahlige Werte annehmen müssen, dann handelt es sich um eine ganzzahlige lineare Programmierung (ILP) [50]. Eine gemischte ganzzahlige lineare Programmierung liegt vor, wenn die Variablen sowohl reelle als auch ganze Werte annehmen dürfen. Im Rahmen dieser Arbeit wird die gemischte ganzzahlige lineare Programmierung nicht betrachtet.

Mathematische Formulierung eines ILP

Seien $\{a_1, \ldots, a_n\} \in \mathbb{R}$ eine Menge ganzer Zahlen und $\{x_1, \ldots, x_n\} \in \mathbb{V}ar$ eine Menge von Variablen, die typischerweise nicht negativ sind. Die zu optimierende lineare Zielfunktion definiert als:

$$a_1 \cdot x_1 + \cdots + a_n \cdot x_n = \sum_{j=1}^{n} a_j \cdot x_j \tag{2.1}$$

Wenn die Zielfunktion minimiert wird, dann handelt es um ein Minimierungsprogramm. Wenn die Zielfunktion zu maximieren ist, dann liegt ein Maximierungsprogramm vor.

Die Nebenbedingungen werden durch Gleichungen und Ungleichungen dargestellt. Sei b eine ganze Zahl und f eine lineare Funktion. Dann ist die lineare Gleichung wie folgt definiert:

$$f(x_1, \ldots, x_n) = b. \tag{2.2}$$

Die linearen Ungleichungen sind folgendermaßen definiert:

$$f(x_1, \ldots, x_n) \leq \; b \tag{2.3}$$

$$f(x_1, \ldots, x_n) \geq \; b. \tag{2.4}$$

Nachdem eine Formulierung eines Problems als ILP fest steht, wird ein iteratives Verfahren oder ein Algorithmus benötigt, um eine optimale Lösung für das ILP zu finden. Optimierungsprobleme sind Probleme, die im Allgemeinen viele zulässige Lösungen besitzen. Jeder Lösung ist ein bestimmter Zielfunktionswert zugeordnet. Optimierungsalgorithmen suchen in der Menge aller zulässigen Lösungen diejenigen mit dem besten, dem optimalen, Wert [44].

Simplex-Algorithmus

Es existieren viele Algorithmen, die ein ILP lösen können. Der Simplex-Algorithmus ist der älteste und in der Praxis sehr schnell und effizient, im schlimmsten Fall ist er jedoch langsam. Der Simplex-Algorithmus ist ein iteratives Verfahren, das in einer endlichen Anzahl von Schritten entweder

- zu dem Schluss kommt, dass ein ILP überhaupt keine Lösung hat, oder
- zu dem Schluss kommt, dass ein ILP eine unbegrenzte Lösung hat, oder
- eine optimale Lösung für das ILP findet.

Wenn der erste oder zweite Fall im Simplex-Algorithmus in einer Live-Anwendung der linearen Programmierung auftritt, dann ist mit hoher Wahrscheinlichkeit etwas in der Formulierungsphase schief gelaufen. Dieses Ergebnis ist ein Indikator dafür, dass die Formulierung überprüft werden muss [44]. Wenn eine Lösung zurück gegeben wird, dann ist diese Lösung definitiv **optimal**.

Nicht jede Optimierungsaufgabe besitzt eine optimale Lösung. Wenn sich die ins Modell aufgenommenen Nebenbedingungen widersprechen, gibt es nicht einmal zulässige Lösungen. Selbst wenn zulässige Lösungen bekannt sind, kann es sein, dass keine zulässige Lösung besser ist als jede andere. Umgekehrt können auch mehrere, sogar unendlich viele beste zulässige Lösungen vorhanden sein, die dann im Sinne der Zielsetzung alle optimal sind. Die zentrale Aufgabe der mathematischen Optimierung ist, konstruktive Verfahren zur Berechnung von optimalen Lösungen zu entwickeln, zu analysieren und zu bewerten [23].
Es existiert eine Reihe von Algorithmen, die ILPs lösen, wie z. B. der Ellipsoid-Algorithmus, Schnittebenen-Verfahren, Branch-and-Bound und Branch-and-Cut Methoden. Diese und andere Algorithmen werden in [39], [59], [25], [53] näher erläutert.

Fast jedes Optimierungsproblem lässt sich anhand von Heuristiken losen, die für dieses spezielle Problem schnell zulässige Lösungen findet. Der Begriff Heuristik ist von dem griechischen Verb $\epsilon\upsilon\rho\iota\sigma\kappa\epsilon\iota\nu$ (heuriskein) abgeleitet und bedeutet herausfinden, entdecken. Heuristische Verfahren (Heuristiken) unterscheiden sich von exakten Verfahren, indem sie nicht garantieren, dass die gefundene Lösung optimal ist. Heuristiken bestehen aus bestimmten Vorgehensschritten und erscheinen als sinnvoll, zweckmäßig und erfolgversprechend. Heuristische Verfahren sind jeweils auf spezielle Probleme zugeschnitten [24].

2.3.3 ILP-Beispiel

Die ganzzahlige lineare Programmierung lässt sich am besten anhand konkreter Beispiele verstehen. Das folgende Exempel aus der Politik stellt ein konkretes Problem dar, das als ein ganzzahliges lineares Programm formuliert werden kann (vgl. [50]).

Ein Politiker möchte eine Wahl gewinnen. Seine Hauptthemen sind Straßenbau, Sicherheitspolitik, Landwirtschaftsbeihilfe und Mineralölsteuer. Der Wahlkreis besteht aus drei Typen von Gebieten – urbanen, suburbanen und ländlichen, die entsprechend 100.000, 200.000 und 50.000 registrierte Wahlberechtigte haben. Angenommen 50% der Wahlberechtigten gehen tatsächlich zur Wahl.

Bestimmte Themen in bestimmten Gebieten sind besonders dazu geeignet, Stimmen zu gewinnen. Aufgrund der Untersuchungen der Wahlkampfberater ist es möglich, für jedes der Wahlkampfthemen einzuschätzen, wie viele Stimmen der Politiker in jeder Bevölkerungsschicht verlieren oder gewinnen kann, wenn er jeweils 1.000 Euro an Werbemitteln für ein Thema einsetzt. Diese Informationen sind in der Tabelle 2.1 veranschaulicht.

Wahlkampfthema	urban	suburban	ländlich
Straßenbau	-2	5	3
Sicherheit	8	2	-5
Landwirtschaftsbeihilfe	0	0	10
Mineralölsteuer	10	0	-2

Tabelle 2.1: Beispiel: Die Auswirkungen von Wahlkampftaktiken auf die Wähler

Jeder Eintrag dieser Tabelle gibt die Anzahl der Stimmberechtigten (in Tausenden) aus entsprechenden Wahlgebieten an, die durch die Investition von 1.000 Euro an Werbemitteln für ein bestimmtes Wahlkampfthema gewonnen werden können. Negative Einträge geben die Stimmen an, die verloren gehen würden.

Die Aufgabe besteht darin den minimalen Geldbetrag zu bestimmen, den der Politiker investieren müsste, um 50.000 Stimmen in den urbanen Gebieten, 100.000 Stimmen in den suburbanen Gebieten und 25.000 Stimmen in den ländlichen Gebieten zu gewinnen.

Durch Ausprobieren lässt sich schnell eine mögliche Lösung zu finden, mit der die erforderlichen Stimmen zusammen kommen. Eine solche Lösung muss nicht unbedingt die kostengünstigste sein. Um eine bestmögliche Lösung zu finden, lässt sich das Problem mit Hilfe von der ganzzahligen linearen Programmierung lösen.

Sei x_1 der Betrag in Euro (in Tausenden), der für die Kampagne für den Straßenbau ausgegeben wird. Seien x_2, x_3 und x_4 entsprechend die Beträge, die für die Kampagnen für die Sicherheit, die Landwirtschaftsbeihilfen und die Mineralölsteuer ausgegeben werden.

Die Forderung, mindestens 50.000 Stimmen in den urbanen Gebieten zu gewinnen, kann durch die Ungleichung ausgedrückt werden:

$$-2 \cdot x_1 + 8 \cdot x_2 + 0 \cdot x_3 + 10 \cdot x_4 \geq 50. \tag{2.5}$$

Entsprechend die Forderung, mindestens 100.000 Stimmen in den suburbanen und mindestens 25.000 Stimmen in den ländlichen Gebieten zu gewinnen, in der Form:

$$5 \cdot x_1 + 2 \cdot x_2 + 0 \cdot x_3 + 0 \cdot x_4 \geq 100 \tag{2.6}$$

und

$$3 \cdot x_1 - 5 \cdot x_2 + 10 \cdot x_3 - 2 \cdot x_4 \geq 25. \tag{2.7}$$

Da es keine Werbung mit negativen Kosten gibt, gelten weitere Einschränkungen:

$$x_1 \geq 0,\ x_2 \geq 0,\ x_3 \geq 0,\ x_4 \geq 0. \tag{2.8}$$

Jede Belegung der Variablen x_1, x_2, x_3 und x_4, die die Ungleichungen (2.5)-(2.7) erfüllt, führt zu einer Lösung, in jeder der Bevölkerungsgruppen die notwendige Stimmenanzahl zu bekommen.

Dass die Kosten so gering wie möglich gehalten werden sollen, entspricht dem Ausdruck:

$$\textit{minimiere} \;\; x_1 + x_2 + x_3 + x_4. \tag{2.9}$$

Zusammengefasst sieht das ganzzahlige lineare Programm für das Problem folgendermaßen aus:

$$minimiere \ \ x_1 + x_2 + x_3 + x_4 \tag{2.10}$$

unter den Nebenbedingungen

$$-2 \cdot x_1 + 8 \cdot x_2 + 0 \cdot x_3 + 10 \cdot x_4 \geq 50 \tag{2.11}$$

$$5 \cdot x_1 + 2 \cdot x_2 + 0 \cdot x_3 + 0 \cdot x_4 \geq 100 \tag{2.12}$$

$$3 \cdot x_1 - 5 \cdot x_2 + 10 \cdot x_3 - 2 \cdot x_4 \geq 25 \tag{2.13}$$

$$x_1,\ x_2,\ x_3,\ x_4 \geq 0. \tag{2.14}$$

Die Lösung dieses ganzzahligen Programms liefert eine optimale Strategie.

Weitere Beispiele werden in den Quellen [44], [20] näher erläutert.

2.3.4 Big-M-Methode

Viele Fragestellungen aus der Praxis, auch solche, die auf den ersten Blick nicht wie ein mathematisches Optimierungsproblem aussehen, lassen sich gut als ILP formulieren und lösen. Mansche Entscheidungssituationen benötigen eine sogenannte Hilfsvariable [49].

Das Big-M ist eine künstliche, fiktive Variable und steht für eine „hinreichend große Zahl“. Diese Zahl wird den Gleichungen zugefügt, wenn z.B. in Unternehmen manche Produkte nur ab einer Mindestmenge angekauft, produziert oder verkauft werden [25]. Das Hilfsmodell wird aufgestellt, indem der Zulässigkeitsbereich durch das Hinzufügen von Hilfsvariablen zunächst künstlich vergrößert wird, sodass im Nullpunkt gestartet werden kann [56].

Für das Big-M ist ein so großer Wert wie nötig, aber ein so kleiner Wert wie möglich, zu wählen. Wenn der Wert von M um Größenordnungen höher als die anderen Variablenwerte des Modells ist, kann es bei großen Modellen zu numerischen Ungenauigkeiten und somit zu Konvergenzproblemen kommen [49].

Insbesondere ist die Big-M-Methode eine generalisierte Form des Simplex-Verfahrens und erlaubt es, sowohl Maximierungs- und Minimierungsprobleme zu lösen [56].

Beispiel

Die Big-M-Methode lässt sich am besten anhand eines konkretes Beispiele verstehen. Das folgende Beispiel zeigt eine der möglichen Big-M-Methoden Anwendungen. Im Kapitel 4 wird die Big-M-Methode verwendet, um „wenn-dann“ -Bedingung in linearer Form auszudrücken. Das folgende Beispiel veranschaulicht ein ähnliches Problem und zwar eine Fallunterscheidung [49].

Der Ausdruck

$$\mathrm{x}_1 = \begin{cases} x_2, & \text{wenn } y = 1 \\ 0, & \text{sonst} \end{cases}$$

lässt sich als $x_1 = x_2 \cdot y$ mit x_1, $x_2 \geq 0$ darstellen, ist jedoch in dieser Form nicht linear.

Die Forderungen hier sind der Form:

- wenn $y = 1$ dann $x_1 = x_2$
- wenn $y = 0$ dann $x_1 = 0$.

Die zweite Forderung lässt sich folgend darstellen:

$$x_1 \leq M_1 \cdot y, \tag{2.15}$$

wobei M_1 als obere Schranke von x_1 gewählt wird.

Die Gleichung $x_1 = x_2$ wird folgendermaßen transformiert: $x_1 - x_2 \leq 0$ und $x_2 - x_1 \leq 0$.

Die Wirksamkeit dieser Ungleichungen für $y = 1$ wird dann durch:

$$x_1 - x_2 \leq M_2 \cdot (1 - y) \tag{2.16}$$

$$x_2 - x_1 \leq M_3 \cdot (1 - y) \tag{2.17}$$

gewährleistet, wobei M_2 bzw. M_3 als obere Schranke des Ausdrucks $x_1 - x_2$ bzw. $x_2 - x_1$ gewählt wird.

Falls $y = 1$, dann gelten (2.16) und (2.17). Falls $y = 0$, dann gilt (2.15).

Weitere Beispiele zu der Big-M-Methode werden in [49], [56], [25], [53] näher betrachtet.

Kapitel 3

Design der Filterkonfiguration

Dieses Kapitel stellt eine Heuristik vor, die eine perfekte Hardware-Filterkonfiguration berechnet. Im System ist bereits bekannt, welche Nachrichten existieren sowie für welches Steuergeräte welche Nachrichten Desired Messages und Undesired Message sind. Eine perfekte Filterkonfiguration eines Steuergerätes soll alle für das Gerät relevante Nachrichten akzeptieren und alle irrelevanten Nachrichten blockieren. In [28] wird eine Metrik eingeführt, die auf der Vergabe sogenannten Strafpunkten basiert (s. (3.1)). Sie ermöglicht es, die Güte der verschiedenen Filterkonfigurationen zu vergleichen.
Die oben genannte Metrik, genannt Quality of Filter (QoF), ist folgendermaßen definiert:

$$QoF = \sum_{m \in M^U} \frac{[m \textit{ is accepted}]}{P_m}. \tag{3.1}$$

Dabei gilt $[m \textit{ is accepted}] = 1$, wenn die Nachricht m von dem Filter akzeptiert wird, und $[m \textit{ is accepted}] = 0$ sonst. Ein Filter ist perfekt, wenn $QoF = 0$, also keine Undesired Messages akzeptiert werden.

3.1 Problembeschreibung

Das System besteht aus einer Menge von ECUs, genannt Knoten, die an einen Broadcast-Bus angeschlossen sind. Jeder Knoten kann Nachrichten senden, die alle anderen Knoten empfangen können. Jede Nachricht m ist durch einen eindeutig vordefinierten Message Identifier (ID) x_m und eine minimale Zwischenankunftszeit oder Periode P_m gekennzeichnet. Solche Broadcast-Nachrichten enthalten weitere Frames, wie z. B. Nutzdaten, die aber für diese Arbeit nicht relevant sind. Die IDs können eine Länge l von 11 oder 29 Bits haben, je nach Spezifikation (Base Frame Format, Extended Frame Format, vgl. Kap 2.1.2). M ist die Menge aller vordefinierten Message Identifier und diese wird auch als Satz vordefinierter Nachrichten bezeichnet.

Jeder Knoten verfügt über eine Menge von Puffern, die die Nachrichten senden oder empfangen können. Nicht jede Broadcast-Nachricht ist für jeden Knoten relevant. Deswegen bezeichnet jeder Knoten eine Teilmenge von Nachrichten, die er empfangen muss, als Desired Messages M^D, und eine Teilmenge von Nachrichten, an denen er nicht interessiert ist, die als Undesired Messages M^U. Für jeden Knoten gilt $M^D \cup M^U = M$ und $M^D \cap M^U = \emptyset$.

Um die Nachrichtenakzeptanz zu prüfen, verfügt jeder Knoten über eine Menge von Filtern $F = f_1, \ldots, f_n$, genannt Filterkonfiguration. Jeder Filter wird durch ein Tupel $(mask, tag)$ charakterisiert, wobei $mask$ die Maske ist und tag der Tag. Die Maske und der Tag sind Bitvektoren der Länge l. Die Maske gibt an, welche Bits der ID berücksichtigt werden, und der Tag gibt die entsprechenden ID-Werte an, die akzeptiert werden und passieren dürfen.

Die Menge $M^D \cup M^U = M$ enthält nicht alle möglichen Nachrichten, da sie nur aus vordefinierten Nachrichten besteht. Die Menge nicht vordefinierten Nachrichten heißt Unspecified oder Nonexistent Messages M^{ok}, wobei $M^{ok} = M^{all} \cap M^U$ ist und deswegen keine Auswirkung auf QoF hat [40].

3.2 Formulierung des Problems als Heuristik

Die Desired Messages M^D sowie deren Anzahl $|M^D|$ sind bekannt.

Die Undesired Messages M^U sowie deren Anzahl $|M^U|$ sind ebenfalls bekannt.

Die Länge der Nachrichten-ID ist l, wobei $l = 11$ oder $l = 29$ ist, je nach Spezifikation (vgl. Kap. 2.1.2).

Gesucht sind die Maske $mask$ und der Tag tag, die den Filter f beschreiben.

Die Idee besteht darin, dass für jede Nachricht $x_{m,j}$ aus der Menge der Desired Messages M^D einzelne Summen über die einzelnen Bits gebildet werden und aus dem Ergebnis der einzelnen Summen werden die Maske $mask$ und der Tag tag gebildet. Die Maske und der Tag bilden zusammen die Filterkonfiguration f: die Maske zeigt an, welcher Bit des Tags zu betrachten ist. Die Filtervariable f dient nur zum Zweck der Veranschaulichung der Lösung und besitzt keine eigentliche Funktion.

Da die Länge einer Nachricht l ist, werden dementsprechend l Mal die Summen Sum_j berechnet. An dieser Stelle ist es nicht essentiell, ob die Länge der Nachricht 11 oder 29 ist, die Berechnung lässt sich leicht erweitern.

Die Regeln, anhand deren nun die Bitwerte des Filters gewählt werden, lauten wie folgt:

$$Sum_j = \sum_{m=1}^{|M^D|} x_{m,j}, \;\; \forall j = 1, \ldots, l. \tag{3.2}$$

- Wenn die Summe Sum_j gleich 0 ist, dann ist die Maske $mask_j$ gleich 1 und der Tag tag_j gleich 0. Der Filter f_j wird auf 0 gesetzt:

$$wenn\ Sum_j = 0,\ dann\ mask_j = 1,\ tag_j = 0,\ f_j = 0,\ \forall j = 1, \ldots, l. \tag{3.3}$$

- Wenn die Summe Sum_j gleich $|M^D|$, dann ist die Maske $mask_j$ gleich 1 und der Tag tag_j gleich 1. Der Filter f_j wird auf 1 gesetzt:

$$wenn\ Sum_j = |M^D|,\ dann\ mask_j = 1,\ tag_j = 1,\ f_j = 1,\ \forall j = 1, \ldots, l. \tag{3.4}$$

- Sonst ist die Maske $mask_j$ gleich 0 und der Tag tag_j kann zufällig gewählt werden, entweder 0 oder 1. Der Filter f_j wird auf $*$ (don't care) gesetzt:

$$wenn\ 0 < Sum_j < |M^D|\ dann\ mask_j = 0,\ tag_j \in \{0, 1\},\ f_j = *, \forall j = 1, \ldots, l. \tag{3.5}$$

Die Laufzeit dieser Heuristik ist linear.

Sollte eine Nachricht $x_{m,j}$ aus Undesired Messages $|M^U|$ die Filterkonfiguration F passieren, wird nach der Formel (3.1) die Qualität des Filters berechnet. Diese Metrik der Güte bietet die Möglichkeit, unterschiedliche Filterqualitäten zu vergleichen.
Wenn die Qualität des Filters gleich 0 ist, ist der Filter perfekt.
Diese Heuristik wird im Kapitel 6 für unterschiedliche Eingabedaten getestet und ausführlich beschrieben.

3.3 Beispiel

In diesem Abschnitt wird ein Beispiel vorgestellt, das die Vorgehensweise der Heuristik verdeutlicht.

Die Eingabe ist wie in [40] definiert (vgl. Tabelle 3.1). Es sind sechs Nachrichten $x_1, \ldots,$ x_6 gegeben, x_2, x_3, x_5, x_6 sind Desired Messages. Für die x_1 und x_4, die aus der Menge der Undesired Messages sind, betragen die Kosten für das Passieren des Filters $\frac{1}{P_1}$ und $\frac{1}{P_4}$. Für die Desired Messages entstehen keine unerwünschten Kosten.

Die Summen werden über die einzelnen Bits der Nachrichten aus der Menge der Desired Messages gebildet:

ID	Bit_1	Bit_2	Bit_3	Bit_4	Bit_4	Bit_6	Bit_7	Bit_8	Bit_9	Bit_{10}	Bit_{11}	desired
x_1	0	0	0	1	1	0	0	0	0	0	1	
x_2	0	0	0	1	1	0	0	1	0	0	0	ja
x_3	0	0	0	1	1	0	0	1	0	0	1	ja
x_4	1	0	0	1	0	0	0	1	0	0	0	
x_5	1	0	0	1	1	0	0	1	0	0	0	ja
x_6	1	0	0	1	1	0	0	1	0	0	1	ja

Tabelle 3.1: Beispiel: Desired Messages und Undesired Messages

Bit_1: $Sum_1 = 0 + 0 + 1 + 1 = 2$
Bit_2: $Sum_2 = 0 + 0 + 0 + 0 = 0$
Bit_3: $Sum_3 = 0 + 0 + 0 + 0 = 0$
Bit_4: $Sum_4 = 1 + 1 + 1 + 1 = 4$
Bit_5: $Sum_5 = 1 + 1 + 1 + 1 = 4$
Bit_6: $Sum_6 = 0 + 0 + 0 + 0 = 0$
Bit_7: $Sum_7 = 0 + 0 + 0 + 0 = 0$
Bit_8: $Sum_8 = 1 + 1 + 1 + 1 = 4$
Bit_9: $Sum_9 = 0 + 0 + 0 + 0 = 0$
Bit_{10}: $Sum_{10} = 0 + 0 + 0 + 0 = 0$
Bit_{11}: $Sum_{11} = 0 + 1 + 0 + 1 = 2$

Daraus ergibt sich die folgende Filterkonfiguration, der in der Tabelle 3.2 zusammengetragen ist.

ID	Bit_1	Bit_2	Bit_3	Bit_4	Bit_5	Bit_6	Bit_7	Bit_8	Bit_9	Bit_{10}	Bit_{11}
x_2	0	0	0	1	1	0	0	1	0	0	0
x_3	0	0	0	1	1	0	0	1	0	0	1
x_5	1	0	0	1	1	0	0	1	0	0	0
x_6	1	0	0	1	1	0	0	1	0	0	1
Summe	2	0	0	4	4	0	0	4	0	0	2
Maske	0	1	1	1	1	1	1	1	1	1	0
Tag	0	0	0	1	1	0	0	1	0	0	1
Filter	*	0	0	1	1	0	0	1	0	0	*

Tabelle 3.2: Summe der einzelnen Bits, Maske, Tag und Filterkonfiguration [40]

Gemäß des QoF aus der Formel (3.1) ist dieser Filter perfekt, da die Nachrichten x_1 und x_4 aus der Menge der Undesired Messages von dem Filter nicht akzeptiert werden.

Kapitel 4

Design der Nachrichten

Dieses Kapitel stellt eine Lösung zum Designen von Nachrichten vor, wenn die Filterkonfiguration bereits bekannt ist. Im ersten Abschnitt wird das Problem vorgestellt und im zweiten Abschnitt wird das Problem als Integer Linear Programm(ILP) formuliert.

4.1 Problembeschreibung

Das System besteht aus einer Menge von elektronischen Steuergeräten, die an einen Broadcast-Bus angeschlossen sind. Jede über Netzwerk versendete Nachricht m ist durch eine eindeutige Nachrichten-ID x_m und eine Länge l gekennzeichnet, wobei $l = 11$ oder $l = 29$.

Die Anzahl der Desired Messages $|M^D|$ ist bekannt, jedoch die Nachrichten-IDs selbst sind nicht bekannt.

Jedem Steuergerät ist seine Filterkonfiguration bekannt, also die Maske $mask$ und der Tag tag sind bekannt.

Das Ziel ist anhand von der bekannten Filterkonfiguration herauszufinden, wie die Nachrichten-IDs aussehen können, um vom dieser Filterkonfiguration akzeptiert zu werden.

4.2 Formulierung des Problems als ILP

Gesucht ist eine ILP-Formulierung, die anhand der Variablenbelegung der optimalen Lösung ermöglicht die Nachrichten-IDs zu designen, die von der gegebenen Filterkonfiguration akzeptiert werden.

Da der Tag eine l-stellige Kombination aus 0 und 1 ist, ist es möglich den Tag in zwei Teilmengen zu separieren:

- T_0 ist die Menge aller Tag-Bits, die gleich 0 sind,
- T_1 ist die Menge aller Tag-Bits, die gleich 1 sind,

wobei

$$T = T_0 \cup T_1 \tag{4.1}$$

$$T_0 \cap T_1 = \emptyset \tag{4.2}$$

mit $tag_j \in T$ mit $j = 1, \ldots, l$.

Sein C_1, C_2 und C_3 hinreichend große Zahlen (vgl. Kap. 2.3.4).

Neue Formulierung des ILPs:

$$maximiere \quad 0 \cdot \alpha \tag{4.3}$$

unter den Nebenbedingungen:

$$(1 - mask_j) \cdot C_1 + \sum_{m=1}^{|M^D|} x_{m,j} = 0, \ \forall\, tag_j \in T_0 \tag{4.4}$$

$$(1 - mask_j) \cdot C_2 + \sum_{m=1}^{|M^D|} x_{m,j} = |M^D|, \ \forall\, tag_j \in T_1 \tag{4.5}$$

$$mask_j \cdot C_3 + \sum_{m=1}^{|M^D|} x_{m,j} \ \leq \ |M^D|, \ \forall\, tag_j \in T \tag{4.6}$$

mit

$$\alpha \in \{0, 1\} \tag{4.7}$$

$$\forall j = 1, \ldots, l : \ mask_j \in \{0, 1\} \tag{4.8}$$

$$\forall j = 1, \ldots, l, \ \forall m = 1, \ldots, |M^D| : \ x_{m,j} \in \{0, 1\} \tag{4.9}$$

Das erhaltene ILP wird später in dieser Arbeit von einem Solver (vgl. Kap. 5.1.4) gelöst. Die Variablenbelegung der optimalen Lösung entspricht den Nachrichten-IDs. Im Kapitel 6.2 wird das Verfahren getestet und die Ergebnisse beschrieben.

Kapitel 5

Implementierung

Im Folgenden werden die Werkzeuge und die Details zur Umsetzung der in den Kapiteln 3 und 4 beschriebenen Probleme näher erläutert.

5.1 Verwendete Werkzeuge (Tools)

In diesem Kapitel werden die zur Implementierung der Arbeit die Programmiersprache Python kurz vorgestellt, ihre Geschichte, Eigenschaften, Vorteile und Nachteile beschrieben und die Unterschiede zu anderen Programmiersprachen verdeutlicht. Danach wird auf die Softwarekonfiguration eingegangen, sprich der Interpreter, sowie notwendige Bibliotheken und Pakete vorgestellt. Abschließend werden die Entwicklungsumgebung PyCharm und der Solver Gurobi näher beleuchtet.

5.1.1 Programmiersprache Python

Python wurde Anfang der 90er Jahre von Guido van Rossum entwickelt. Er ist und bleibt der Hauptautor von Python, seitdem viele weitere Beiträge von Anderen geschrieben worden sind. Das Logo der Sprache ist eine Schlange (Python), den Namen hat sie aber der britischen Komikergruppe Monty Python zu verdanken [54]. Im Jahr 2001 wurde die Python Software Foundation (PSF) gegründet, eine gemeinnützige Organisation, die speziell für den Besitz von geistigem Eigentum im Zusammenhang mit Python gegründet wurde. Alle Python-Versionen sind Open Source-Versionen [12]. Python besitzt eine umfangreiche, kostenlos zugängliche Dokumentation. Zahlreiche Bibliotheken, Anleitungen und Erweiterungsmodule sind im Internet ebenfalls kostenlos zu finden. Die Sprache ist portabel, da sie auf verschiedenen Systemen ausgeführt werden kann: Windows, macOS, Unix und Linux [54]. Mittlerweile ist Python sehr beliebt und eine der meist genutzten Programmiersprachen [57].

Python ist eine Programmiersprache auf hoher Ebene, die auf viele verschiedene Problemklassen angewendet werden kann. Sie ermöglicht effiziente abstrakte Datenstrukturen, dynamische Datentypisierung und verfolgt einen einfachen aber effektiven Ansatz zur objektorientierten Programmierung sowie zur Erstellung grafischer Oberflächen. Neben der Objektorientierung unterstützt Python auch die funktionale Programmierung, sodass die ProgrammiererInnen nicht zu einem einzigen Programmierstil gezwungen werden.

Python ist eine mächtige und einfach zu erlernende Skriptsprache mit einer eigenen integrierten Speicherverwaltung und guten Möglichkeiten, andere Programme aufzurufen und mit ihnen zusammenzuarbeiten. Python-Programme lassen sich als Module in Programme, geschrieben in anderen Sprachen, integrieren [48]. Das Hauptziel der Entwicklung von Python besteht in einem gut lesbaren, kompakten Programmtext. Elegante und einfache Syntax und die dynamische Semantik machen den geschriebenen Code verständlich, gut lesbar, wie z. B. Einrücken statt die Klammer-Syntax. Im Unterschied zu anderen Programmiersprachen findet in Python keine explizite Datentopzuordnung der Variablen statt, also auch keine Variablendeklaration. Der Datentyp ergibt sich automatisch aus dem Kontext. Eine automatische Typkonvertierung ist ebenfalls möglich [54]. Python elaubt es, viele Aufgaben mit wenig Code zu lösen. Der Einstiegsaufwand in die Sprache ist gering.

Eine weitere nützliche Python-Eigenschaft zeigt sich in dem interaktiven Modus, in dem einzelne Programmabschnitte eingegeben und die Ergebnisse direkt betrachtet werden können, ohne zuerst das vollständige Programm erstellen zu müssen. Der interaktive Modus ermöglicht es, etwas schnell auszuprobieren und in den früheren Entwicklungsstadien etwaige Fehler in den früheren Entwicklungsstadien zu erkennen, ohne sie bis zum Ende mitlaufen zu lassen (vgl. [55], [26]).

Eine weitere starke Eigenschaft von Python ist die Flexibilität. Als interpretierende Sprache, sowohl für Skripte als eine Erweiterungssprache für anpassbare Anwendungen, die in anderen Programmiersprachen geschrieben sind, ist Python hervorragend geeignet [26]. Python ist mit vielen Programmiersprachen verwandt [48], ist sehr beliebt und wird gerne in verschiedenen Branchen weltweit eingesetzt [11]. Die Python Programme werden in der Regel langsamer als vergleichbare Programme in Java, C oder C++ ausgeführt, dafür sind die in Python 3- bis 5 Mal so kurz wie in Java und 5- bis 10 Mal so kurz wie in C oder C++. Die Synatax von Python ist minimalistisch. Diese Python Eigenschaft ist ein bedeutender Vorteil, da größere Projekte im Team entwickelt werden, und die einfache Python-Syntax die Entwicklungszeit reduziert sowie die Lesbarkeit hervorhebt und dadurch die Notwendigkeit des Veränderns, Erweiterns und Verbesserns von Programmen reduziert. Java und Python zusammen bilden eine hervorragende Kombination: In Java entwickelte Komponenten können mit Python Anwendungen kombiniert werden; Python

kann auch zum Prototypen von Komponenten verwendet werden, bis diese in Java implementiert werden. Auch in C oder C++ geschriebene Komponenten können von Python um weiteren Funktionen und Datentypen erweitert werden. Im Vergleich zu Javascript ermöglicht Python das Schreiben größerer Programme und eine bessere Wiederverwendung von Code durch einen objektorientierten Programmierstil, bei dem Klassen und Vererbung eine wichtige Rolle spielen. Python und Perl besitzen einen ähnlichen auf Unix basierenden Hintergrund, allerdings ist Perl nicht für objektorientierte Programmierung geeignet. Diese Vergleiche beziehen sich nur auf Unterschiede zwischen den Sprachen. In der Praxis wird der Einsatz einer Programmiersprache oft durch andere Gründe wie z. B. Kosten, Verfügbarkeit oder vorherige Investitionen bestimmt (vgl. [3]).

Zahlreiche Linux Distributions haben ihre Installations- bzw. Systemverwaltungssoftware ganz oder zum Teil in Python geschrieben, wie z. B. Raspberry Pi. Unternehmen wie Google, Yahoo, YouTube, Dropbox verwenden Python intern. Zu den bekanntesten Python-Projekten gehören der Mailman (the GNU Mailing List Manager) und der Zope (Application Server) [12]. Python ist ein Teil von vielen Unternehmen und Institutionen auf der ganzen Welt und hat ein sehr breites Anwendungsgebiet [11].

5.1.2 Installierte Bibliotheken und Packages

Ein Python-Programm ist ein Text, auch Skript genannt, der von einem Interpreter ausgeführt wird [54]. Python-Programme werden nicht kompiliert sondern direkt interpretiert [52]. Dies ermöglicht einen schnellen Wechsel zwischen dem Schreiben von Code und der Testphase [52].

Python ist sehr stabil, derzeit erfolgen die Hauptveröffentlichungen in der Regel alle 18 Monate. Python 2.0 erschien im Oktober 2000. Python 2.X ist für viele Bibliotheken von Drittanbietern bekannt und obwohl es noch weit verbreitet ist, wird es ab dem 1. Januar 2020 nicht mehr gewartet. Python 3.0 (auch Python 3000) ist im Dezember 2008 erschienen. Python 3.7.2 ist die aktuelle Version und sie verfügt über eine große Standardbibliothek (Application Programming Interface, API), die solche Bereiche wie Zeichenfolgeverarbeitung, Internetprotokolle, Software Engineering und Betriebssystemschnittstellen umfasst und zahlreiche Pakete (Python Package Index, PyPI) [48], die eine leichte Erweiterbarkeit möglich machen. Die zahlreiche Erweiterungen von Drittanbietern werden von dieser Version ebenfalls unterstützt [12]. Alle Bibliotheken sind kostenlos und sind für alle gängigen Entwicklungsplattformen verfügbar; sie können frei weiterverarbeitet werden.

Für diese Arbeit sind verschiedene Konfigurationen ausprobiert worden, bis eine endgültige für die Problemstellung dieser Arbeit gut funktionierende gefunden wurde.

Für die Implementierung sind Python-Interpreter Version 3.6.3 (64-Bit) für Windows Release vom 3.10.2017 [10] und die folgenden Bibliotheken bzw. Pakete erforderlich:

- *cython* (Version 0.29.6) - ist ein optimierender statischer Compiler sowohl für Python als auch für die erweiterte Programmiersprache Cython, die auf Pyrex basiert. Es gestaltet das Schreiben von C-Erweiterungen für Python einfacher [45].
- *gurobipy* (Version 5.0.2) - ist der State-of-the-Art Löser für mathematische Programme [7]. In dieser Arbeit wird er zum Lösen von ILP eingesetzt, mehr dazu im Abschnitt 5.1.4.
- *pip* (Version 19.0.3) - ist ein Package Manager für Python, der Installation aus dem zentralen Python-Paketpool, genannt Package Index(PyPI), erleichtert [17].
- *pygurobi* (Version 0.5) - ist eine erweiterte Gurobi-Bibliothek, die eine schnelle interaktive Modellierung und Analyse von Gurobi-Modellen mit insgesamt weniger Codeaufwand erlaubt [42].
- *setuptools* (Version 28.8.0) - ist eine stabile Bibliothek, die das Erstellen und die Installation von Python-Modulen erleichtert [16].
- *virtualenv* (Version 16.4.3) - ist eine virtuelle Umgebung. Sie hilft Abhängigkeiten beizubehalten, wenn an verschiedenen Projekten gleichzeitig gearbeitet wird [21].
- *numpy* (Version 1.16.2) - ist das Basispaket für wissenschaftliches Rechnen mit Python [19].
- *matplotlib* (Version 3.0.3) - ist eine 2D Plottbibliothek, die Daten in guter Qualität erstellt [29].

5.1.3 Entwicklungsumgebung

Ein Python-Programm ist eine einfache Textdatei oder ein Skript, das mit einem beliebigen Texteditor bearbeitet werden kann. Von ProgrammiererInnen werden sogenannte Editoren oder Entwicklungsumgebungen (engl. Integrated Development Environment, IDE) zum Entwickeln und Debuggen von Programmen benutzt. Diese besitzen viele hilfreiche unterstützende Funktionen, die ein Texteditor nicht hat. Es existiert eine Reihe von Python-Entwicklungsumgebungen: Multiplattformeditoren, für einzelne Betriebssysteme geeignete Editoren, Onlineeditoren etc. [13]. Die bekannten IDEs, die Python unterstützen, sind z.B. Anaconda [2], Komodo [1] und Spyder [15]. Die Entscheidung, welche Entwicklungsumgebung am besten geeignet ist, wird durch die individuelle bzw. subjektive Wahrnehmung

bestimmt. Jede Umgebung bietet nützliche Werkzeuge an, die bei einer Code-Entwicklung helfen. Übliche integrierte Features sind z. B. Hervorhebung der Syntax, Debugging und Versionsverwaltung.

Diese Arbeit nutzt die Umgebung PyCharm Professional [8] Version: 2019.1, Build: 191.6183.50, Released: 27.03.2019, die sich insbesondere durch einen guten Smart-Editor auszeichnet.

PyCharm ist eine integrierte Entwicklungsumgebung von JetBrains [9], die speziell für das Programmieren in Python entwickelt wurde. JetBrains verfügt über mehrere IDEs für verschiedene Sprachen, wie z. B. RubyMine für Ruby, CLion für C++. PyCharm unterstützt neben Python weiteren Sprachen, wie z. B. Javascript, HTML und Node.js. Die PyCharm Umgebung kann unter Windows-, macOS-, Unix- und Linux-Betriebssystemen laufen. PyCharm ist in zwei Varianten erhältich: die kostenfreie Community Edition und die kostenpflichtige Professional Edition mit einer 30-Tage-Testversion. Die Community Edition enthält Features wie automatische Codevervollständigung, Codeanalysen, Refactorings, Debugger, Testrunner und vitualenv-Support. Die Professional Edition bietet zusätzliche Features, wie die Webframeworks Django, Pyramid, Flask und web2py, Googles Platform as a Service App Engine sowie den objektrelationalen Mapper SQLAlchemy. Die Softwareinstallation erfordert keine speziellen Kenntnisse.

Die Entwicklungsumgebung PyCharm unterstützt in einer einzigen, einheitlichen Oberfläche folgende Funktionen (vgl. [30]):

- Syntax-Hervorhebung
- Schriftarten und Farbdarstellung lassen sich anpassen
- Code-Vervollständigung
- Code Refactoring und -Generierung
- Codeanalyse und Verhinderung von Code-Duplikation
- integrierte Testumgebung: ermöglicht die Ergebnisse sofort zu betrachten
- Tools zur Versionskontrolle: erlaubt verschiedene Interpreter Versionen auszuwählen
- Projektnavigation: eignet sich gut für größere Projekte, bietet bessere Projektübersicht
- Keymap: ermöglicht Tastenkombinationen zum schnellen Navigieren einzustellen
- Auto-Import: erlaubt Installation von Modulen aus PyCharm heraus

- Integrierter Debugger: ermöglicht einfache Fehlersuche direkt im Editor
- Warnungen bei nicht-vorhandenen Methoden
- Fehler werden automatisch rot markiert
- leistungsstarke interaktive Python-Konsole
- Konfigurierbare Injektion anderer Sprachen
- unterstützt viele sprachspezifische Plugins
- unterstützt wissenschaftliche Anwendungen, wie Anaconda, Numpy, Matplotlib
- unterstützt Webtechnologien, wie JavaScript, CoffeeScript, TypeScript, Cython, SQL, HTML/CSS, template languages, AngularJS, Node.js
- unterstützt Datenbanken, wie Oracle, SQL Server, PostgreSQL, MySQL.

Diese und weitere Funktionen gestalten die Entwicklung von Programmiercode wesentlich leichter. PyCharm stellt einen ein leistungsfähiger, sehr mächtiger und empfehlenswerter Editor dar. Er ist zugänglich und einfach zu bedienen. Diese Entwicklungsumgebung eignet sich gut sowohl für AnfängerInnen als auch für erfahrene, professionelle ProgrammiererInnen.

5.1.4 Verwendete Solver: Gurobi

Gurobi Optimizer ist ein hochmoderner Löser (engl. Solver) für mathematische Programme und wurde 2008 von der Gurobi GmbH entwickelt. Der Name setzt sich aus den Anfangsbuchstaben der Nachnamen der drei Gründer zusammen: Zonghao Gu, Edward Rothberg und Robert Bixby [51]. Das Tool ist kostenpflichtig; es besteht die Möglichkeit eine kostenlose akademische Lizenz zu beantragen; auch eine kostenlose Testversion steht zur Verfügung. Die Software wird regelmäßig gewartet und vom Support unterstützt [7].

Gurobi ist robust, skalierbar und zuverlässig. Der Solver ermöglicht schwierige und komplexe Probleme mathematisch zu formulieren und ist in der Lage, mit Millionen von Entscheidungsvariablen umzugehen und betrachtet dabei Milliarden von verschiedenen möglichen Lösungen, um die beste zu finden. Die Löser von Gurobi Optimizer wurden für die Arbeit mit modernen Architekturen und Multi-Core-Prozessoren konzipiert und nutzen die fortschrittlichsten Implementierungen aktueller Algorithmen. Das erlaubt Nutzern eine deutlich höhere Flexibilität beim Modellaufbau. Außerdem haben sie so die Möglichkeit, ihr Modell noch komplexer zu gestalten, um die Realität der Probleme, die sie lösen, noch besser abzubilden. Gurobi beinhaltet die folgenden Löser zu den allen wichtigen Problemklassen(vgl. [6]):

- Löser für die lineare Programmierung (engl. Linear Programming, LP)
- Löser für die gemischt-ganzzahlige lineare Programmierung (engl. Mixed-Integer Linear Program, MILP)
- Löser für die gemischt-ganzzahlige quadratische Programmierung (engl. Mixed-Integer Quadratic Programming, MIQP)
- Löser für die quadratische Programmierung (engl. Quadratic Programming, QP)
- Löser für die Programmierung mit quadratischen Nebenbedingungen (engl. Quadratically Constrained Programming, QCP)
- Löser für die gemischt-ganzzahlige Programmierung mit quadratischen Nebenbedingungen (engl. Mixed-Integer Quadratically Constrained Programming, MIQCP).

In den letzten Jahren haben die Löser von Gurobi Optimizer die Leistung der MIP-Solver um den Faktor 43 und der LP-Solver um den Faktor 6 gesteigert. Eine feste Größenbegrenzung existiert nicht. Ohne besondere Schwierigkeiten löst Gurobi sogar auf Notebooks und Desktop-PCs größere und schwierigere Modelle mit Millionen von Variablen und Nebenbedingungen. Neben der hervorragenden Performance bietet Gurobi weitere Features, die es einfacher erlauben, Modelle zu erstellen und sie in kürzerer Zeit zu lösen, wie z. B. Werkzeuge für verteilte Optimierung oder Optimierung in der Cloud [6]:

Der Löser kann sowohl von erfahrenen ExpertenInnen als auch voll automatisiert ohne menschlichen Einsatz verwendet werden. Es ist möglich Gurobi zu bestehenden Tools hinzuzufügen oder aber komplett auf Gurobi umsteigen. Gurobi-Interfaces sind als moderne Programmierschnittstelle (engl. Application Programming Interface, API) implementiert und können intuitiv genutzt werden, was einen einfachen Einstieg ermöglicht. Die Schnittstellen sowie die Bibliotheken arbeiten schnell und benötigen vergleichsweise wenig Speicher. Gurobi unterstützt folgende Schnittstellen (vgl. [6]):

- Objektorientierte Schnittstellen für C++, Java, .NET und Python
- Matrixorientierte Schnittstellen für C, MATLAB® und R
- Verknüpfungen mit Standard-Modellierungssprachen: AIMMS, AMPL, GAMS und MPL
- Verknüpfungen mit Excel via Premium Solver Platform und Risk Solver Platform.

Der Gurobi Optimizer wird von über 1600 Firmen in knapp zwei Dutzend Branchen eingesetzt, wie z. B. Autoindustrie, Telekommunikation, Industrielle Automatisierung, Biotechnologie, Medizin und Pharmazeutik, und hat zu messbaren Verbesserungen beigetragen [5]. Einige Fallbeispiele zum Gurobi Einsatz [7]:

- Hewlett Packard Global IT: Optimierung der Auswahl und Terminplanung von Projekten
- National Institute for Environmental Studies: Löst komplexe Modelle in nur wenigen Stunden, die mit dem vorher verwendeten Löser nur mit enormen Vereinfachungen überhaupt gelöst werden konnten
- ICRON: Optimierung der End-to-End-Lieferkette für Vestel
- Collaborative Research Center for Energy Engineering: Optimierung des Strompreises für den folgenden Tag mit Hilfe von Wettervorhersagen
- Die Deutsche Bahn: Lösen von gigantische Optimierungsprobleme rund um Fahrplan, Fahrzeiten und Umlaufplanung für Loks.

Es existiert eine Reihe weiterer ILP-Solver, wie z.B. GNU Linear Programming Kit (GLPK) [4] oder Mixed Integer Linear Programming (MILP) Solver lp_solve [14].

Der GLPK-Solver ist ein kostenloser Solver und wurde für die Lösung von umfangreichen LP, MIP und anderen verwandten Problemen entworfen. GLPK stellt eine Bibliothek dar, die in ANSI C geschrieben ist [4].

lp_solve repräsentiert einen kostenfreien Löser für MILP, der auf der überarbeiteten Simplex- und der Branch-and-Bound-Methode für ganze Zahlen basiert. Der Löser ist für alle Betriebssysteme erhältlich. lp_solve ist eine Bibliothek, die von fast jeder Programmiersprache aus aufgerufen werden kann [14].

Diese Arbeit setzt den Gurobi Optimizer (Version 8.1.0 für Windows 64-Bit) [7] zum Lösen von ILP-Problemen ein, da die Performance und die Einfachheit von Gurobi überzeugt.

5.2 Implementierungsdetails zum Design der Filterkonfiguration

Das Kapitel 3 stellte das Problem des Designens der Filterkonfiguration für einen Broadcast-Bus dar. In diesem Kapitel geht es um die praktische Umsetzung des Problems mittels der Programmiersprache Python. Die wichtigsten Ausschnitte des Programmcodes werden im Folgenden vorgeführt. Die Funktionsweise der einzelnen Blöcke wird detailliert erklärt. Die zu testenden Daten werden im Abschnitt 6.1.1 vorgestellt und genau betrachtet. Die Ergebnisse dieser Experimente sind im Abschnitt 6.1.2 zu finden. Eine umfassende Diskussion dieser Forschungsresultate steht in dem Abschnitt 6.1.3 zur Verfügung.

Als Erstes wird das Format für die Eingabedaten festgelegt. Das Format gilt für alle Experimente, die das Design der Filterkonfiguration betrifft und in dem Abschnitt 6.1.1 beschrieben sind. Jede Nachricht ist ein Tupel aus drei Elementen:

- das erste Element das Tupel stellt die Nachrichten-ID dar,
- das zweite Element das Tupel ist eine (ganze) Zahl, die die Periode der jeweiligen Nachricht in Millisekunden darstellt,
- das dritte Element zeigt an, ob die Nachricht zu den Desired Messages oder den Undesired Messages gehört.

In Python ist dieses Tupel durch den Datentyp *List* realisiert:

- das erste Element in der Liste ist vom Typ *String*,
- das zweite Element in der Liste ist vom Typ *Integer*,
- das dritte Element in der Liste ist vom Typ *Boolean*. Wenn die jeweilige Nachricht zu den Desired Messages gehört, dann ist der Wert *True*. Wenn die Nachricht zu den Undesired Messages gehört, dann bedeutet der Wert dieser Variable *False*.

Ein Datensatz besteht aus mehreren Nachrichten. In Python ist der Datensatz eine Liste, die sich aus den einzelnen Listen zusammensetzt. Dieser Datensatz wird durch die Variable **id_liste** repräsentiert. Somit sieht ein Beispieldatensatz folgendermaßen aus:

```
id_liste = [
    ["00011000001", 1000, False],
    ["00011001000", 1000, True ], #desired
    ["00011001001", 100,  True ], #desired
    ["10010001000", 100,  False],
    ["10011001000", 100,  True ], #desired
    ["10011001001", 10,   True ]  #desired
]
```

Dieser Beispieldatensatz besteht aus 6 Nachrichten, von denen vier Nachrichten aus der Menge der Desired Messages sind. Diese Nachrichten sind durch den Wert **True** an der dritten Position in der **Liste** zu erkennen. Zusätzlich werden die Nachrichten durch die Kommentarfunktion in Python zum Verdeutlichen markiert.
Es ist möglich, solche Daten automatisch zu generieren. Dabei wir die gesamte Anzahl der Nachrichten sowie die Anzahl der Desired Messages festgelegt. Entsprechend werden die restlichen Nachrichten als Undesired Messages definiert. Die Periode kann auch zufällig erzeugt werden. Da dies keine Herausforderung im Programmieren darstellt, sondern reines Anwenden eines Werkzeugs ist, wird an diese Stelle kein Code zu dem Thema gezeigt.

Als Nächstes wird der Code vorgestellt, der die im Abschnitt 3.2 beschriebene Summe berechnet. Die Summe wird für jede einzelne Bit-Position separat ermittelt. Es werden nur die Nachrichten-IDs der Desired Messages in Betracht gezogen. Zuerst wird die Variable Summe **sum** definiert. Die Summe ist eine Liste aus 11 Elementen. Wenn die Nachrichten-ID 29-stellig ist, dann ist dies an dieser Stelle zu beachten. Jedes Element ist eine ganze Zahl (*integer*). Bei der Initialisierung ist jedes einzelne Element der Summe auf 0 gesetzt. Jedes Element der Variable **sum** stellt die Summe einzelner Bits der Nachrichten-IDs dar: An der Postion sum_i wird die Summe über die Nachrichtenbits gebildet, die an der Stelle i vorkommen. Die äußere *for*-Schleife läuft alle einzelne Nachrichten durch. Die Länge der Liste ist nicht festgelegt und die Anzahl der Nachrichten darf beliebig sein. Die *if*-Anweisung überprüft, ob die jeweilige Nachricht aus der Menge der Desired Messages stammt. Wenn das der Fall ist, dann wird diese Nachricht in der zweiten *for*-Schleife weiter betrachtet. Die zweite *for*-Schleife berücksichtigt keine Undesired Messages und berechnet die jeweiligen Elemente der Variable **sum** an der Position j einzeln. Wenn die Nachrichten-ID 29-stellig ist, dann muss dies ebenfalls an der Stelle beachtet:

```
sum = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

for i in range(0, len(id_liste)):
    if id_liste[i][2]:
        for j in range(0, 11):
            sum[j] = sum[j] + int(id_liste[i][0][j:j + 1])
```

Im Folgenden wird verdeutlicht, wie Maske und Tag berechnet werden, die zusammen eine Filterkonfiguration bilden. Im Abschnitt 3.2 ist genau erklärt, wie anhand der Variable **sum** die Maske und der Tag berechnet werden daher wird an dieser Stelle nicht darauf eingegangen. Maske und Tag einer Filterkonfiguration werden durch die Variablen **mask** und **tag** dargestellt. Die Maske zeigt an, welcher Bit des Tags zu betrachten ist. Die Variable **filter** repräsentiert die Filterkonfiguration. In diesem Code ist die Variable **filter** nur zum Zweck der Veranschaulichung der Lösung und besitzt keine eigentliche Funktion. Zum Beginn werden die drei Variablen **mask**, **tag** und **filter** als Listen aus 11 Elementen definiert. Wenn die Nachrichten-ID 29-stellig ist, dann muss dieser Umstand ebenfalls an der Stelle zu beachten. Die Variable **des_counter** ist bekannt. Sie stellt die Anzahl der Desired Messages im System dar. Im Dritten *if*-Block des Codes wird der Variable **tag** an der Position i zufällig ein Wert 0 oder 1 zugewiesen. An der Position i ist die Maske gleich 0, deswegen erweist sich der Wert des Tags an der Postion i als irrelevant. Die Variable **filter** bekommt an der Position i den Wert **don't care**, dargestellt in Python durch den Stern. Der folgende Code bestimmt die Maske und den Tag:

```
mask =    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
tag =     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
filter = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

for i in range(0, 11):
    if sum[i] ==0:
        mask[i] = 1
        tag[i] = 0
        filter[i] = 0
    elif sum[i] == des_counter:
        mask[i] = 1
        tag[i] = 1
        filter[i] = 1
    else:
        mask[i] = 0
        tag[i] = random.randint(0,1)
        filter[i] = "*"
```

Zuletzt wird die Güte der gefundenen Filterkonfiguration berechnet, welche anhand der Formel (3.1) berechnet. Diese Formeln ist im Kapitel 3.1 vorgestellt und genau erklärt. Die Variable **QoF** berechnet die Qualität der Filterkoniguration. Zum Beginn ist sie mit 0 initialisiert. Die Variable **filter_pas** berechnet die Anzahl der Undesired Messages, die den Filter passieren. Die erste *for*-Schleife betrachtet alle Desired Messages und Undesired Messages. Die *if*-Anweisung überprüft, ob die Nachricht aus der Menge der Desired Messages stammt. Die Variable **temp** ist eine Hilfsvariable und speichert die Anzahl der Nachrichtenbits zwischen. Wenn der Wert dieser Variable zum Schluss gleich 11 ist, dann darf die Nachricht den Filter passieren, obwohl diese Nachricht aus Undesired Messages ist. Der Zähler **filter_pas** wird um eins erhöht und die Variable **QoF** wird neu bewertet. Die zweite *for*-Schleife vergleicht Bit für Bit die Nachrichten-ID mit dem Filter und berechnet den Wert der Variable **temp**. Der Code dazu sieht folgendermaßen aus:

```
QoF=0
filter_pas = 0

for i in range(0, len(id_liste)):
    if not id_liste[i][2]:
        temp = 0
        for j in range(0, 11):
            if filter[j] == "*":
                temp += 1
```

```
            elif str(id_liste[i][0][j:j + 1]) == str(filter[j]):
                temp += 1

        if temp == 11:
            filter_pas += 1
            QoF += 1/int(id_liste[i][1])
```

5.3 Implementierungsdetails zum Design der Nachrichten

Das Kapitel 4 präsentiert das Problem des Designens der Nachrichten für einen Broadcast-Bus, wobei in diesem Abschnitt die Realisierung betrachtet wird. Wie im vorherigen Abschnitt wird die Programmiersprache Python (vgl. Kap. 5.1.1) sowie der mächtige ILP-Löser Gurobi (vgl. Kap. 5.1.4) eingesetzt. Das ILP Problem ist komplex und es ist daher nicht möglich, auf jedes einzelne Detail einzugehen, da dies den Rahmen der Arbeit sprengen würde. Dennoch werden die grundlegenden Codeausschnitte im Folgenden gezeigt.

Wie bereits erwähnt, sind lineare Programme komplex. Nichtsdestotrotz hat diese Arbeit den Versuch unternommen, die Idee aus dem Kapitel 4 auf verständliche und einfache Weise zu präsentieren.

Die Tabelle 6.4 stellt die Maske und den Tag dar:

ID	Bit_0	Bit_1	Bit_2	Bit_3	Bit_4	Bit_5	Bit_6	Bit_7	Bit_8	Bit_9	Bit_{10}
Maske	0	1	1	1	1	1	1	1	1	1	0
Tag	0	0	0	1	1	0	0	1	0	0	1

Tabelle 5.1: Die Maske und der Tag für ILP

Das Modell wird mit Hilfe von Gurobi aus der Datei namens *„ilp_problem.lp“* ausgelesen und die Funktion *„m.optimize()“* startet das ILP.

```
import sys
from gurobipy import *
import gurobipy as pg

m = pg.read('ilp_problem.lp')

m.optimize()
```

Die Datei *„ilp_problem.lp“* sieht wie nachfolgend beschrieben aus:

```
Maximize
x[1] + x[2] + x[3] + x[4]

Subject To
 nb_dummy(0): x[1] + x[2] + x[3] + x[4] = 4
```

Es ist noch zu neachten: Diese Arbeit nicht an dem zu optimierenden Wert interessiert, sondern an seiner Variablenbelegung. Daher ist die Zielfunktion (ref4.3) eine sogenannte Platzhalterfunktion.

Die Nebenbedingung (4.4) heißt im Model $nb_1(i)$, die Big-M-Zahl C_1 ist auf 1000 gesetzt. Der Code dazu sieht folgendermaßen aus:

```
nb_1(0): 1000 (1 - mask[0]) + x[1,0] + x[2,0] + x[3,0] + x[4,0]  = 0
nb_1(1): 1000 (1 - mask[1]) + x[1,1] + x[2,1] + x[3,1] + x[4,1]  = 0
nb_1(2): 1000 (1 - mask[2]) + x[1,2] + x[2,2] + x[3,2] + x[4,2]  = 0
nb_1(3): 1000 (1 - mask[3]) + x[1,3] + x[2,3] + x[3,3] + x[4,3]  = 0
nb_1(4): 1000 (1 - mask[4]) + x[1,4] + x[2,4] + x[3,4] + x[4,4]  = 0
nb_1(5): 1000 (1 - mask[5]) + x[1,5] + x[2,5] + x[3,5] + x[4,5]  = 0
nb_1(6): 1000 (1 - mask[6]) + x[1,6] + x[2,6] + x[3,6] + x[4,6]  = 0
nb_1(7): 1000 (1 - mask[7]) + x[1,7] + x[2,7] + x[3,7] + x[4,7]  = 0
nb_1(8): 1000 (1 - mask[8]) + x[1,8] + x[2,8] + x[3,8] + x[4,8]  = 0
nb_1(9): 1000 (1 - mask[9]) + x[1,9] + x[2,9] + x[3,9] + x[4,9]  = 0
nb_1(10):1000 (1 - mask[10])+ x[1,10]+ x[2,10]+ x[3,10]+ x[4,10] = 0
```

Ähnlich werden die Nebenbedingung (4.5) als $nb_2(i)$ und die Nebenbedingung (4.6) als $nb_3(i)$ formuliert.

Die Big-M-Zahlen C_2 und C_3 bekommen dementsprechend die Werte 2000 und 3000. Der Code dazu:

```
nb_2(0):  2000 (1 - mask[0]) + x[1,0] + x[2,0] + x[3,0] + x[4,0] = 4
nb_2(1):  2000 (1 - mask[1]) + x[1,1] + x[2,1] + x[3,1] + x[4,1] = 4
nb_2(2):  2000 (1 - mask[2]) + x[1,2] + x[2,2] + x[3,2] + x[4,2] = 4
nb_2(3):  2000 (1 - mask[3]) + x[1,3] + x[2,3] + x[3,3] + x[4,3] = 4
nb_2(4):  2000 (1 - mask[4]) + x[1,4] + x[2,4] + x[3,4] + x[4,4] = 4
```

```
nb_2(5):   2000 (1 − mask[5]) + x[1,5] + x[2,5] + x[3,5] + x[4,5] = 4
nb_2(6):   2000 (1 − mask[6]) + x[1,6] + x[2,6] + x[3,6] + x[4,6] = 4
nb_2(7):   2000 (1 − mask[7]) + x[1,7] + x[2,7] + x[3,7] + x[4,7] = 4
nb_2(8):   2000 (1 − mask[8]) + x[1,8] + x[2,8] + x[3,8] + x[4,8] = 4
nb_2(9):   2000 (1 − mask[9]) + x[1,9] + x[2,9] + x[3,9] + x[4,9] = 4
nb_2(10):  2000 (1 − mask[10])+ x[1,10]+ x[2,10 + x[3,10]+ x[4,10]= 4


nb_3(0):   3000 mask[0]  + x[1,0]  + x[2,0]  + x[3,0]  + x[4,0]  <= 4
nb_3(1):   3000 mask[1]  + x[1,1]  + x[2,1]  + x[3,1]  + x[4,1]  <= 4
nb_3(2):   3000 mask[2]  + x[1,2]  + x[2,2]  + x[3,2]  + x[4,2]  <= 4
nb_3(3):   3000 mask[3]  + x[1,3]  + x[2,3]  + x[3,3]  + x[4,3]  <= 4
nb_3(4):   3000 mask[4]  + x[1,4]  + x[2,4]  + x[3,4]  + x[4,4]  <= 4
nb_3(5):   3000 mask[5]  + x[1,5]  + x[2,5]  + x[3,5]  + x[4,5]  <= 4
nb_3(6):   3000 mask[6]  + x[1,6]  + x[2,6]  + x[3,6]  + x[4,6]  <= 4
nb_3(7):   3000 mask[7]  + x[1,7]  + x[2,7]  + x[3,7]  + x[4,7]  <= 4
nb_3(8):   3000 mask[8]  + x[1,8]  + x[2,8]  + x[3,8]  + x[4,8]  <= 4
nb_3(9):   3000 mask[9]  + x[1,9]  + x[2,9]  + x[3,9]  + x[4,9]  <= 4
nb_3(10):  3000 mask[10] + x[1,10] + x[2,10] + x[3,10] + x[4,10] <= 4
```

Der Fall, indem die Maske gleich 0 ist, wird anhand der Nebenbedingungen formuliert:

```
nb_3(0):   mask[0]  = 0
nb_3(1):   mask[1]  = 0
nb_3(2):   mask[2]  = 0
nb_3(3):   mask[3]  = 0
nb_3(4):   mask[4]  = 0
nb_3(5):   mask[5]  = 0
nb_3(6):   mask[6]  = 0
nb_3(7):   mask[7]  = 0
nb_3(8):   mask[8]  = 0
nb_3(9):   mask[9]  = 0
nb_3(10):  mask[10] = 0
```

Aufgrund der Komplexität der Lösung, werden die weiteren Fallunterscheidungen in dieser Arbeit nicht visualisiert.

Die letzte dennoch bedeutende Nebenbedingung sagt aus, dass die Variablenbelegung $x_{i,j} \in \{0,1\}$. Dies wird nun in gekürzter Form gezeigt:

```
 gb(1):   x[1,0]   <= 1
 gb(2):   x[2,0]   <= 1
 gb(3):   x[3,0]   <= 1
 gb(4):   x[4,0]   <= 1
 gb(5):   x[1,1]   <= 1
 gb(6):   x[2,1]   <= 1
 gb(7):   x[3,1]   <= 1
 gb(8):   x[4,1]   <= 1
  ...
 gb(37):  x[1,9]   <= 1
 gb(38):  x[2,9]   <= 1
 gb(39):  x[3,9]   <= 1
 gb(40):  x[4,9]   <= 1
 gb(41):  x[1,10]  <= 1
 gb(42):  x[2,10]  <= 1
 gb(43):  x[3,10]  <= 1
 gb(44):  x[4,10]  <= 1
Bounds
End
```

Aus der von Gurobe erhaltenen Lösung kann die Variablenbelegung durch folgenden Code erfragt werden:

```
for v in m.getVars():
    print('%s %s' % (v.VarName, v.X))
print('Obj: %s' % m.ObjVal)
```

Kapitel 6

Evaluation

In diesem Kapitel werden Experimente beschrieben und deren Ergebnisse ausgewertet. In dem Abschnitt 6.1 beziehen sich die Forschungen auf das Design der Filterkonfiguration, das im Kapitel 3 beschrieben ist. Das ist der heuristischer Ansatz, dessen Implementierung im Kapitel 5.2 betrachtet wurde. Der Abschnitt 6.2 beschäftigt sich mit den Experimenten, die das Problem des Designs der Nachrichten betrachten. Das Problem wird mittels Integer Linear Programming 2.3 gelöst. Die genaue Beschreibung des Problem ist im Kapitel 4 vorgestellt. Die Implementierung ist im Abschnitt 5.3 erläutert.

Die Autorin dieser Arbeit hat alle Experimente auf eigenem PC Notebook HP EliteBook 2506p durchgeführt. Das Notebook hat folgende Konfiguration (Performance):

- Windows 10 Pro Version 1809, 64-Bit-Betriebssystem
- Prozessor Intel(R) Core(TM) $i7 - 2620M$ CPU 2.70 GHz 2.70 GHz
- Arbeitsspeicher (RAM) 12 GB
- Festplatte Samsung SSD 840 EVO, 250 GB.

6.1 Design der Filterkonfiguration

Dieser Abschnitt betrachtet die Experimente, bei denen anhand der Heuristik aus dem Kapitel 3 die Filterkonfiguration bestimmt wird. Die Heuristik ist in der Programmiersprache Python implementiert und im Abschnitt 5.2 detailliert beschrieben. Zuerst werden im Abschnitt 6.1.1 die zu testende Datensätze vorgestellt. Im Abschnitt 6.1.2 werden die Ergebnisse vorgetragen, und im Abschnitt 6.1.3 wird über die gewonnene Ergebnisse diskutiert.

6.1.1 Experimenteller Aufbau

Dieser Abschnitt stellt die Testdaten der Experimente zum Design der Filterkonfiguration vor. Es gibt 4 unterschiedlichen Dateneingaben. Der erste Datensatz besteht aus 6 Nachrichten, der zweite Datensatz aus 55 Nachrichten, der dritte Datensatz aus automatisch generierten Datensätzen und der vierte Datensatz stellt Eingabedaten vor, die eine perfekte Filterkonfiguration für diese Heuristik garantieren.

Beispiel mit 6 Nachrichten

Der erste Datensatz besteht aus 6 Nachrichten. Die Nachrichten sind in der Tabelle 6.1 zusammengetragen und stammen aus [40]. Dabei handelt es sich um dieselben Nachrichten, die in dem Beispiel 3.3 vorher betrachtet worden sind. Zwei der Nachrichten sind aus der Menge der Undesired Messages und vier der Nachrichten sind aus der Menge der Desired Messages. Die Periode der Nachrichten wurde zufällig von der Autorin dieser Arbeit gewählt, da sie in diesem Fall keine Rolle spielt, da die Filterkonfiguration perfekt ist und keine Undesired Messages die Filterkonfiguration passieren. Die Nachrichten sind im Binärformat und die Periode ist in Millisekunden dargestellt.

ID (binär)	Periode (ms)	desired
00011000001	1000	
00011001000	1000	ja
00011001001	100	ja
10010001000	100	
10011001000	1000	ja
10011001001	10	ja

Tabelle 6.1: Beispiel mit 6 Nachrichten

Epsilon Benchmark

In diesem Experiment stammt der Datensatz aus [28] und ist in der Tabelle 6.2 dargestellt. Es sind insgesamt 55 Nachrichten, 11 davon Desired Messages und 44 Undesired Messages. Die Nachrichten-IDs sowie dazugehörigen Perioden stammen aus einem realen System (Epsilon Electric Vehicles), sie sind nicht zufällig gewählt oder generiert. Die Nachrichten sind im Binärformat und die Periode ist in Millisekunden dargestellt.

SAE Benchmark

In diesem Experiment besteht der Datensatz aus insgesamt 51 Nachrichten. Die Nachrichten-IDs werden zufällig generiert. Die Perioden entsprechen denen, die in realen Systemen

ID (binär)	Periode (ms)	desired	ID (binär)	Periode (ms)	desired
00000001010	1000		01011110010	100	
00000010000	1000	ja	01100000000	1000	
00000100000	100		01101010000	1000	
00001100000	100		01110101100	100	
00001101010	100	ja	10000000000	100	
00001101110	10		10000010000	100	
00001110001	100		10000010001	100	
00001111000	100		10000100000	1000	
00001111101	100		10000100101	1000	
00010000001	1000	ja	10101100101	500	
00010001100	100		11000010000	100	
00010010001	100		11000010001	1000	
00010010110	500		11000010010	100	
00100000000	10		11000010011	1000	ja
00100000001	10		11000010100	1000	
00100000010	10		11000011000	100	
00100010000	10		11000100000	100	ja
00100100000	10		11011110000	1000	ja
00100110000	10		11011111000	1000	
00101010000	10		11100000000	1000	
00101100000	1000		11100000010	1000	
00101100001	1000		11100010000	1000	ja
00000000000	1000		11100010001	1000	
01000000001	10		11100100000	1000	
01000010000	10	ja	11100110000	1000	ja
01000100000	10	ja	11101110000	1000	ja
01011110000	10		11101110001	1000	
01011110001	100				

Tabelle 6.2: Network Specification of Battery Electric Vehicle [28]

(Strategic Applications Engineering, SAE) verwendet werden [40] und werden zufällig den Nachrichten-IDs zugewiesen.

Dieses Experiment unterscheidet sich von den anderen, da es 50 Mal wiederholt wird, um die Durchschnittswerte zu berechnen. Auch die Anzahl der Nachrichten aus der Menge der Desired Messages wird variiert. Zuerst wird das Experiment 50 Mal mit 5 Desired Messages durchgeführt. Dann erhöht sich die Anzahl der Desired Messages um weitere 5. Insgesamt

wird das Experiment je 50 Mal für 5, 10, 15, 20, 25, 30 und 35 Desired Messages wiederolt. Die durchschnittliche Laufzeitwerte und die durchschnittlichen Werte für die Qualität der Filterkonfiguration werden betrachtet und analysiert.

Der zweite Teil der Experimenten aus dieser Gruppe ist ähnlich aufgebaut, nun werden hier die Durchschnittswerte für 1, 2, 3, 4 und 5 Nachrichten aus der Menge der Desired Messages berechnet.

Da der Datensatz in dieser Gruppe der Experimente sehr groß ist, wird er nicht in einer Tabelle visualisiert. Stattdessen werden die Ergebnisse grafisch dargestellt und im Abschnitt 6.1.2 präsentiert.

Perfekte Filterkonfiguration

Die Daten in diesem Experiment sind auf geschickte Art und Weise generiert worden, sodass die Filterkonfiguration perfekt ist. Die Nachrichten-IDs basieren auf dem Epsilon Benchmark Experimenten (vgl. Kap. 6.1.1) und die Perioden sind ebenfalls aus dem Experiment übernommen. Es sind insgesamt 55 Nachrichten, von denen 11 Nachrichten Desired Messages sind und 44 Nachrichten Undesired Messages. Die Periode spielt hier keine Rolle, da die Filterkonfiguration perfekt ist und keine Undesired Messages den Filter passieren.

Was bedeutet es nun, dass die Nachrichten „geschickt generiert" sind? Das bedeutet, dass alle Nachrichten aus der Menge der Desired Messages eine bestimmtes Muster haben sollten. Gleichzeitig bedeutet es auch, dass alle Nachrichten aus der Menge der Undesired Messages dieses Muster nicht haben dürfen.

Die Autorin dieser Arbeit legt willkürlich ein solches Muster fest, das die obigen Bedienungen erfüllt: In diesem Datensatz sollen alle Desired Messages die letzten 4 Bits gleich 1 haben. Das bedeutet aber auch, dass die letzten 4 Bits der Undesired Messages nicht alle gleichzeitig 1 sein dürfen. Die Nachrichten-ID für das Experiment werden in der Tabelle 6.3 zusammengetragen. Die Nachrichten sind im Binärformat und die Periode ist in Millisekunden dargestellt.

6.1.2 Ergebnisse

In diesem Abschnitt werden die Ergebnisse der Experimente zusammengetragen. Die Dateneingabe der Experimente wurde im Abschnitt 6.1.1 vorgestellt. Die Diskussion über die Ergebnisse ist in dem Abschnitt 6.1.3 zu finden.

ID (binär)	Periode (ms)	desired	ID (binär)	Periode (ms)	desired
00000001010	1000		01011110010	100	
0000001**1111**	1000	ja	01100000000	1000	
00000100000	100		01101010000	1000	
00001100000	100		01110101100	100	
0000110**1111**	100	ja	10000000000	100	
00001101110	10		10000010000	100	
00001110001	100		10000010001	100	
00001111000	100		10000100000	1000	
00001111101	100		10000100101	1000	
0001000**1111**	1000	ja	10101100101	500	
00010001100	100		11000010000	100	
00010010001	100		11000010001	1000	
00010010110	500		11000010010	100	
00100000000	10		1100001**1111**	1000	ja
00100000001	10		11000010100	1000	
00100000010	10		11000011000	100	
00100010000	10		1100010**1111**	100	ja
00100100000	10		1101111**1111**	1000	ja
00100110000	10		11011111000	1000	
00101010000	10		11100000000	1000	
00101100000	1000		11100000010	1000	
00101100001	1000		1110001**1111**	1000	ja
00000000000	1000		11100010001	1000	
01000000001	10		11100100000	1000	
0100001**1111**	10	ja	1110011**1111**	1000	ja
0100010**1111**	10	ja	1110111**1111**	1000	ja
01011110000	10		11101110001	1000	
01011110001	100				

Tabelle 6.3: Modified Network Specification of Battery Electric Vehicle [28]

Beispiel mit 6 Nachrichten

In diesem Versuch werden alle Desired Messages von der Filterkonfiguration akzeptiert. Die Maske ist 01111111110 und der Tag 00011001001. Die Filterkonfiguration *001100100*. Keine der Nachrichten aus der Menge der Undesired Messages passiert den Filter. Die Qualität des Filters ist gleich 0, daher ist in diesem Fall die Filterkonfiguration perfekt. Die Laufzeit nähert sich 0 an.

Epsilon Benchmark

In diesem Experiment werden alle Desired Messages von der Filterkonfiguration akzeptiert. Die Maske ist 00000000100 und der Tag 00000001011. Die Filterkonfiguration ********0**. 36 von insgesamt 44 Undesired Messages passieren den Filter. Die Qualität des Filters ist gleich $1,044$, daher ist in diesem Fall die Filterkonfiguration nicht perfekt. Die Laufzeit ist gleich 0.000498 Millisekunden.

SAE Benchmark

Diese Versuchsreihe besteht aus zwei Gruppen. Die erste Versuchsgruppe untersucht das Verhalten des Filters für eine große Anzahl der relevanten Nachrichten. Die zweite Gruppe untersucht im Gegensatz zu der ersten das Verhalten des Filters für eine kleinere Anzahl an Desired Messages.

Für alle Versuche gilt:

- die Laufzeit nähert sich 0 an (vgl. Diagrammen 6.2, 6.5)
- alle Desired Messages werden von dem Filter akzeptiert
- aufgrund der großen Anzahl der Versuche werden hier keine Masken und Tags präsentiert
- die Durchschnittswerte für die Filterqualität sowie für das Passieren der irrelevanten Nachrichten den Filter sind den Diagrammen 6.1, 6.3, 6.4, 6.6 zu entnehmen.

Die Filterkonfiguration ist in einem Fall einzigen perfekt, wenn es nur eine Nachricht aus Desired Messages akzeptiert wird. Wenn es bis zur 3 akzeptierenden Nachrichten gibt, dann ist die Qualität des Filters zu tolerieren. Sonst passieren den Filter mehr als 15 Nachrichten aus der Menge der Undesired Messages. Der Grund dafür ist, die Nachrichten-IDs werden zufällig aus $2^{11} = 2048$ IDs generiert. Es ist unwahrscheinlich, dass 51 generierte Nachrichten nah bei einander liegen und eine gewisse Ähnlichkeit aufweisen.

Perfekte Filterkonfiguration

In diesem Experiment, das die Daten aus der Tabelle 6.3 verwendet, werden alle Desired Messages von der Filterkonfiguration akzeptiert. Die Maske ist 00000001111 und der Tag 10110111111. Die Filterkonfiguration *******1111. Keine Undesired Messages passiert den Filter. Die Qualität des Filters ist gleich 0, die Filterkonfiguration ist also perfekt. Die Laufzeit ist gleich 0.000496 Millisekunden.

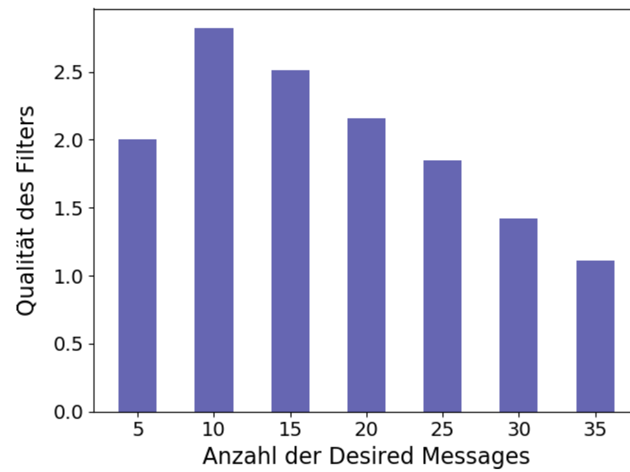

Abbildung 6.1: SAE Messungen : durchschnittliche Qualität des Filters für unterschiedliche Anzahl der Desired Messages

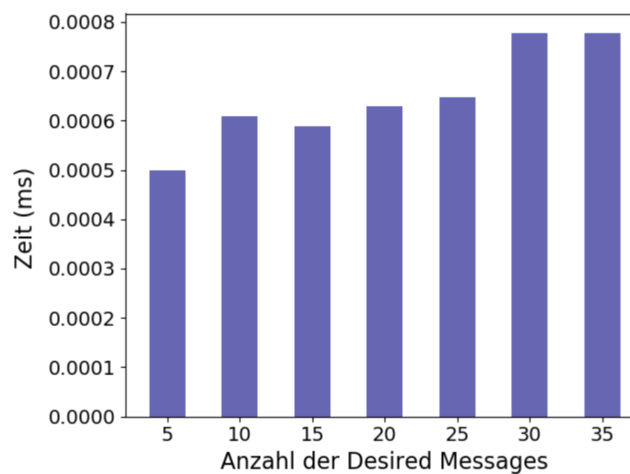

Abbildung 6.2: SAE Messungen: durchschnittliche Laufzeit (in Millisekunden) für unterschiedliche Anzahl der Desired Messages

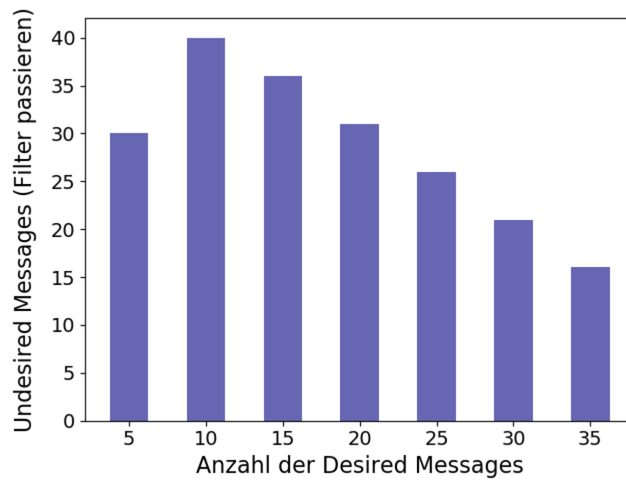

Abbildung 6.3: SAE Messungen: durchschnittliche Anzahl der Nachrichten aus der Menge der Undesired Messages, die vom Filter akzeptiert werden, für unterschiedliche Anzahl der Desired Messages

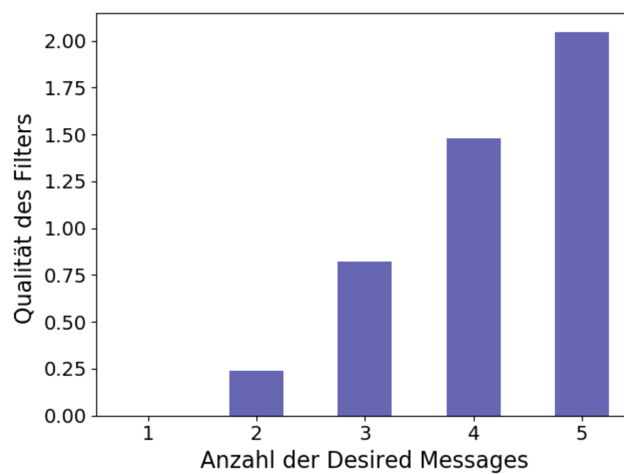

Abbildung 6.4: SAE Messungen für kleine Anzahl Desired Messages: durchschnittliche Qualität des Filters für unterschiedliche Anzahl der Desired Messages

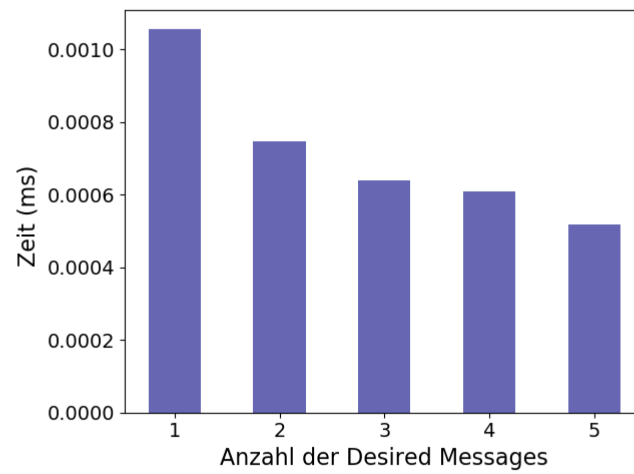

Abbildung 6.5: SAE Messungen für kleine Anzahl Desired Messages: durchschnittliche Laufzeit (in Millisekunden) für unterschiedliche Anzahl der Desired Messages

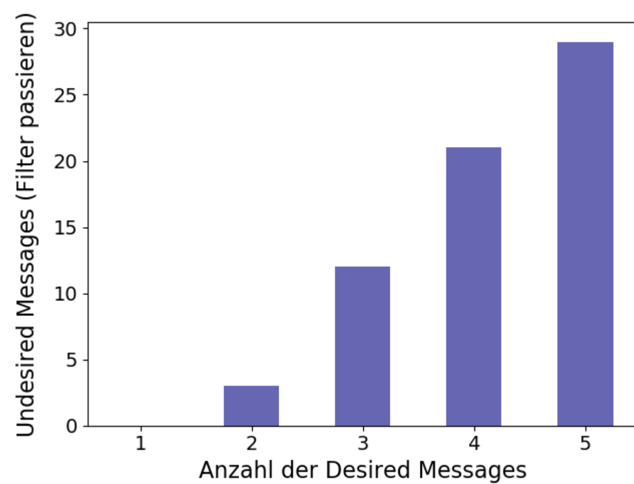

Abbildung 6.6: SAE Messungen für kleine Anzahl Desired Messages : durchschnittliche Anzahl der Nachrichten aus der Menge der Undesired Messages, die vom Filter akzeptiert werden, für unterschiedliche Anzahl der Desired Messages

6.1.3 Diskussion und Schlussfolgerungen

Die Experimente haben gezeigt, dass die Heuristik schnell die Filterkonfiguration berechnet. Die Laufzeiten nähern sich 0 an.

Für kleinere Systeme mit insgesamt kleiner Anzahl an Nachrichten liefert die Heuristik eine gute bis perfekte Filterqualität.

Damit die Qualität auch für größere Systeme perfekt wird, sollen die Nachrichten aus der Menge der Desired Messages geschickt gewählt werden, in dem sie einem bestimmten Muster entsprechen und zusätzlich keine Nachricht aus der Menge der Unesired Messages diese Muster aufweist. Aus diesem Grund sollten auf keine Fall die Nachrichten-IDs zufällig generiert werden.

6.2 Design der Nachrichten

Dieser Abschnitt betrachtet das ILP-Model, das der Solver Gurobi (vgl. Kap. 5.1.4) löst.

Der Löser gibt einen optimalen Wert zurück und die Belegung der Variablen stellt die gesuchten Nachrichten dar.

6.2.1 Experimenteller Aufbau

Das Model *„ilp_problem.lp“* ist basierend auf dem ILP-Szenario kreiert und wird nicht im Kapitel 5.3 beschrieben. Die Problembeschreibung (vgl. Kap. 4.1) erläutert, dem System sind die Anzahl der Desired Messages sowie Maske und Tag bekannt. Das ILP-Szenario legt die Anzahl der Desired Messages auf vier fest. Die Maske und der Tag sind wie in dem Beispiel 3.2 definiert und in der Tabelle 6.4 visualisiert. Die Nachrichten-ID ist 11 Bit lang, für 29 Bit wird im Rahmen dieser Arbeit nicht getestet.

ID	Bit_0	Bit_1	Bit_2	Bit_3	Bit_4	Bit_5	Bit_6	Bit_7	Bit_8	Bit_9	Bit_{10}
Maske	0	1	1	1	1	1	1	1	1	1	0
Tag	0	0	0	1	1	0	0	1	0	0	1

Tabelle 6.4: Die Maske und der Tag für ILP

6.2.2 Ergebnisse

Die Variablenbelegung, die Gurobi für die beschriebenen Eingabedaten berechnet, sieht wie folgt aus:

```
 x[1,0] 1.0  x[2,0] 0.0  x[3,0] 0.0  x[4,0] 1.0
 x[1,1] 0.0  x[2,1] 0.0  x[3,1] 0.0  x[4,1] 0.0
 x[1,2] 0.0  x[2,2] 0.0  x[3,2] 0.0  x[4,2] 0.0
 x[1,3] 1.0  x[2,3] 1.0  x[3,3] 1.0  x[4,3] 1.0
 x[1,4] 1.0  x[2,4] 1.0  x[3,4] 1.0  x[4,4] 1.0
 x[1,5] 0.0  x[2,5] 0.0  x[3,5] 0.0  x[4,5] 0.0
 x[1,6] 0.0  x[2,6] 0.0  x[3,6] 0.0  x[4,6] 0.0
 x[1,7] 1.0  x[2,7] 1.0  x[3,7] 1.0  x[4,7] 1.0
 x[1,8] 0.0  x[2,8] 0.0  x[3,8] 0.0  x[4,8] 0.0
 x[1,9] 0.0  x[2,9] 0.0  x[3,9] 0.0  x[4,9] 0.0
x[1,10] 0.0 x[2,10] 0.0 x[3,10] 1.0 x[4,10] 1.0
```

Diese Variablenbelegung entspricht den gesuchten Nachrichten-IDs. Die IDs sind in der Tabelle 6.5 visualisiert.

ID_1	ID_2	ID_3	ID_4
1	0	0	1
0	0	0	0
0	0	0	0
1	1	1	1
1	1	1	1
0	0	0	0
0	0	0	0
1	1	1	1
0	0	0	0
0	0	0	0
0	0	1	1

Tabelle 6.5: Variablenbelegung des ILPs

Die Laufzeit nähert sich 0 an.

6.2.3 Diskussion und Schlussfolgerungen

Das Modell für das das Problem zu erstellen, ist ein sehr aufwändiges Prozess gewesen, da bei vielen Variablen und Nebenbedingungen schnell einen Fehler gemacht werden kann, also sind Konzentration und Aufmerksamkeit gefragt. Sobald das Modell steht, löst Gurobi das ILP in weniger als eine Sekunde. Das Modell entspricht dem Szenario. Das Ergebnis ist korrekt.

Mit der gewählten Maske und dem Tag können maximal 4 Nachrichten designet werden. Um mehr Nachrichten designen zu dürfen, soll die Maske mehr Flexibilität bieten, indem sie mehr 0-Bits hat. Wenn das Maskenbit gleich 1 ist, zwingt es das Nachrichtenbit, den Wert des Tagbits anzunehmen, um den Filter passieren zu dürfen.

Kapitel 7

Fazit und Ausblick

Ziel dieser Arbeit war zwei Fragen zu erforschen: Das Design einer Hardware-Filterkonfiguration für Broadcast-Bus-basierte Anwendungen und das Design der Nachrichten von Kontext Integer Linear Programming.

Broadcast-Bus-basierte Systeme werden von einer großen Menge von Nachrichten überschwemmt. Die für ein elektronischen Steuergerät irrelevanten Nachrichten können von einem Hardwarefilter blockiert werden und somit den Prozessor nicht unnötig belasten. Das Kapitel 3 formuliert einen heuristischen Ansatz, um Filterkonfigurationen zu designen. Die Güte der Lösung wird Anhand einer Metrik gemessen, die auf Basis des Passierens der irrelevanten Nachrichten gemessen wird. Diese Heuristik wurde in Python implementiert und für verschiedene Eingaben getestet (vgl. Kap. 6.1). Ein Ergebnis des Ansatzes wird schnell erhalten, die Laufzeit beträgt weniger als 1 Sekunde. Diese Heuristik findet eine gute Filterkonfiguration, wenn entweder die Anzahl der für ein Gerät interessanten Nachrichten gering ist oder wenn die interessante Nachrichten auf eine strategische Art gewählt sind.

Die in dieser Arbeit vorgeschlagene Lösung zum Finden einer Hardware-Filterkonfiguration basiert auf einem Filter. Die Qualität der Filterkonfiguration kann verbessert werden, indem diese auf weitere Filter erweitert wird. Die Anzahl an Nachrichten zu erhöhen, kann neue Erkenntnisse mit sich bringen.

Die zweite Fragestellung beschäftigt sich mit dem Design der Nachrichten. Das mathematische Model ist als ILP formuliert (vgl. Kap. 4.2), mittels Python umgesetzt (vgl. Kap. 5.3), mit Hilfe von dem Gurobi Optimizer (vgl. Kap. 5.1.4) gelöst und für ein bestimmtes Szenario getestet (vgl. Kap. 6.2). Das Modell ist komplex und dies zu entwickeln war ein mühsamer Prozess, jedoch lässt Gurobi das ILP Modell in weniger als einer Sekunde lösen. Das ILP-Modell basiert auf 11-Bit-Nachrichten-IDs, lässt sich aber auf 29 Bit erweitern.

Leider war es nicht möglich, im Rahmen dieser Arbeit die Komplexität der vorgeschlagenen Lösung in Bezug auf Nachrichten-ID-Länge oder Anzahl der zu designenden Nachrichten zu erforschen. Aus Zeitgründen war es nicht möglich, die Performance näher zu untersuchen. An dieser Stelle ließen sich weiterführende Forschung betreiben und möglicherweise neue Erkenntnisse gewinnen.

Abbildungsverzeichnis

2.1 CAN 2.0A: Base Frame Format [18] 11

2.2 CAN 2.0B: Extended Frame Format [18] 12

6.1 SAE Messungen : durchschnittliche Qualität des Filters für unterschiedliche Anzahl der Desired Messages 53

6.2 SAE Messungen: durchschnittliche Laufzeit (in Millisekunden) für unterschiedliche Anzahl der Desired Messages 53

6.3 SAE Messungen: durchschnittliche Anzahl der Nachrichten aus der Menge der Undesired Messages, die vom Filter akzeptiert werden, für unterschiedliche Anzahl der Desired Messages 54

6.4 SAE Messungen für kleine Anzahl Desired Messages: durchschnittliche Qualität des Filters für unterschiedliche Anzahl der Desired Messages 54

6.5 SAE Messungen für kleine Anzahl Desired Messages: durchschnittliche Laufzeit (in Millisekunden) für unterschiedliche Anzahl der Desired Messages . . 55

6.6 SAE Messungen für kleine Anzahl Desired Messages : durchschnittliche Anzahl der Nachrichten aus der Menge der Undesired Messages, die vom Filter akzeptiert werden, für unterschiedliche Anzahl der Desired Messages 55

Tabellenverzeichnis

2.1 Beispiel: Die Auswirkungen von Wahlkampftaktiken auf die Wähler 20

3.1 Beispiel: Desired Messages und Undesired Messages 28
3.2 Summe der einzelnen Bits, Maske, Tag und Filterkonfiguration [40] 28

5.1 Die Maske und der Tag für ILP 42

6.1 Beispiel mit 6 Nachrichten 48
6.2 Network Specification of Battery Electric Vehicle [28] 49
6.3 **Modified** Network Specification of Battery Electric Vehicle [28] 51
6.4 Die Maske und der Tag für ILP 56
6.5 Variablenbelegung des ILPs 57

Literaturverzeichnis

[1] *ActiveState: Komodo. Website.* https://www.activestate.com/products/komodo-ide/. Aufgerufen am 19.03.2019.

[2] *Anaconda. Website.* https://www.anaconda.com/. Aufgerufen am 19.03.2019.

[3] *Comparing Python to Other Languages.* https://www.python.org/doc/essays/comparisons/. Aufgerufen am 24.03.2019.

[4] *GLPK - GNU Projekt - Free Software Foundation (FSF).* https://www.gnu.org/software/glpk/. Aufgerufen am 26.03.2019.

[5] *Gurobi: Beispielkunden.* http://www.gurobi.com/company/example-customers. Aufgerufen am 26.03.2019.

[6] *Gurobi: Beispielkunden.* http://www.gurobi.com/products/gurobi-optimizer. Aufgerufen am 26.03.2019.

[7] *Gurobi. Website.* http://www.gurobi.com/. Aufgerufen am 19.03.2019.

[8] *JetBrains: Entwicklungsumgebung PyCharm. Website.* https://www.jetbrains.com/pycharm/. Aufgerufen am 19.03.2019.

[9] *JetBrains. Website.* https://www.jetbrains.com/. Aufgerufen am 19.03.2019.

[10] *Python 3.6.3. Website.* https://www.python.org/downloads/release/python-363/. Aufgerufen am 19.03.2019.

[11] *Python Anwendungsgebiete.* https://www.python.org/about/success/. Aufgerufen am 24.03.2019.

[12] *Python Dokumentation.* https://docs.python.org/3/. Aufgerufen am 24.03.2019.

[13] *Python Editors.* https://wiki.python.org/moin/PythonEditors. Aufgerufen am 24.03.2019.

[14] *SourceForge: lpsolve.* https://sourceforge.net/projects/lpsolve/. Aufgerufen am 26.03.2019.

[15] *Spyder. Website.* `https://www.spyder-ide.org/`. Aufgerufen am 19.03.2019.

[16] Python Packaging Authority. *Setuptools for Python packages.* `https://github.com/pypa/setuptools`. Aufgerufen am 19.03.2019.

[17] The Pip Developers. *Pip - The Python Package Installer. Website.* `https://pip.pypa.io/en/stable/`. Aufgerufen am 19.03.2019.

[18] *Section 23. CAN Module.* Technischer Bericht, Microchip Technology Inc., 2007. Seiten 32-33.

[19] al., T. E. Oliphant et: *NumPy. Website.* `https://www.numpy.org/`. Aufgerufen am 05.04.2019.

[20] Alevras, D. und M. W. Padberg: *Linear Optimization and Extensions: Problems and Solutions.* Springer Berlin Heidelberg, 2001.

[21] Bicking, I.: *Virtual Python Environment builder. Website.* `https://virtualenv.pypa.io`. Aufgerufen am 19.03.2019.

[22] Bosch: *Controller Area Network Specification 2.0*, 1991.

[23] Burkard, R. E. und U. T. Zimmermann: *Einführung in die Mathematische Optimierung.* Springer-Verlag Berlin Heidelberg, 2012.

[24] Domschke, W.: *Logistik: Rundreisen und Touren.* De Gruyter, Oldenbourg, 4 Auflage, 2018. Seite 20-29.

[25] Domschke, W., A. Drexl, R. Klein, Robert und A. Scholl: *Einführung in Operations Research.* Springer Gabler, 9 Auflage, 2015. Seiten 1-69.

[26] Ernesti, J. und P. Kaiser: *Python 3. Das umfassende Handbuch.* Galileo Press, Bonn, 3 Auflage, 2012. Page 27-40.

[27] Etschberger, K. (Herausgeber): *Controller- Area- Network: Grundlagen, Protokolle, Bausteine, Anwendungen.* Carl Hanser Verlag München Wien, 3 Auflage, 2002.

[28] F. Pölzlbauer, R. I. Davis und I. Bate: *Analysis and optimization of message acceptance filter configurations for controller area network (CAN).* Int. Conf. on Real-Time Networks and Systems (RTNS), Seiten 247–256, Oktober 2017.

[29] Hunter, J. D. und M. Droettboom: *matplotlib. Website.* `https://matplotlib.org/`. Aufgerufen am 05.04.2019.

[30] Islam, Q. N.: *Mastering PyCharm. Use PyCharm with fluid efficiency.* Packt Publishing, 2015.

[31] *International Organization for Standardization ISO 11898. Road vehicles – Controller Area Network (CAN).* ISO 11898, 2016.

[32] *International Organization for Standardization ISO 11989-4. Road vehicles – Controller area network (CAN) – Part 4: Time-triggered communication.* ISO 11898-4, August 2004.

[33] *International Organization for Standardization ISO 11989-3. Road vehicles – Controller area network (CAN) – Part 3: Low-speed, fault-tolerant, medium-dependent interface.* ISO 11898-3, Juni 2006.

[34] *International Organization for Standardization ISO 11989-5. Road vehicles – Controller area network (CAN) – Part 5: High-speed medium access unit with low-power mode.* ISO 11898-5, Juni 2007.

[35] *International Organization for Standardization ISO 11989-6. Road vehicles – Controller area network (CAN) – Part 6: High-speed medium access unit with selective wake-up functionality.* ISO 11898-6, November 2013.

[36] *International Organization for Standardization ISO 11989-1. Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling.* ISO 11898-1, Dezember 2015.

[37] *International Organization for Standardization ISO 11989-2. Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit.* ISO 11898-2, Dezember 2016.

[38] *International Organization for Standardization ISO 17987. Road vehicles – Local Interconnect Network (LIN).* ISO 17987, 2016.

[39] Korte, B. und J. Vygen: *Kombinatorische Optimierung: Theorie und Algorithmen.* Springer Spektrum, 3 Auflage, 2018.

[40] L. Schönberger, G. v. d. Brüggen, H. Schirmeier und J.-J. Chen: *Design Optimization for Hardware-Based Message Filters in Broadcast Buses.* March 25-29 2019.

[41] Lawrenz, W. (Herausgeber): *CAN System Engineering.* Springer London, 2013.

[42] Martin, A. B.: *PyGurobi: Rapid interactive Gurobi model modification and analysis.* `https://github.com/AndrewBMartin/pygurobi`. Aufgerufen am 19.03.2019.

[43] Marwedel, P.: *Eingebettete Systeme.* Springer-Verlag Berlin Heidelberg, 2008.

[44] Padberg, M.: *Linear Optimization and Extensions: Second, Revised and Expanded Edition.* Springer Berlin Heidelberg, 1999.

[45] R. Bradshaw, S. Behnel, D. Seljeboth und G. Ewing: *Cython: C-Extensions for Python. Website.* https://cython.org/. Aufgerufen am 19.03.2019.

[46] R. I. Davis, A. Burns, R. J. Bril und J. J. Lukkien: *Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised.* Real-Time Systems, 35(3):239–272, jan 2007.

[47] Reif, K.: *Bosch Autoelektrik und Autoelektronik: Bordnetze, Sensoren und elektronische Systeme.* Vieweg+Teubner Verlag, Springer Fachmedien Wiesbaden, 6 Auflage, 2011. Seiten 92-105.

[48] Steyer, R.: *Programmierung in Python. Ein kompakter Einstieg für die Praxis.* Springer Vieweg, 2018. Seiten 2-5.

[49] Suhl, L. und T. Mellouli: *Optimierungssysteme: Modelle, Verfahren, Software, Anwendungen.* Springer Berlin Heidelberg, 3 Auflage, 2013.

[50] T. H. Cormen, C. E. Leiserson, R. Rivest und C. Stein: *Algorithmen - Eine Einführung.* Oldenbourg Verlag München, 4 Auflage, 2017. Seiten 857-910.

[51] Taylor, J.: *First Look – Gurobi Optimization.* http://jtonedm.com/2011/03/02/first-look-gurobi-optimization/, März 2011. Aufgerufen am 20.03.2019.

[52] Theis, T.: *Einstieg in Python 3.* Galileo Press, Bonn, 2 Auflage, 2009. Seiten 13-14.

[53] Unger, T. und S. Dempe: *Lineare Optimierung: Modell, Lösung, Anwendung.* Vieweg+Teubner Verlag, 1 Auflage, 2010.

[54] Weigend, M.: *Python. Ge-Packt.* MITP Verlags GmbH, 7 Auflage, 2017. Seiten 13-15.

[55] Weigend, M.: *Python 3. Lernen und professionell anwenden. Das umfassende Praxisbuch.* MITP Verlags GmbH, 7 Auflage, 2018. Seite 23.

[56] Werners, B.: *Grundlagen des Operations Research: Mit Aufgaben und Lösungen.* Springer-Verlag Berlin Heidelberg, 3 Auflage, 2013.

[57] Woyand, H.-B.: *Python für Ingenieure und Naturwissenschaftler. Einführung in die Programmierung, mathematische Anwendungen und Visualisierungen.* Hanser Fachbuchverlag, 2 Auflage, 2018.

[58] Wörn, H. und U. Brinkschulte: *Echtzeitsysteme. Grundlagen, Funktionsweisen, Anwendungen.* Springer-Verlag Berlin Heidelberg, 2005. Seiten 278-289.

[59] Zimmermann, H. J.: *Operations Research: Methoden und Modelle.* Vieweg & Sohn Verlag, 2 Auflage, 2008. Seiten 1-187.

[60] ZIMMERMANN, W. und R. SCHMIDGALL: *Bussysteme in der Fahrzeugtechnik. Protokolle, Standards und Softwarearchitektur.* Springer Vieweg, Springer Fachmedien Wiesbaden, 5 Auflage, 2014. Seiten 57-79.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 8. April 2019

Iryna Denysenko