Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Jonas Engler

Ein Framework zu einer OMNeT++ basierten Simulation von CAN-Netzwerken auf der Sicherungsschicht

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Jonas Engler

Ein Framework zu einer OMNeT++ basierten Simulation von CAN-Netzwerken auf der Sicherungsschicht

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Franz Korf
Zweitgutachter : Prof. Dr. Hans Heinrich Heitmann

Abgegeben am 02.05.2013

Jonas Engler

Thema der Arbeit

Ein Framework zu einer OMNeT++ basierten Simulation von CAN-Netzwerken auf der Sicherungsschicht

Stichworte

Framework, OMNeT++, Feldbus, CAN-Bus, Simulation, Netzwerk, Automotive, Fahrzeug-Elektronik, Automatisierungstechnik

Kurzzusammenfassung

Die Bachelorarbeit erarbeitet Konzept und Umsetzung einer Simulation, die die Kommunikation des CAN-Busses auf Ebene der Sicherungsschicht abbildet. In der Entwicklungsumgebung OMNeT++ wird ein Framework umgesetzt, das die Kommunikation von CAN-Netzwerken veranschaulicht und anschließend eine Analyse dafür anfertigt. In dem Bereich der Qualitätssicherung werden die Kernfunktionen der Simulation getestet. Anschließend wird eine Betrachtung bzw. Evaluierung der Ergebnisse betrieben, um sicherzustellen, dass alle Anforderungen erfüllt wurden.

Jonas Engler

Title of the paper

A framework for an OMNeT++ based Simulation of CAN-Networks on the data-link-layer

Keywords

Framework, OMNeT++, Fieldbus, CAN-Bus, Simulation, Network, Automotive, vehicle electronics, Automation Engineering

Abstract

This thesis develops a conception and realization of a simulation modeling the CAN-Bus communication on the data-link-layer. Using the development environment OMNeT++, a framework for the illustration of CAN-Network-communication and subsequent creation of an analysis is built. The chapter quality assurance performs tests on the main functions of the simulation. Eventually, the results will be monitored and evaluated, to ensure the correct transformation of the requirements into the final simulation.

Inhaltsverzeichnis

1 Einleitung 7

2 Technische Grundlagen 9

2.1 Feldbus 9

2.2 CAN-Protokoll 9

2.2.1 Nachrichtenübertragung 10

2.2.2 Nachrichtenpakete 12

2.2.3 Bit-Stuffing 13

2.2.4 ACK-Bit 14

2.2.5 Message-Buffer 14

2.3 OMNeT++ 14

2.4 Begriffserklärungen 15

3 Anforderungsspezifikation 17

3.1 Abgebildete Schichten 17

3.2 Abstraktionsebene 17

3.3 Frames 18

3.4 Konfigurationsmöglichkeiten 19

3.5 Fehler 20

3.6 Ausgabeumfang 20

4 Konzept 22

4.1 Related Work 22

4.2 Aufbau 24

4.2.1 Teilnehmer ... 24
4.2.2 Bus ... 25
4.3 Arbitrierung-Strategie ... 26
4.4 Nachrichtenarten ... 32
4.4.1 Data-Frame ... 32
4.4.2 Remote-Frame ... 32
4.4.3 Error-Frame ... 33
4.4.4 Overload-Frame ... 33
4.5 Versendung von Nachrichten ... 33
4.5.1 Versendung eines Data-Frames, Fehler deaktiviert ... 33
4.5.2 Fehlersignalisierung ... 36
4.5.3 Versendung eines Data-Frames mit Fehler beim Sender ... 37
4.5.4 Versendung eines Data-Frames mit Fehler beim Empfänger ... 40
4.6 Konfigurationsmöglichkeiten ... 41
4.6.1 Grundsätzliche Funktionen ... 41
4.6.2 Erweiterte Funktionen ... 43
4.6.3 Stimuli-Generator ... 44
4.6.4 Verbindungen ... 44
5 Umsetzung ... 45
5.1 Übersicht ... 45
5.2 Nachrichten und Informations-Container ... 47
6 Qualitätssicherung ... 48
6.1 Testfall 1: Arbitrierung ... 48
6.2 Testfall 2: Zeitverhalten ... 50
6.3 Testfall 3: Payload ... 51
6.4 Testfall 4: Fehlerinjektion durch Stimuli-Generator ... 52
6.5 Laufzeitfehler ... 55
7 Ergebnisse / Evaluierung ... 56
7.1 Philips-Modell ... 56
7.2 Effizienz-Tests ... 59
7.2.1 Bestmögliche Effizienz ... 59

7.2.2 Test 1: ACK-Bit 60
7.2.3 Test 2: Payload 60
7.2.4 Test 3: Fehler 61
8 Zusammenfassung und Ausblick 65
Literaturverzeichnis 67
Abbildungsverzeichnis 69

1 Einleitung

Lange vorbei sind die Zeiten, in denen ein Auto nur aus Karosserie, Motor, Reifen und mechanischem Getriebe bestand. „Das Ziel war in den 1970er-Jahren schon, mit neuen Techniken einen Beitrag für sichere, saubere und sparsame Autos zu leisten.“ [1, p. 10] Technische Neuerungen wie das Antiblockiersystem sorgten für ein Umdenken, was fortan die stetig wachsende Integration von neuer Technik und Elektronik in der Fahrzeugentwicklung zur Folge hatte. Mehr Elektronik bedeutete allerdings, dass die Kabelbäume in Autos immer größer und schwerer wurden.

In den 1980er Jahren fand dieses Problem eine Lösung: Die ersten Feldbusse wurden vorgestellt. Ein Feldbus verbindet alle verbauten Komponenten seriell mit nur einer Leitung und spart damit extrem an Kabellänge. Es entstanden viele Protokolle, mit denen ein Feldbus kommunizieren kann. Um die Nutzung von Feldbussen weltweit zu standardisieren, wurde 1999 die Norm IEC 61158 (vgl. [2]) entworfen.

Außerhalb dieser Norm wurde ab 1993 der CAN(Controller Area Network)-Bus in ISO 11898 (vgl. [3]) standardisiert. Er wurde 1991 als erstes Bussystem in einem Serienfahrzeug verbaut. „Im Kfz-Bereich hat er sich seitdem als Standard etabliert.“ [1, p. 92] Nicht nur die Automobil-Industrie zieht einen Nutzen aus dieser Entwicklung, auch die Automatisierungstechnik benutzt Feldbus-Systeme wie CAN. Die Automatisierungstechnik hat das Ziel, „[…] Systeme so zu steuern, dass die automatisierten Systeme selbstständig ihre Funktion erfüllen.“ [4, p. 1] Die Automatisierungstechnik umfasst einige Gebiete, interessant für diese Arbeit ist allerdings der Bereich der Medizintechnik, speziell die Röntgentechnik.

Die Röntgentechnik unterliegt diversen Bestimmungen, die den Schutz für die Personen garantieren sollen, die an solchen Systemen arbeiten. Die Automatisierung der Steuerung von Röntgenstrahlern durch Generatoren trägt gleichermaßen zur Sicherheit, als auch zu einem erhöhten Bedienkomfort solcher Anlagen bei. Diesem Automatisierungsvorgang geht

ein Prozess der Planung für die Kommunikation zwischen den einzelnen Komponenten voraus.

Durch Berechnung der Buslast muss im Laufe der Kommunikations-Planung sichergestellt werden, dass alle Nachrichten bei ihrem Ziel ankommen. Ist dies nicht der Fall, ist eventuell die Sicherheit des Systems beeinträchtigt. Die Berechnung der Buslast kann bei vielen Komponenten und Nachrichten sehr aufwendig sein.

Dies schafft einen Teil der Grundlage dieser Bachelorarbeit: Die Erleichterung der Kommunikations-Planung durch Bereitstellung einer Simulation, die eine Analyse zu abgebildeten CAN-Netzwerken generiert. Die Erstellung der Analyse ist von der Abteilung DXR Development des Unternehmens Philips Healthcare (vgl. [5]) gefordert. Aus diesem Grund wird diese Arbeit ein durch Philips vorgegebenes Modell eines CAN-Netzwerk-Aufbaus verwenden.

Der andere Teil der Grundlage wird durch die Arbeitsgruppe CoRE (vgl. [6]) an der HAW vorgegeben: Die Gruppe beschäftigt sich hauptsächlich mit der Umsetzung von Kommunikation über Realtime-Ethernet. Für das Aufstellen der Zeitplanung wird die Entwicklungsumgebung OMNeT++ benutzt. Diese ist für die Erstellung von Netzwerk-Simulationen ausgelegt. Da die Simulation der Gruppe für Realtime-Ethernet bereits unter OMNeT++ betrieben wird, ist es ihnen ein Anliegen, weitere Netzwerksimulationen möglichst unkompliziert integrieren zu können. Aus diesem Grund wird die Simulation, die im Rahmen dieser Arbeit erstellt wird, mit OMNeT++ umgesetzt.

Zu Beginn der Arbeit werden die nötigen Grundlagen erklärt. Kapitel 3 stellt die Anforderungsspezifikation auf. Anschließend wird die Vorgehensweise konzipiert. Kapitel 5 gibt einen kurzen Überblick auf die Umsetzung der Simulation. Die anschließenden Kapitel prüfen die Kernfunktionen der Simulationen und evaluieren die Ergebnisse. Abschließend wird eine Zusammenfassung mit Ausblick geboten.

2 Technische Grundlagen

Dieses Kapitel bietet eine Hilfestellung zur Bachelorarbeit. Es vermittelt einen Eindruck der Materie, geht auf die Funktion des CAN-Busses ein und erklärt themenbezogene Begriffe.

2.1 Feldbus

Ein Feldbus agiert als System, dass die Verbindung zwischen Sensoren (z.B. Messgeräte) und Aktoren (z.B. Steuergeräte) herstellt. Die angeschlossenen Sensoren und Aktoren kommunizieren nur über den Bus miteinander, sind also auf keine andere Weise miteinander verbunden. Dies wird als „serielle Verdrahtung“ bezeichnet. Der Gegensatz dazu ist die parallele Verdrahtung, die alle Teilnehmer untereinander verbindet.
Einige relevante Feldbus-Protokolle:

- ***CAN:*** Für den Automotive-Bereich ausgelegt, CSMA/CR-Verfahren
- *EtherCAT* (vgl. [7])*:* In der Automatisierungstechnik eingesetzt, ist auf Echtzeitanforderungen ausgelegt
- *FlexRay* (vgl. [8])*:* Für den Automotive-Bereich ausgelegt. Deterministisch und fehlertolerant
- *LIN* (vgl. [9])*:* Für den Automotive-Bereich ausgelegt
- *MOST* (vgl. [10])*:* Für den Automotive-Bereich ausgelegt

2.2 CAN-Protokoll

Der CAN-Bus wurde 1983 von Bosch als Kommunikationsprotokoll entwickelt und nach der Zusammenarbeit mit Intel 1987 erstmals vorgestellt. Er wurde „[…]für robuste Anwendungen im Automobil[…]“ [11, p. 3] konzipiert. Er wurde 1990 erstmals von Mercedes Benz verbaut und gewinnt seitdem stetig an Absatz. Die produzierte Stückzahl überschritt Mitte der 1990er Jahre die Millionen-Marke „[…] und heute werden es mehr als eine Milliarde pro Jahr sein.“ [11, p. 3].

Das CAN-Protokoll liegt in der aktuellen Spezifikation in 2 Fassungen vor: CAN2.0A und CAN2.0B. Version 2.0A unterstützt 11Bit lange Identifier (dezimal = 2047), Version 2.0B bis zu 29 Bit (dezimal = 536870911). Eine Implementierung muss 2.0A akzeptieren und 2.0B zumindest tolerieren (vgl. [12, pp. 13, 46]). Die Simulation bietet eine Konfiguration auf 2.0A oder 2.0B.

Das CAN-Protokoll implementiert im ISO/OSI-Schichtmodell nur Schicht 1 (Bitübertragung) und Schicht 2 (Sicherung, Datenübertragung auf MAC-Ebene). Die entscheidende Anwendungsschicht 7 muss zusätzlich implementiert werden, damit das CAN-Protokoll sinnvoll genutzt werden kann. Diese Arbeit beschäftigt sich nicht mit den höheren Protokollen. Die Simulation bildet nur die Kommunikation auf Schicht 2 ab.
Einige für Schicht 7 entwickelte höhere Protokolle sind:

- CANopen (vgl. [13]): Heute am meisten verbreitet
- CAL – CAN Application Layer (vgl. [13]): Inzwischen veraltet und inaktiv
- DeviceNet (vgl. [14])
- TTCAN – Time-Triggered Communication on CAN (vgl. [15])

Im Folgenden wird die Funktionsweise des CAN-Busses kurz erläutert.

2.2.1 Nachrichtenübertragung

Der CAN-Bus arbeitet nach dem CSMA/CR(Carrier Sense Multiple Access / Collision Resolution)-Verfahren. „Bei diesem einfachen Zugriffsverfahren entfällt die bei kontrollierten Verfahren erforderliche Busbelegung für die Verwaltung des Buszugriffs […].“ [16] Es ist darauf ausgelegt, Kollisionen bei einem Buszugriff zu verhindern und dabei eine Übertragung ohne Zeitverzögerung zu garantieren. Hierfür wird die (bitweise) Arbitrierung verwendet.

Arbitrierung

Die Arbitrierung ist der Vorgang des Vergleichens von Nachrichten-IDs. Zu jeder Zeit, in der ein Knoten mit Sendewunsch den Bus in einem „Idle“-Zustand vorfindet, wartet er auf das Ende der aktuellen Bitzeit und startet dann die Arbitrierungs-Phase mit einem Start-Of-Frame-Bit (vgl. [17]). Der Vorgang entscheidet darüber, wer seine Nachricht auf dem Bus übertragen darf. Die Nachrichten-IDs stellen also die Priorität der Nachrichten dar. Je niedriger die ID, desto höher die Priorität.

Funktionsweise:

Wenn der Bus im Zustand „idle“ ist, kann die Arbitrierung beginnen. Alle an der Arbitrierung teilnehmenden Knoten fangen gleichzeitig an, ihre Nachrichten-ID bitweise auf dem Bus zu übertragen. Sie fangen mit dem höchstwertigsten Bit an. Da der Bus nur eine einzige Leitung hat (er kann nur den Pegel 1 oder 0 aufweisen), beeinflussen alle gleichzeitig den Pegel. Es gibt einen dominanten (=0) und rezessiven (=1) Pegel auf dem Bus. Wenn ein Knoten eine 1 (rezessiv) schreibt und ein zweiter Knoten eine 0 (dominant) schreibt, so wird der Pegel des Busses den dominanten Wert, also 0, annehmen.

Während der ID-Übermittlung vergleichen alle Knoten den Pegel mit dem selbst übermittelten Bit. Wenn ein Knoten nach der Übertragung einen anderen Pegel auf dem Bus liest stoppt er die Übertragung. Auf diese Weise wird spätestens nach Übertragung des letzten ID-Bits sichergestellt, dass nur noch der Knoten mit der niedrigsten ID sendet.

Im Folgenden wird ein Beispiel von drei Knoten betrachtet, die gleichzeitig senden wollen. Die Nachrichten-IDs (dezimal – binär in 11 Bits) sind:
Knoten 1: 362 – 00101101010
Knoten 2: 378 – 00101111010
Knoten 3: 418 – 00110100010
Abb. 1 veranschaulicht den Prozess der Arbitrierung.

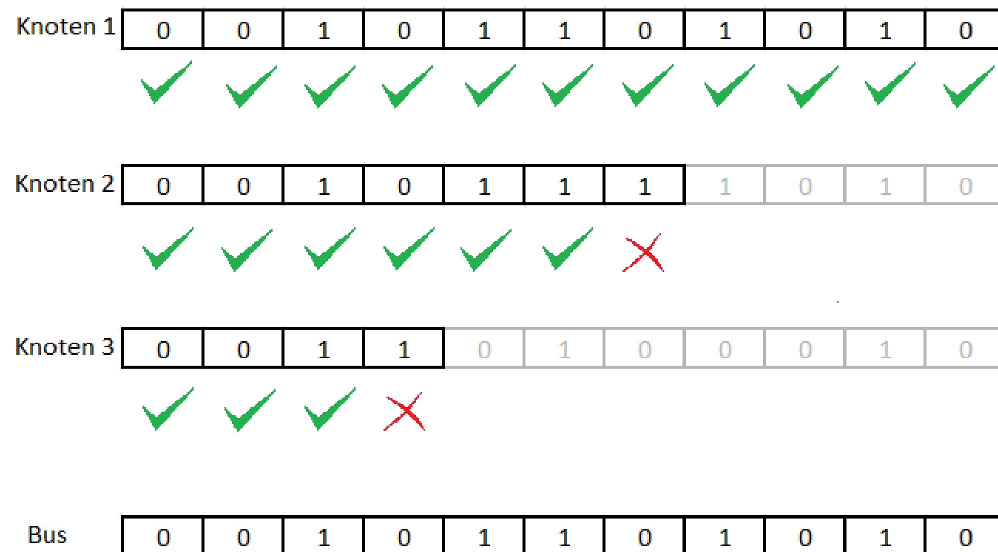

Abbildung 1: Arbitrierung mit 3 Knoten

Da Knoten 1 am Ende der einzige noch sendende Knoten ist, fährt er anschließend mit der Sendung der gesamten Nachricht fort. Durch dieses Verfahren entsteht keine Zeitverzögerung bei der Sendung.

Daten

Der Knoten, der nach der Arbitrierung weiter sendet, übermittelt neben Informationen über den Aufbau der Nachricht (Overhead) die Daten. Diese können 0-8 Byte lang sein. Der Knoten sendet somit ein ganzes Paket.

Ende der Übertragung

Nach der kompletten Übersendung des Data-Frames folgen die 10 Bits „End-of-Frame“ und „Inter-Frame-Space“. Nach Abschluss dieser beiden Teile ist der Bus erneut im Zustand „idle“. Ab jetzt kann erneut die Arbitrierung beginnen.

2.2.2 Nachrichtenpakete

Die Knoten eines CAN-Netzwerkes kommunizieren durch Nachrichtenpakete, sogenannte „Frames“. Es gibt insgesamt 4 unterschiedliche Frames.

Data-Frame

Der Data-Frame ist die Struktur, mit der die Knoten Daten austauschen. Er enthält Daten (Payload) und Overhead. Zum Overhead gehört auch die Nachrichten-ID. Abb. 2 zeigt den Aufbau des Data-Frames mit dem 11-Bit-Identifier.

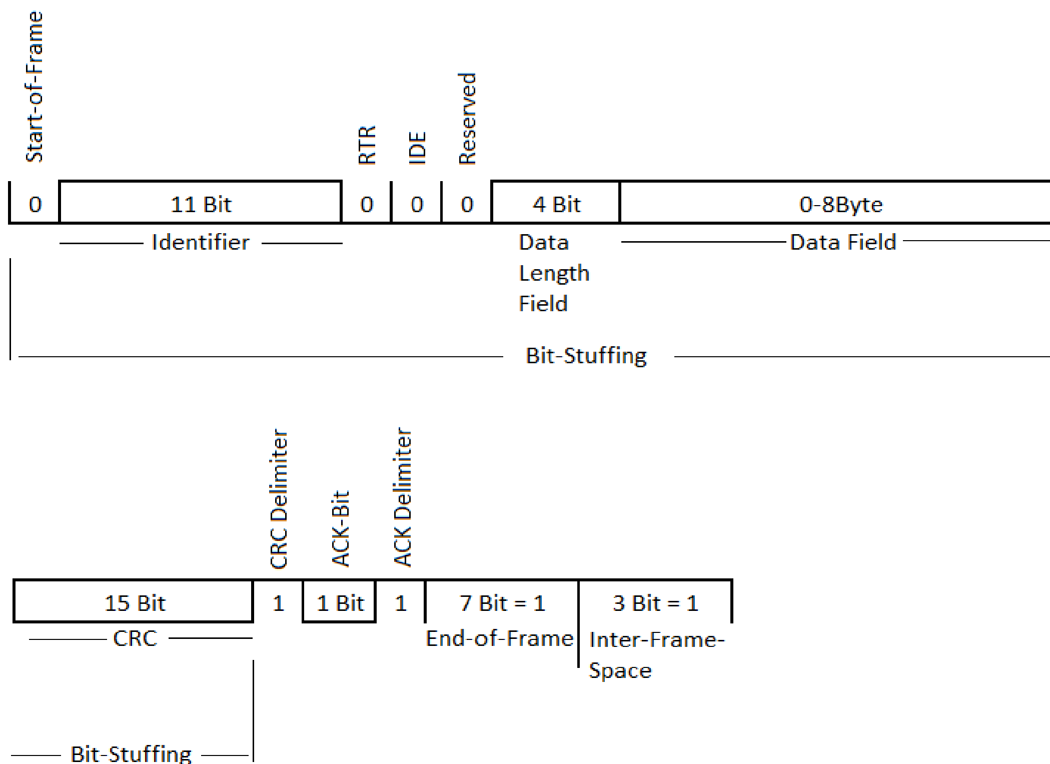

Abbildung 2: Data-Frame (CAN2.0A)

Start-of-Frame, RTR, IDE, CRC- und ACK-Delimiter haben immer die gleichen vorgegebenen Werte. Das Datenfeld besteht aus 0-8 Bytes und das Daten-Längen-Feld gibt diese Länge in Bits an. Ab dem CRC-Delimiter ist kein Bit-Stuffing mehr aktiv und nach dem Inter-Frame-Space schaltet der Bus in den Zustand „Idle“. CAN2.0B-Frames sind durch den längeren Identifier und weiteren Overhead insgesamt 20 Bit länger.

Remote-Frame

Der Remote-Frame wird von einem Knoten verschickt, um von einem anderen Knoten im Netzwerk einen Data-Frame anzufordern. Ein Remote-Frame ist ein Data-Frame, der an der Stelle des RTR-Bits (s. Abb. 2) eine 1 schreibt. Wenn ein Remote-Frame gleichzeitig mit einem Data-Frame mit der gleichen ID die Arbitrierung beginnt, so wird der Remote-Frame seine Sendung stoppen, da das RTR-Bit als letztes Bit der Arbitrierungs-Phase geschickt wird und als rezessives Bit (=1) gegen das dominante Bit des Data-Frames (=0) verliert.

Error-Frame

Der Error-Frame wird von den Knoten zur Signalisierung von Fehlern benutzt. Er besteht aus sechs dominant auf den Bus geschriebenen Bits. Wenn andere Knoten den Error-Frame erkennen, werden sie ihrerseits mit einem Error-Frame antworten. Da dies spätestens nach dem sechsten Bit des Error-Frames geschieht, ist die Belegungszeit des Busses durch Error-Frames auf 12 Bit limitiert.

Overload-Frame

Der Overload-Frame wird von den Knoten verwendet, um die Zeit zwischen der Sendung zu verlängern.

2.2.3 Bit-Stuffing

Das Bit-Stuffing ist der Synchronisationsmechanismus des CAN-Protokolls.

Funktionsweise:

Wenn ein Knoten etwas langsamer oder schneller den Pegel abliest, könnte es zu Synchronisationsfehlern kommen. Wenn viele gleiche Bits am Stück geschrieben werden, führt ein verschobenes Lesen schnell zu Fehlern. Um dem vorzubeugen werden sogenannte Stuffing Bits in die Übertragung integriert. Die Integration folgt einer Regel: Wenn entweder

- fünf dominante oder
- fünf rezessive Bits

aufeinander folgen, wird ein komplementäres Bit in die Übertragung eingefügt. In die Zählung eingeschlossen sind auch die Stuffing Bits selber. Die eingefügten Bits werden auf Empfänger-Seite automatisch erkannt und für die Verarbeitung des Frames entfernt. Der Bereich der Data-Frame-Übertragung, in dem Bit-Stuffing aktiviert ist, ist in Abb. 2 markiert. Abb. 3 zeigt eine Bitsequenz und ihre Veränderung durch Bit-Stuffing.

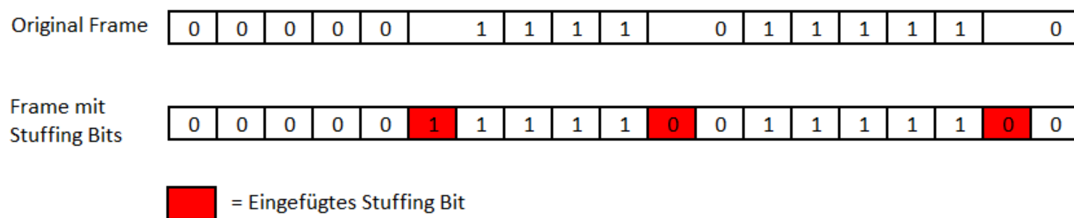

Abbildung 3: Bit-Stuffing Sequenz

Durch die eingefügten Stuffing Bits werden drei zusätzliche Bits geschickt.

2.2.4 ACK-Bit

Nach der Versendung eines Data-Frames erwartet der Sender eine Empfangsbestätigung. Der Sender schreibt während des ACK-Slots (s. Abb. 2 – ACK-Bit) einen rezessiven Pegel. Jeder Knoten, der die Nachricht bis dahin erfolgreich empfangen hat, schreibt ein dominantes Bit auf den Bus. Der Sender liest den Bus erneut aus und weiß bei dominantem Pegel, dass die Übertragung erfolgreich war. Wenn er weiterhin einen rezessiven Pegel abliest, bedeutet dies einen Fehler in der Übertragung – er wiederholt die Sendung.

2.2.5 Message-Buffer

Die Message-Buffer sind ein Filterungs-Mechanismus des CAN-Protokolls. Sie sind Bestandteil eines Knotens und geben an, welche Nachrichten-IDs versendet und/oder empfangen werden dürfen. Nur durch Message-Buffer zugelassene Nachrichten lösen im empfangenden Knoten einen Interrupt aus.

2.3 OMNeT++

OMNeT++ ist eine erweiterbare, modulare, Komponenten-basierte C++ Simulations-Programmbibliothek und ein Framework, das primär für die Erstellung von Netzwerk-Simulatoren verwendet wird. (vgl. [18]) Diese Entwicklungsumgebung wird für die Erstellung der Simulation verwendet.

Unter OMNeT++ erstellte Netzwerke kommunizieren grundsätzlich mit Paketen bzw. Nachrichten. Abb. 4 zeigt die Versendung einer Nachricht zwischen zwei Knoten eines Netzwerkes. Jeder Knoten ist eine Instanz eines Moduls, das per C++ implementiert wird. Die Nachrichten-Kommunikation eignet sich nur teilweise für die Umsetzung der CAN-Bus-Simulation. Auf diese Problematik wird unter anderem in 4.2 eingegangen.

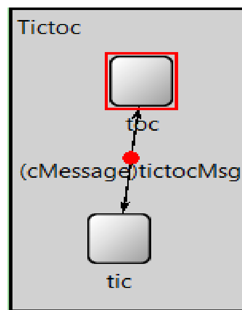

Abbildung 4: Übertragung einer Nachricht in Beispielnetz

Die Simulationszeit unter OMNeT++ wird für die Zeitplanung der Events verwendet. OMNeT++ ist Event-basiert (vgl. [19]) Das heißt, dass die Zeitplanung nur dann funktioniert, wenn weitere Events ausgeführt werden müssen, andernfalls stoppt die Simulation. Aus diesem Grund ist ein Stimuli-Generator unverzichtbar für aufwendige Systeme.

2.4 Begriffserklärungen

Folgende Begriffe werden im Laufe der Arbeit verwendet:

- *Aktor:*
  Der Gegenpart zu einem Sensor. Ein Aktor kann zum Beispiel ein Steuergerät sein, das aufgrund von Sensor-Daten eine bestimmte Aktion ausführt.
- *Bandbreite:*
  Die Bandbreite beschreibt die mögliche Anzahl an Bits, die auf dem Bus in einer Sekunde übertragen werden können. 1Mbps = 1Million Bits in der Sekunde.
- *Pegel:*
  Der Bus besteht aus einer Leitung, die die Zustände 1 oder 0 annehmen kann. Der Zustand der Leitung wird Pegel genannt.
- Rezessiv – Dominant
  Ein dominantes Bit überschreibt das rezessive Bit, ein rezessives Bit hat bei dominantem Pegel keine Auswirkungen. Im CAN-Protokoll ist die 1 rezessiv und die 0 dominant.
- *Buslast:*
  Die Buslast steht synonym für die Auslastung des Busses. Wenn der Bus z.B. über den gemessenen Zeitraum nur 10% der Zeit belegt ist, so liegt die Buslast bei 10%.
- *Durchsatz:*
  Gibt die Menge an, die in einem bestimmten Zeitraum übertragen wird.
- *Bitzeit:*
  Dies beschreibt eine relative Zeiteinheit bei der Datenübertragung. Bitzeit in Realzeit variiert je nach Bandbreite. Ist die Bandbreite 100bps, so braucht die Übertragung eines Bits $\frac{1s}{100} = 0.01s$. Also ist in diesem Fall eine Bitzeit = 0.01s.

- *<u>Stimuli-Generator:</u>*
  Eine Simulation braucht Komponenten, die in der Simulation abgebildet werden, sonst würde die Simulation eine leere Menge abbilden (=nichts tun). Der Stimuli-Generator erstellt diese Komponenten und/oder gibt ihnen Aufgaben, die die Simulation abbildet.
- *<u>Abstraktionsebene:</u>*
  Eine hohe Abstraktionsebene vereinfacht die Implementierung in der Programmierung. Wenn Abstraktionsebene 1 beispielsweise die Möglichkeit bietet, einen Wert zu lesen <u>oder</u> zu schreiben, so könnte Abstraktionsebene 2 eine Möglichkeit bieten, in einem Schritt einen Wert zu lesen (Ebene 1), zu verändern und wieder zu schreiben (Ebene 1).
- *<u>ISO/OSI-Schichten-Modell</u>*
  Für die Kommunikation in Netzwerken werden verschiedene Ebenen betrachtet. Diese Arbeit behandelt nur Schicht 2 (Sicherungsschicht). CAN im Allgemeinen steht in Verbindung mit den Schichten 1, 2 und 7, daher werden die Schichten 3,4,5 und 6 hier nicht weiter erklärt.
  Schicht 1 - Bitübertragungsschicht:
  Die physikalische Schicht beschäftigt sich mit der Übertragung der „rohen" Bits über einen Kommunikations-Kanal. Wenn die eine Seite ein Bit mit dem Wert 1 sendet, muss sichergestellt sein, dass die andere Seite das Bit als 1 und nicht als 0 empfängt. (vgl. [20, p. 35]) Die Schicht beschäftigt sich mit der Verbindung, Leitungsbeschaffenheit, Signalart und weiteren physikalischen Aspekten.
  Schicht 2 – Sicherungsschicht:
  Die Hauptaufgabe der Sicherungsschicht ist die Transformierung einer „rohen" Übertragung in eine Reihe von Bits, die frei von unerkannten Übertragungsfehlern für die Vermittlungsschicht (Schicht 3) ist. Die Sicherungsschicht sieht vor, dass der Sender seine Daten in Datenpakete (engl. Data-Frames) aufteilt. (vgl. [20, p. 35])
  Die Aufteilung in Datenpakete wird von CAN direkt mit dem Modell des Data-Frames umgesetzt.
  Schicht 7 – Anwendungsschicht:
  Die Anwendungsschicht enthält eine Vielzahl von Protokollen, die häufig von Benutzern gebraucht werden. (vgl. [20, p. 36]) Die Anwendungsschicht stellt das Interface zum Benutzer dar. Sie umfasst im Falle von CAN die oben genannten höheren Protokolle, die einen Umgang mit CAN zugänglicher machen.

3 Anforderungsspezifikation

In diesem Kapitel wird eine Übersicht der Anforderungen an die Simulation aufgestellt. Da hier auf die Anforderungen von Philips DXR-Development (nachfolgend als „Philips“ bezeichnet) als auch auf Anforderungen der Arbeitsgruppe „CoRE“ an der HAW (nachfolgend als „HAW“ bezeichnet) geachtet werden muss, werden diese gegenüber gestellt und im Anschluss eine allgemeine Anforderungsspezifikation entworfen.

3.1 Abgebildete Schichten

Die abgebildeten Schichten beziehen sich auf das ISO/OSI-Schichten-Modell. Der CAN-Bus benutzt aus diesem Schichtenmodell die Schichten 1 („Physical Layer“ – dt. „Bitübertragungsschicht“), 2 („Data Link Layer“ – dt. „Sicherungsschicht“) und 7 („Application Layer“ – dt. „Anwendungsschicht“).

Philips

Wichtig für die Simulation ist in erster Linie die Schicht 2, mit der die Datenübertragung abgebildet wird. Eine physikalische Abbildung (entsprechend Ebene 1) ist ebenso wie eine Abbildung der Ebene 7 nicht nötig.

HAW

Abbildung der Schicht 2 ist nötig.

Gemeinsame Anforderungsspezifikation

Die Simulation des CAN-Bus bildet die ISO/OSI-Schicht 2 (Sicherungsschicht) ab.

3.2 Abstraktionsebene

Die Abstraktionsebene entspricht hier der Komplexitätsebene. Es beschreibt beispielsweise die Art, wie die Kommunikation zwischen den Knoten abgebildet werden muss.

Philips

Für die Übertragung von Nachrichten genügt eine komplette Übertragung des Frames. Dies beinhaltet Data-Frames, Error-Frames und Remote-Frames. Das heißt, es wird der Frame als ein Paket geschickt und nicht bitweise über den Bus übermittelt. In der Übermittlung eines Pakets über den Bus werden aus Synchronisierungsgründen „Stuffing Bits“ eingefügt.

Diese gehen als berechnete Einheit (die größtmögliche Anzahl an Stuffing Bits) mit in das Paket. Dies ist wichtig, um die richtige Übertragungsgeschwindigkeit des Pakets zu ermitteln. Je nach Einstellung des Systems können Stuffing Bits berechnet werden, s. dazu Punkt 4.5.1.

Die Arbitrierung erfolgt mit gesamter ID. Wenn ein Busteilnehmer seine ID für die Arbitrierung sendet, wird die komplette ID betrachtet und mit in Konkurrenz tretenden Nachrichten-IDs verglichen. Der Teilnehmer mit der höchsten Priorität sendet dann ohne Zeitverlust weiter.

Die Buszustände können „Idle“ oder „Busy“ sein.

HAW

Es sollen Nachrichten komplett übertragen werden, also nicht bitweise.
Die Arbitrierung soll mit der gesamten ID erfolgen.
Die Buszustände sind die oben genannten.

Ein Teilnehmer quittiert den Empfang einer Nachricht mit einem „ACK“-Bit. Dieses wird mindestens an den Knoten geschickt, der die betreffende Nachricht geschickt hat.

Gemeinsame Anforderungsspezifikation

- Pakete (Frames) werden komplett geschickt (nicht bitweise)
- Arbitrierung nur per ID (nicht bitweise)
- Buszustände „Idle“, „Busy“
- Die Knoten quittieren Nachrichtenempfang mit „ACK“-Bit

3.3 Frames

Ein Frame bezeichnet ein Paket, das von einem Busteilnehmer gesendet wird.

Philips

Die 4 möglichen Frames sind:

- Data-Frame mit Payload
    - In diesen Data-Frames wird die genaue Größe des Datenframes angegeben und der Inhalt benutzt. Wenn ein anderer Knoten diese Nachricht empfängt, dann soll er den Inhalt des Frames genau wiedergeben können.
- Data-Frame ohne Payload
    - In diesen Data-Frames wird lediglich die Größe des Inhalts angegeben.
    - In der Simulationszeit sollen sich die beiden Frames (mit und ohne Payload) nur anhand der gesamten Größe unterscheiden.
- Error-Frame
- Remote-Frame

HAW
Keine weiteren Frames.

Gemeinsame Anforderungsspezifikation
Mögliche Frames:
- Daten-Frame mit Payload
- Daten-Frame ohne Payload
- Error-Frame
- Remote-Frame

3.4 Konfigurationsmöglichkeiten
Die Konfigurationsmöglichkeiten sind die Eigenschaften des simulierten Systems, die vom Nutzer bei dessen Erstellung unterschiedlich gestaltet werden können.

Philips
Die Knoten haben folgende dynamische Eigenschaften:
- Periode der versendeten Nachricht (Wiederholungsrate)
- Art der versendeten Nachricht (s. 3.3)
- ID der versendeten Nachricht (Priorität)
- Länge der versendeten Nachricht
- Payload (optional eingebbar)

HAW
Zusätzlich zu den oben genannten Punkten:
- Konfiguration der Message Object Buffer eines Knoten. Das heißt, die Anzahl der möglichen empfangbaren und versendbaren Nachrichten wird mit der Anzahl der Message Object Buffer in einem Knoten geregelt.

Gemeinsame Anforderungsspezifikation
Eingabemöglichkeiten der Knoten:
- Periode der versendeten Nachrichten
- Art der versendeten Nachrichten (s. 3.3)
- ID der versendeten Nachricht (Priorität)
- Länge der versendeten Nachricht
- Payload (optional)
- Konfiguration der Message Object Buffer

3.5 Fehler

Fehler können auf Bussystemen natürlich vorkommen. Dieser Punkt beschreibt den Umfang, in dem diese generiert, überprüft und bearbeitet werden können und sollen.

Philips

Es ist keine Fehlererkennung nötig und es müssen keine Fehler generiert werden.

HAW

Es können Fehler auf dem Bus generiert werden.
Die korrekte Sendung der Nachrichten wird überprüft. Entweder haben die Frames die Eigenschaft „fehlerhaft“ oder ein Knoten liest fehlerhaft.
Das System kann mit fehlerhaften Nachrichten umgehen (Error-Frame).

Gemeinsame Anforderungsspezifikation

Es gibt:

- Eine Möglichkeit, Fehler zu erzeugen
- Eigenschaft „fehlerhaft“ von Nachrichten oder fehlerhaftes Lesen der Teilnehmer
- Eine Verarbeitung von fehlerhaften Nachrichten

3.6 Ausgabeumfang

Der Simulation folgt eine Auswertung, die in einer Ausgabe zusammengefasst wird.

Philips

Es sollen mindestens folgende Punkte in der Auswertung angezeigt werden:

- Gesamtbuslast
- Latenz der Nachrichten (zwischen den Knoten)
- Nachrichtenanzahl
- Error-Frames

HAW

Zusätzlich zu den oben genannten Punkten:

- Fehler auf dem Bus
- Fehlverhalten der Simulation (Laufzeitfehler durch z.B. falsche Eingaben)
- Jitter (Zeitdifferenz zwischen längster und kürzester Übertragung einer Nachricht)

Gemeinsame Anforderungsspezifikation

- Gesamtbuslast
- Latenz der Nachrichten
- Nachrichtenanzahl

- Error-Frames
- Fehler auf dem Bus
- Fehlverhalten der Simulation
- Jitter

4 Konzept

Dieses Kapitel vermittelt Ideen und Ansätze, die für die Erstellung der Simulation verwendet werden. Nach einem einleitenden Blick auf verwandte Veröffentlichungen wird der Aufbau des Netzwerks mit seinen Teilnehmern und deren Verknüpfungen betrachtet. Anschließend wird die Strategie für die Rechtevergabe in verschiedenen Ansätzen auf ihre Effizienz hin diskutiert und die effizienteste Vorgehensweise erörtert. Die verschiedenen Arten von Nachrichten und die genaue Funktionsweise der Nachrichtenübertragung werden im Anschluss erläutert. Abschließend werden Parameter veranschaulicht, die die Performanz des Systems durch Ein- oder Ausschalten von Funktionen beeinflussen, und die Konfigurations-Möglichkeiten des eingebauten Stimuli-Generators erklärt.

4.1 Related Work

Dieser Abschnitt wirft einen Blick auf verwandte Veröffentlichungen. Sie befassen sich mit dem CAN-Bus unter analytischen und/oder simulatorischen Gesichtspunkten. Hierbei wird insbesondere auf die Konzepte und Ergebnisse eingegangen und geprüft, ob diese für die Konzeptfindung dieser Arbeit nützlich sein können.

Der CAN-Bus kann aus dem Blickpunkt der ISO/OSI-Schichten 1 oder 2 betrachtet werden. Eine Abbildung des CAN-Busses durch eine Simulation beispielsweise hat also die Möglichkeit beide Schichten miteinzubeziehen oder sich auf eine der beiden zu konzentrieren. Eine solche Simulation kann dann zur Analyse einer der beiden oder beider Schichten verwendet werden.

Simulationen

Long Cheng beschreibt in seinem Bericht *„An Interactive Simulation Framework for Satellite Design“* [21] den Entwurf einer Simulation, die helfen soll, den physikalischen Aufbau eines CAN-Netzwerkes nachzuvollziehen und zu prüfen. Ähnlich verfährt W. Prodanov in seinem Entwurf *„A controller area network Bus Transceiver Behavioral Model for Network Design and Simulation“* [22]. Diskutierte Themen dieser Veröffentlichungen sind Treiber und Verbindungsmechanismen der einzelnen Netzwerkknoten bzw. direkte physikalische Einflüsse und Gegebenheiten wie zum Beispiel Temperaturänderungen oder Integrität der Signale. Die daraus entstandenen Simulations-Modelle setzen zu diesem Zweck auf die *ISO/OSI-Schicht 1* auf. Durch ihren unkomplizierten Aufbau liefern sie einen einfachen Zugang und tragen somit zum besseren Verständnis für die physikalischen Abläufe in der

Kommunikation des CAN-Busses bei. Die Zielsetzung dieser Bachelorarbeit ist es allerdings, eine Abbildung für die Kommunikation auf ISO/OSI-Schicht 2 zu erstellen.

Xiao Hong legt in seinem Bericht *„Modeling and Simulation Analysis of CAN-bus on Bus Body"* [23] den Entstehungsprozess einer Simulation dar, die ihren Fokus auf die *ISO/OSI-Schicht 2* legt. Ebenso auf Schicht 2 konzentriert sich die Veröffentlichung *„Modeling and Simulation of CAN Network Based on OPNET"* [24] von Jia Hao. Für den Bericht von Hao ist hervorzuheben, dass die Umsetzung unter OPNET geschieht. OPNET ist zwar kein Bestandteil dieser Bachelorarbeit, ist aber in der Vorgehensweise ähnlich wie OMNeT++.

Hong konzentriert sich auf den Entwurf eines Modells, das detailliert beschrieben wird. Es wird der Aufbau des Netzwerkes mit allen enthaltenen Nachrichten beschrieben. Die Simulation bietet zusätzlich eine Möglichkeit der Analyse, da Statistiken der Belegung und der Übertragung generiert werden. Die Erstellung von Statistiken ist interessant, da diese in dieser Bachelorarbeit auch eine Anforderung an die Simulation stellt.

Im Gegensatz zu dieser detaillierten Beschreibung eines spezifischen Modells widmet sich Hao der Dokumentation seiner Simulation. Es wird der Aufbau der Komponenten erklärt. Diese teilen sich in mehrere Unterkomponenten auf, die den Kommunikations-Ablauf der MAC-Ebene (befindet sich in ISO/OSI-Schicht 2) verdeutlichen und abbilden sollen. Anhand von Automaten werden die Zustände der Komponenten gesteuert. In Bezug auf diese Arbeit kann eine Unterteilung der Komponenten bis zu einem gewissen Punkt sinnvoll sein. Entscheidend ist hierbei, dass keine Verschlechterung der Performanz eintritt. Für die Umsetzung wird daher generell auf Unterteilungen der Komponenten verzichtet.

Analysen

Fang Li und L.M. Pinho geben mit ihren Berichten „CAN(Controller Area Network) Bus Communication (System Based on Matlab/Simulink)" [25] und „Integrating Inaccessibility in Response Time Analysis of CAN Networks" [26] einen Einblick in die Analyse von CAN-Netzwerken.

Li implementiert eine Simulation in Matlab und analysiert diese dann mit Experimenten und zuvor bereitgestellten Formeln. Der Unterschied zu den o.g. Simulationen ist der Fokus, der hier auf Anfertigung von Analysen liegt. Eine Simulation auf Matlab hat zudem keine Relevanz für das Arbeiten mit OMNeT++. Die einzelnen Punkte der Analyse bieten aber einen weiteren Ansatz, wie der Ergebnisumfang der Simulation dieser Bachelorarbeit aussehen kann.

Pinho implementiert keine Simulation, sondern fertigt Formeln für die zeitlichen Abläufe der Kommunikation in einem CAN-Netzwerk an. Beispielsweise wird analysiert, wie sich der Bus im Fehlerfall verhält und welche Auswirkungen dies auf das zeitliche Verhalten mit sich bringt. Zugrunde liegt diesen Fehlerberechnungen eine Eigenschaft des CAN-Protokolls, wie

mit Fehlern einzelner Knoten zu verfahren ist. Die Ergebnisse der Analysen können in der Qualitätssicherung als Vergleich für eigene Ergebnisse herangezogen werden. Da Pinho mit seinen Formeln auf einem Fehlermechanismus des CAN-Busses aufbaut, der im Umfang dieser Arbeit nicht implementiert werden wird, ist dies nur teilweise möglich.

4.2 Aufbau

Im Wesentlichen besteht die Simulation aus zwei Komponenten: Dem Bus und dessen angeschlossenen Teilnehmern. Abb. 5 zeigt ein Beispiel-Aufbau eines Netzwerks. Alle Teilnehmer (Knoten) haben die gleichen Grundfunktionen. Der Bus – ebenfalls ein Knoten, der allerdings die Teilnehmer miteinander verbindet – übernimmt die Steuerung im Netzwerk, vergibt Rechte für das Senden und verteilt ankommende Nachrichten auf jeden anderen Teilnehmer im Netzwerk. Weil die gesamte Simulation auf die Entwicklungsumgebung OMNeT++ aufsetzt, ist die Kommunikation des Netzwerks Nachrichten-basiert. Dies hat zur Folge, dass Informationsaustausch, der in einem CAN-Netzwerk ohne Nachrichten möglich ist (beispielsweise der Zustand des Busses), mit einem Modell aus Nachrichten abgebildet werden muss.

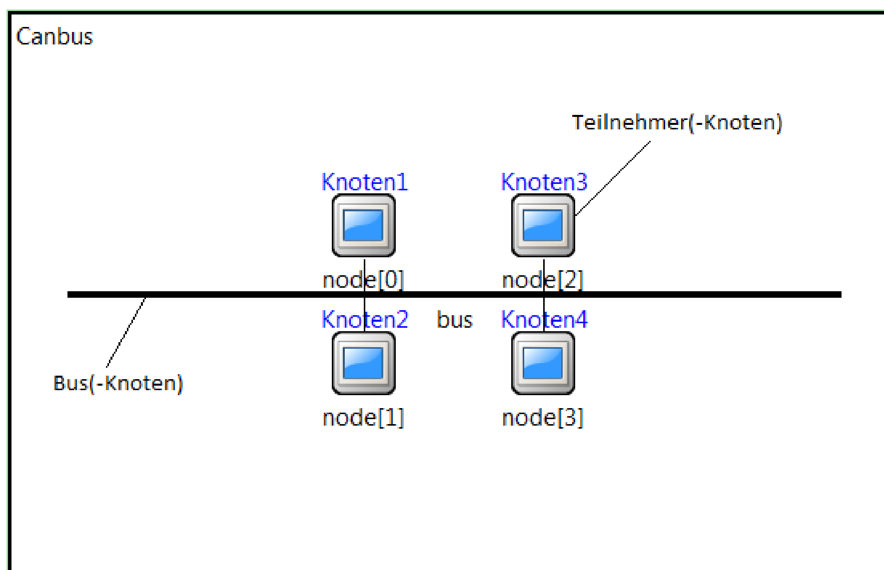

Abbildung 5: Beispielnetzwerk mit 4 angeschlossenen Knoten

4.2.1 Teilnehmer

Für jedes Netzwerk gibt es mehrere Knoten, die über den Bus miteinander kommunizieren. Die Simulation bildet nur die zweite Schicht des ISO/OSI-Schichten-Modells ab, das heißt, dass die physikalische Ebene der Kommunikation (Drahteigenschaften, Verkabelungsplan,

etc.) nicht betrachtet wird, dafür aber die verschickten Nachrichten und ihr Inhalt. Die Kommunikation geschieht über Nachrichten, die als Data-Frames geschickt werden.

Da in der Simulation nicht jedes Bit einzeln abgebildet wird, werden die Nachrichten als zusammenhängendes Paket übertragen (s. 4.4 – Data-Frame). Wenn ein Knoten eine Nachricht verschicken will, versucht er diese auf den Bus zu legen. Dazu muss diese Nachricht die momentan niedrigste ID (also höchste Priorität) aufweisen. Normalerweise funktioniert die Arbitrierung ohne eine zentrale Rechtevergabe. Für die Simulation wird aus technischen Gründen (OMNeT++ ist auf eine Kommunikation mit Nachrichten ausgerichtet) eine Rechtevergabe abgebildet (s. 4.3).

Periodisches Verschicken von Nachrichten

Die Kontrolle über den Zeitpunkt zum Versenden von Nachrichten liegt bei dem Knoten selber. Für jede Nachricht verwendet er einen Timer. Wenn vor Ablauf des Timers ein Remote-Frame für die entsprechende Nachricht eingeht, so hat dies den gleichen Effekt wie ein abgelaufener Timer: Der Knoten versucht, die entsprechende Nachricht auf den Bus zu legen. Wie dies genau passiert erklärt Abschnitt 4.3.

Der Knoten startet den Timer für die erneute Sendung, wenn der vorige Timer abgelaufen ist. Ein eingehender Remote-Frame hat keinen Einfluss auf den Timer, da häufiges Anfragen durch Remote-Frames sonst eine linear steigende Anzahl von verschickten Data-Frames zur Folge hätte. Wie genau ein Timer und Art der verschickten Nachricht konfiguriert werden, ist Abschnitt 4.6.3 zu entnehmen.

4.2.2 Bus

Die gesamte Kommunikation läuft über einen Bus-Knoten. Der Unterschied zum Teilnehmer-Knoten besteht darin, dass der Bus mit jedem Teilnehmer verbunden ist, während ein Teilnehmer nur die Verbindung zum Bus hat. Um eine Nachricht auf den Bus zu legen, muss dieser im „Idle“-Zustand sein. Wenn ein Teilnehmer eine Nachricht auf den Bus legen möchte, so muss er dessen Zustand überprüfen. In Punkt 4.3 wird die beste Strategie hierfür diskutiert.

Weiterleitung der Nachrichten

Alle Nachrichten des Netzwerkes werden an den Bus geschickt. Da jeder Teilnehmer des CAN-Busses gleichzeitig dessen Pegel abliest, muss dies abgebildet werden, indem der Bus die Nachrichten zeitgleich an alle Teilnehmer weiterleitet. Er verschickt also eine Kopie der angekommenen Nachricht an jeden Teilnehmer. Näheres zu der Versendung einer Nachricht in Abschnitt 4.5.

4.3 Arbitrierung-Strategie

Die Arbitrierung ist der Vorgang, der entscheidet, wer das Senderecht auf dem Bus erhält. In der Realität geschieht dies durch bitweises Vergleichen von Nachrichten-IDs, wenn der Bus im „Idle“-Zustand ist. Da in der Simulation keine bitweise Übertragung der Nachrichten realisiert wird, ist es nicht möglich die Arbitrierung originalgetreu abzubilden.

Stattdessen werden IDs via Nachricht verglichen. Die kleinste ID (höchste Priorität) erhält die Berechtigung zum Senden. Es gibt verschiedene Strategien, wie ein Teilnehmer die Berechtigung zum Senden bekommen kann. Um die Performanz der Simulation auf einem hohen Niveau zu halten, wird nachfolgend die beste Strategie der Rechte-Vergabe erörtert.

Der Zustand „Idle“ des Busses folgt auf einen vollständig verschickten Data-Frame. Ab diesem Moment ist der Bus bereit, mit einer erneuten Arbitrierung die Versendung eines Data-Frames zu ermöglichen. Ein sendebereiter Knoten überprüft, ob sich der Bus momentan im Zustand „Idle“ befindet. Da der Knoten unter OMNeT++ nicht einfach eine Variable des Bus-Knotens auslesen kann, muss dies durch Nachrichten zwischen dem Teilnehmer und dem Bus erfolgen. Dieses Problem wird nachfolgend schrittweise gelöst.

Für den Vergleich der Lösungsansätze wird jeweils die Formel für den Nachrichtenaufwand aufgestellt. Diese gibt an, wie viele Nachrichten ab Belegung des Busses für den Zweck der Arbitrierung geschickt werden bis die nächste Busbelegung erfolgt.

Anfrage der Knoten an den Bus

Will ein Knoten eine Nachricht senden, muss er den Zustand des Busses laufend überprüfen. Nach diesem Ansatz schickt ein sendewilliger Teilnehmer eine Nachricht an den Bus, um eine Antwortnachricht über den Zustand zu erhalten. In Abb. 6 wird dieser Ablauf gezeigt. Da nach der CAN-Spezifikation alle sendewilligen Teilnehmer gleichzeitig mit der Übersendung ihrer ID beginnen müssen – damit die Arbitrierung fehlerfrei funktioniert – müssen diese Teilnehmer jede neue Bitzeit den Zustand vom Bus einfordern. Das hat ein ein immenses Aufkommen an Nachrichten und bei wachsender Teilnehmerzahl für jeden Teilnehmer (mit Sendebereitschaft) eine zusätzliche Nachricht pro Bitzeit zur Folge. Je nach Anzahl der sendebereiten Knoten steigt somit der Nachrichtenaufwand.

$$Nachrichtenaufwand = \left(Teilnehmerzahl_{sendewillig} \cdot 2 \cdot Bitzeiten_{sendebereit}\right)$$

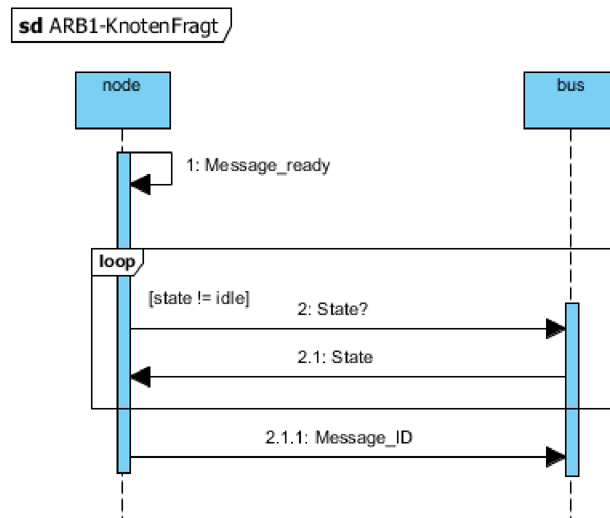

Abbildung 6: Arbitrierungsansatz 1 – Knoten fragt den Bus nach Zustand

Wenn viele Knoten gleichzeitig senden wollen sinkt die Performanz des Systems. Dieser Ansatz entspricht von der Vorgehensweise dem realen Vorgang einer Arbitrierung in einem CAN-Netzwerk, bezahlt dies allerdings mit potentiell sehr niedrigem Durchsatz. Für viele gleichzeitige Sendewünsche ist der folgende Ansatz praktischer.

Anfrage des Busses an die Knoten

Die Umkehr des ersten Ansatzes ist eine Anfrage des Busses an die Teilnehmer. Hier würde der Bus nur dann eine Anfrage stellen, wenn er im „Idle" Zustand ist. Abb. 7 veranschaulicht dies. Bei der Anfrage müssen alle Knoten abgedeckt werden, also muss der Bus eine Nachricht an jeden Teilnehmer schicken und auf die Antwort warten, um sicherzustellen, dass alle Teilnehmer in der Vergabe der Senderechte berücksichtigt werden.

$$Nachrichtenaufwand = (Teilnehmerzahl \cdot 2) \cdot freie\ Bitzeiten + 1$$

Dies ergibt sich durch eine Nachricht des Busses an jeden Teilnehmer, dessen Antwort und die Nachricht für die Sendeberechtigung. Bei hohen Buslasten ist dieser Ansatz gut und viele gleichzeitige Sendewünsche sind unproblematisch. Niedrige Buslasten hingegen wirken sich sehr schlecht auf die Performanz aus, da der Bus ununterbrochen Anfragen stellt. Der Nachrichtenaufwand würde also den des ersten Ansatzes schnell übersteigen.

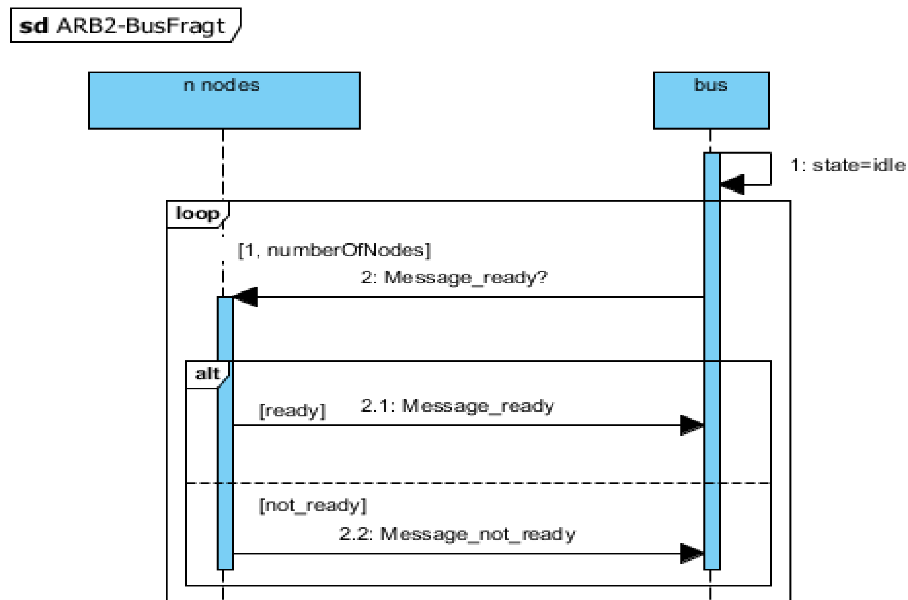

Abbildung 7: Arbitrierungsansatz 2 – Der Bus fragt die Knoten nach Sendewünschen

Ob ein Teilnehmer senden will, oder nicht und ob ein Teilnehmer vielleicht schon im letzten Durchgang eine Nachricht senden wollte, wird komplett vernachlässigt. Dies könnte behoben werden, indem ein solcher Knoten nicht mehr abgefragt wird, aber da ein Teilnehmer mehrere Nachrichten verschicken wollen könnte, würde dieser Wunsch vernachlässigt werden. Also bleibt das Problem der überflüssigen Abfrage von Sendewünschen bestehen. Bei diesem Ansatz sind die freien Bitzeiten problematisch. Unabhängig von diesen und der Buslast muss der Nachrichtenaufwand verringert werden.

Ringverbund der Knoten

In eine andere Richtung geht die Möglichkeit des Ringaufbaus. Hierzu würden alle Knoten – also auch der zentrale Busknoten – in einem Ring vernetzt werden. Der Aufbau würde zusätzlich zu den Verbindungen jedes einzelnen Teilnehmers mit dem zentralen Knoten noch die Verbindungen für einen Ring erfordern. Um den Zustand des Busses nun für alle Teilnehmer bekannt zu machen, würde der zentrale Knoten eine Nachricht herum schicken, die von jedem Teilnehmer mit Sendewunsch überprüft und gegebenenfalls (wenn die bisher vermerkte Nachrichten-ID von niedrigerer Priorität ist) mit der eigenen Nachrichten-ID versehen wird. Wenn die Nachricht am Ende wieder bei dem zentralen Knoten angelangt ist, wird anhand der vermerkten ID die Sendeberechtigung für den entsprechenden Knoten verschickt. Abb. 8 zeigt einen Ringaufbau mit 2 Teilnehmern.

$$Nachrichtenaufwand = (Teilnehmerzahl + 2) \cdot freie\ Bitzeiten$$

Dieser Aufwand ergibt sich durch das Verschicken der Nachrichten im Ring und die Nachricht des Busses an den sendeberechtigten Teilnehmer. Wie bei dem vorigen Ansatz ist hierbei zu beachten, dass dieser Vorgang jede neue Bitzeit wiederholt werden muss, bis ein Sendewunsch vorhanden ist.

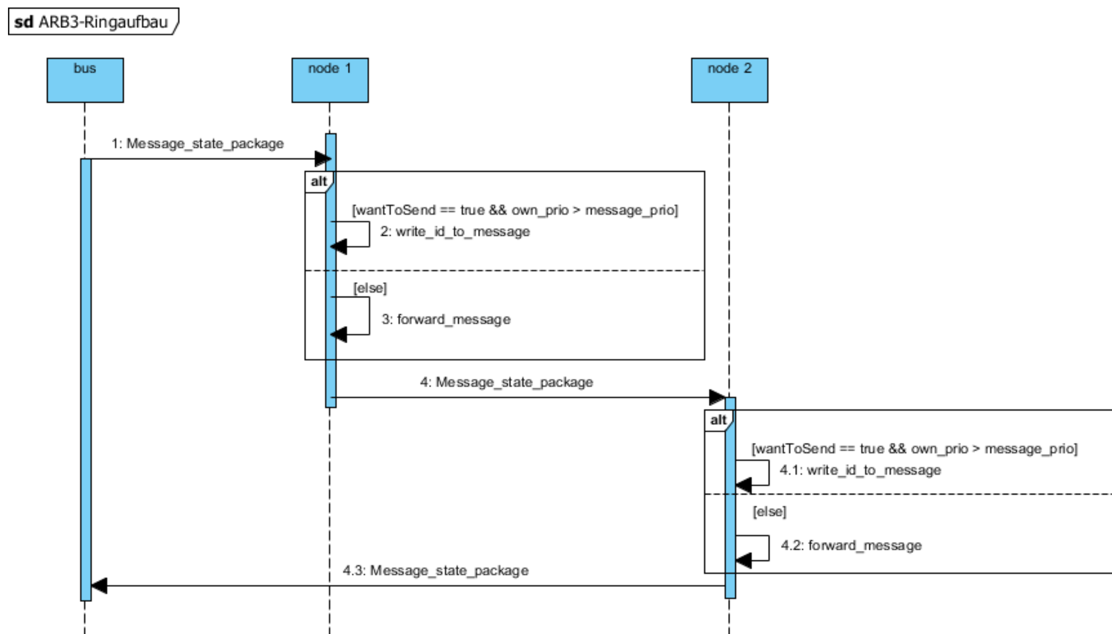

Abbildung 8: Arbitrierungsansatz 3 – Beispielhafter Ringverbund mit 2 Knoten

Der Vorteil dieses Algorithmus wäre das Einsparen von Nachrichten. Man muss nur eine einzige Nachricht erstellen, die von allen Teilnehmern betrachtet und gegebenenfalls bearbeitet wird. Der Aufwand ist allerdings fast so zu betrachten, als müsse von jedem Teilnehmer eine neue Nachricht verfasst werden. Anhand der obigen Formel ist aber ersichtlich, dass diese Methode effizienter als der zweite Ansatz arbeitet und mit jedem weiteren Teilnehmer eine Nachricht mehr einspart.

Bei geringen Buslasten besteht weiterhin das Problem des hohen Nachrichtenaufkommens, freie Bitzeiten sind also nach wie vor der größte Problem-Faktor.

Verschicken des Zustandes an die Knoten

Dieser Ansatz versucht, im Vergleich zu den vorigen Herangehensweisen, bei geringen und hohen Buslasten gleichermaßen den Nachrichtenaufwand zu dezimieren. Um dies zu realisieren, schickt der Bus bei Eintreten in den „Idle“-Zustand einmalig eine Nachricht an alle Knoten. Wenn ein Knoten senden will, muss er überprüfen, ob der Bus ihm die Nachricht zur Signalisierung des „Idle“-Zustandes geschickt hat. Natürlich müsste der Bus auch eine Nachricht schicken, wenn er aus dem Zustand „Idle“ in „Busy“ wechselt.

$$Nachrichtenaufwand = \left(Teilnehmerzahl \cdot 2 + Teilnehmer_{sendewillig} + 1\right)$$

Der Bus schickt an jeden Knoten die Nachricht, dass er sich nun im Zustand „Idle“ befindet. Wenn ein Teilnehmer senden möchte, schickt er die entsprechende Nachrichten-ID an den Bus-Knoten. Dort wird die niedrigste ID (höchste Priorität) ausgewählt und dem zugehörigen Knoten per Nachricht die Sendeberechtigung erteilt. Abb. 9 zeigt den Verlauf.

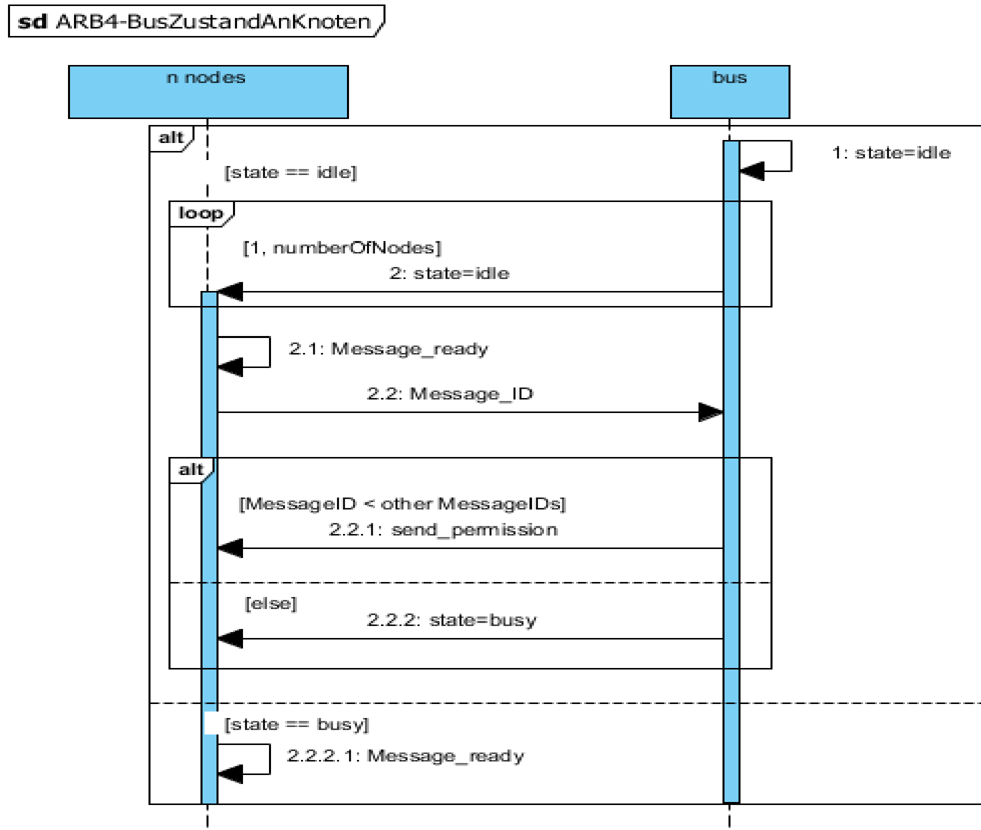

Abbildung 9: Arbitrierungsansatz 4 – Der Bus schickt seinen Zustand an die Knoten

Anhand des Nachrichtenaufwands ist ersichtlich, dass dieser Algorithmus deutlich effizienter als die vorigen Methoden arbeitet. Es müssen keine Nachrichten gesendet werden, wenn kein Sendewunsch bei den Knoten besteht. Das Problem der freien Bitzeiten ist damit ebenfalls gelöst.

Zentrale Vorlagerung der Nachrichten-IDs

Die fünfte Möglichkeit verschärft die Methode ihres Vorgängers. Sie nimmt bei Aufkommen eines Sendewunsches keine Rücksicht auf den virtuellen Zustand des Busses. Der Knoten schickt seinen Sendewunsch sofort und nur ein einziges Mal an den Bus. Dieser führt die

Sendewünsche der Knoten in einer Queue und arbeitet diese immer dann ab, wenn der Bus frei wird. Der Bus schickt zur Bestätigung der Sendeberechtigung noch eine Nachricht an den einen Teilnehmer, ansonsten ist der Nachrichtenaufwand aber sehr gering.

$$Nachrichtenaufwand = Teilnehmer_{neuer_Sendewunsch} + 1$$

Für jeden Sendewunsch wird also nur eine Nachricht geschickt und ist danach laufend aktuell. Abb. 10 veranschaulicht den Vorgang.

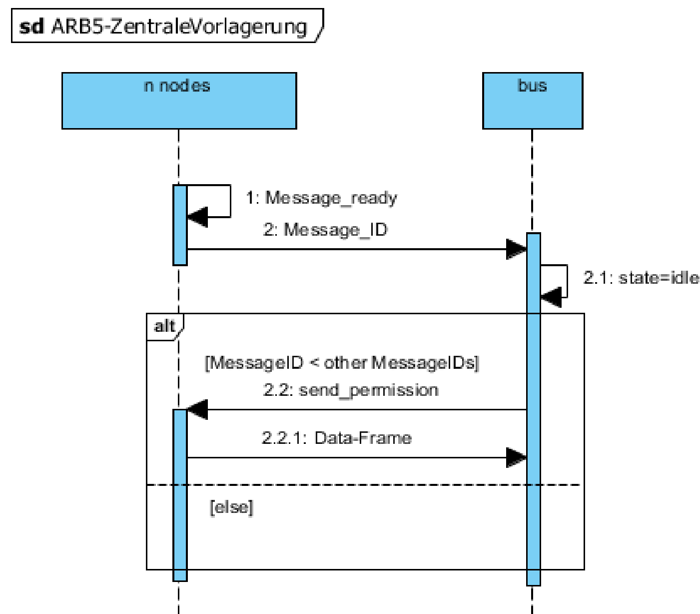

Abbildung 10: Arbitrierungsansatz 5 – Knoten schickt seinen Sendewunsch an den Bus

Der Nachrichtenaufwand in diesem Verfahren ist am geringsten, also sollte dieser Algorithmus mit der höchsten Performanz arbeiten.

Da jeder Knoten nur eine begrenzte Zahl an Message-Buffern besitzt, kann es sein, dass er die Sendung einer Nachricht abbrechen muss, um eine wichtigere Nachricht abzuschicken. Für einen solchen Fall gibt es noch die Möglichkeit, die eigene Nachricht wieder „abzumelden“. Auch dafür schickt der Knoten eine Nachricht an den Bus. Da, bei großzügig gewählten Perioden für das Versenden von Nachrichten, dieser Fall schätzungsweise nicht sehr oft auftritt, ist der zusätzliche Nachrichtenaufwand hierfür aber zu vernachlässigen.

Im Vergleich zu den anderen Lösungsansätzen garantiert dieses Vorgehen den geringsten Nachrichtenaufwand. Dies führt zum höchsten Durchsatz, also zur effizientesten Möglichkeit, die Arbitrierung auf ein Modell aus Nachrichten abzubilden. Aus diesem Grund wird in der Simulation also die zentrale Vorlagerung der Nachrichten-IDs implementiert.

4.4 Nachrichtenarten

Ein CAN-Netzwerk unterstützt vier verschiedene Arten von Nachrichten (Frames). Nähere Informationen hierzu in 2.2.2. Diese Nachrichten bzw. deren Funktionen müssen in der Simulation abgebildet werden. Es wird im Folgenden für jeden Frame gezeigt, welche Eigenschaften er hat und welche Aufgaben er in der Simulation ausübt.

4.4.1 Data-Frame

Die wichtigste Nachricht ist der Data-Frame. Durch ihn geschieht die Kommunikation zwischen den Knoten. Er übermittelt Daten, die andere Knoten auslesen können. Jeder Data-Frame hat eine Nachrichten-ID und ist einem Sender zugeordnet. Da hier kein höheres CAN-Protokoll abgebildet wird, wird nicht überprüft, ob eine Nachrichten-ID von mehreren Knoten verwendet wird. Wenn dies der Fall sein sollte, gibt es bei gleichzeitiger Arbitrierung solcher Nachrichten Probleme, daher wird die Simulation angehalten.

Die Länge des Data-Frames ist entscheidend für die Zeit, die die Nachricht braucht, um übermittelt zu werden. Der Frame setzt sich aus vielen einzelnen Teilen zusammen (s. Abb. 2). Diese bilden dann ein Paket, das von dem Knoten auf den Bus gelegt wird. Da diese Simulation aber nicht jedes Bit einzeln überträgt, sondern eine einzige Nachricht für den gesamten Data-Frame schickt, wird der Frame in zwei Teile aufgeteilt:

- Der Payload, also die enthaltenen Daten
- Der Overhead

Es ist wichtig, die exakte Länge des Frames zu kennen, damit die Simulationszeit korrekt arbeitet. Andererseits ist es nicht wichtig, welche Informationen der Overhead eines Data-Frames enthält. Die Ausnahme ist hier die ID der Nachricht, diese wird mit übermittelt. Der Overhead wird aufaddiert und bildet in Summe mit den Daten die Länge des Frames. Für den Data-Frame wird also neben den Daten und der ID die Länge des gesamten Frames mitgeschickt. Eine weitere Ausnahme für die Teile des Overheads gibt es allerdings noch: Das RTR-Bit. Dieses wird zusätzlich zu der Länge, der ID und den Daten als Information in Nachricht eingefügt und nimmt für einen Data-Frame den Wert 0 an.

4.4.2 Remote-Frame

Der Remote-Frame ist eine Erweiterung des Data-Frames. Für den Remote-Frame wird das oben genannte RTR-Bit = 1 gesetzt. Der Remote-Frame enthält dieselben Informationen, die der entsprechende Data-Frame enthält – ausgenommen des Data-Fields, dieses muss lediglich die gleiche Länge haben. Wenn ein Remote-Frame gleichzeitig die Arbitrierung mit dem zugehörigen Data-Frame durchläuft, so wird der Data-Frame die Arbitrierung erfolgreich beenden und der Knoten des Remote-Frames seinen Sendewunsch stornieren. Der Bus wird bei erfolgreicher Arbitrierung einer Nachrichten-ID alle zugehörigen Remote-Frames aus seiner Liste löschen.

4.4.3 Error-Frame

Ein Fehler beim Senden oder Empfangen eines Frames wird dem Bus von den betroffenen Knoten durch den Error-Frame signalisiert. Wenn der Error-Frame an alle Teilnehmer übermittelt wurde, verschickt der Bus an den Sendeknoten der entsprechenden Nachricht erneut die Sendeerlaubnis. Die Länge des Error-Frames sind 12 Bit (= maximale Belegung des Busses durch Error-Frames).

4.4.4 Overload-Frame

Der Overload-Frame wird in der Simulation nicht benutzt.

4.5 Versendung von Nachrichten

Dieser Abschnitt gibt eine Übersicht über die Abläufe der Versendungen von Nachrichten. Die Abläufe unterscheiden sich je nach Konfiguration des gesamten Systems (s. 4.6). Es wird ebenso gezeigt, wie die Simulationszeit gesteuert wird, die stellvertretend für die reale Zeit in einem CAN-Netzwerk steht. Die Simulationszeit wird für Analyse-Zwecke benutzt.

Bei Ablauf des internen Timers oder eingehenden Remote-Frames versucht der Knoten zu senden. In den Unterkapiteln wird inhaltlich zwischen der Möglichkeit „kein ACK-Bit“ und „ACK-Bit aktiviert“ (s. 4.6.2) unterschieden.

4.5.1 Versendung eines Data-Frames, Fehler deaktiviert

Versendung ohne ACK-Bit:

Abb. 11 zeigt den Ablauf der Versendung ohne ein ACK-Bit.

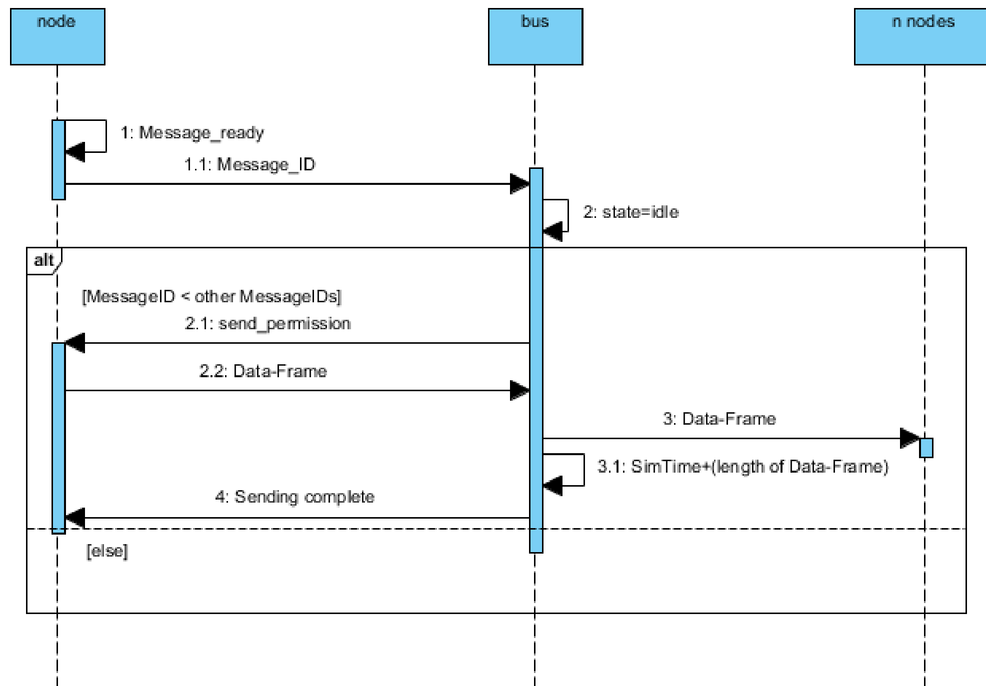

Abbildung 11: Versendung einer Nachricht ohne ACK und ohne Fehler

1. Arbitrierung:

Entsprechend des gewählten Ansatzes der Arbitrierung aus 4.3 verschickt der Knoten die ID der Nachricht an den Bus. Dieser schreibt die ID in eine Prioritäts-Queue.

2. Nachricht auf den Bus legen:

Wenn die Nachrichten-ID in der Prioritäts-Queue des Busses an oberster Stelle steht, wird die Sendeberechtigung an den Knoten geschickt. Der schickt ohne Zeitverzögerung die Nachricht an den Bus. Der Vorgang der Rechtevergabe beansprucht keine Simulationszeit. Um bei gleichzeitiger Arbitrierung mehrerer IDs zu garantieren, dass alle berücksichtigt werden (der Bus könnte die Sendeerlaubnis sonst zu früh verschicken), findet die Verschickung der Sendeerlaubnis eine Bitzeit später statt. Wenn die Nachrichten-ID des Knoten nicht die oberste Priorität hat, muss er warten, bis alle wichtigeren Nachrichten komplett verschickt worden sind.

3. Weiterleiten der Nachricht:

Der Bus verteilt die Nachrichten auf die anderen Teilnehmer, indem er an jeden eine Kopie schickt. Die Teilnehmer selbst regeln über ihre Message-Buffer, ob diese Nachricht für sie interessant oder nicht ist.

4. Meldung an den Sender:
Es wird weder vom Sender ein ACK-Bit erwartet, noch sind Fehler in irgendeiner Weise möglich. Daher verschickt der Bus nach Ablauf der Zeit, die der Data-Frame zur Übermittlung benötigt, die Bestätigung, dass die Nachricht vollständig übermittelt wurde.

Versendung mit ACK-Bit:

Wenn ein ACK-Bit vom Sender erwartet wird, muss die Nachricht überprüft werden, bevor der Bus die Bestätigung an den Sender schickt. Die Arbitrierungsphase unterscheidet sich nicht von der oben genannten Methode ohne ACK-Bit. Abb. 12 zeigt den Verlauf der Sendung.

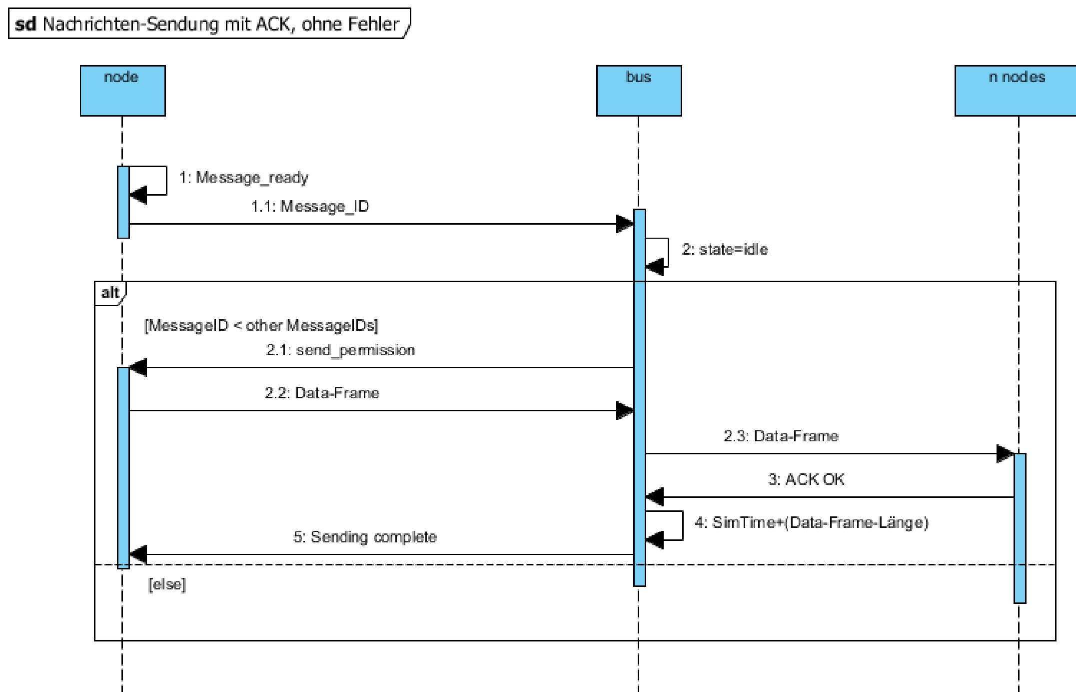

Abbildung 12: Versendung einer Nachricht mit ACK und ohne Fehler

1. Arbitrierung (s. unter Abbildung 11)
2. Nachricht auf den Bus legen:
Wenn der Sender die Nachricht auf den Bus legt, sendet er den Data-Frame. Dieser wird vom Bus empfangen und Kopien an jeden Teilnehmer geschickt.
3. Akzeptieren des Frames durch Empfänger:
Die Empfänger überprüfen, ob sie die Nachricht richtig empfangen. Sie können einen Fehler beim Lesen ermitteln. Da die Fehler in dieser Variante aber deaktiviert sind, kann

kein Fehler in der Nachricht auftreten. Dementsprechend werden die Empfänger den korrekten Empfang des Daten-Frames mit dem ACK-Bit quittieren. Das ACK-Bit ist Bestandteil des Daten-Frames. Um eine aufgeteilte Sendung des Data-Frames zu verhindern, wird erst der gesamte Data-Frame geschickt und zum Simulationszeitpunkt des ACK-Bits eine Nachricht für das ACK-Bit von den Empfängern erwartet. Alle Teilnehmer, die einen Fehler entdeckt haben, senden eine negative ACK-Nachricht.

4. Ende des Frames:
Wenn der Bus mindestens eine positive ACK-Nachricht empfangen hat, kann er annehmen, dass die Nachricht richtig empfangen wurde. Bei dieser Konfiguration ist es nicht möglich, dass ein empfangender Teilnehmer kein positives ACK-Bit schickt. Daher ist es aus Performanz-Gründen nicht empfehlenswert, bei deaktivierten Fehlern ein ACK-Bit zu aktivieren (s. 7.2.2).
5. Meldung an den Sender:
Der Bus verschickt nach Ablauf der Zeit, die der Data-Frame zur Übermittlung benötigt, die Bestätigung, dass die Nachricht vollständig übermittelt wurde.

4.5.2 Fehlersignalisierung

Bei Auftreten eines Fehlers müssen Sender und Empfänger bewerkstelligen, dass der betroffene Data-Frame erneut gesendet wird. Der Bus muss zu diesem Zweck einen Fehler registrieren und zum richtigen Zeitpunkt die erneute Sendung beantragen. Es ist geboten, eine möglichst effiziente Art zu finden, wie ein solcher Fehler signalisiert und die Simulationszeit dabei weiterhin korrekt abgebildet wird. Im Folgenden wird dies für Fehler bei Sender und Empfänger diskutiert.

I. Signalisierung zu Fehlerzeit

Die nächstliegende Methode umfasst, dass der Knoten bei Auftreten einer dieser beiden Fehler einen Error-Frame an den Bus schickt. Dies entspricht dem Vorgehen in einem realen CAN-Netzwerk. Der Teilnehmer schickt einen Error-Frame genau zu dem Zeitpunkt, an dem der Fehler in der Übertragung auftritt. Der Bus schickt nun keine Nachricht, die die Übertragung abschließt, an den Sender, sondern fordert ihn stattdessen auf, seine Sendung zu wiederholen.

Die Problematik an diesem Vorgehen ist die Art der Produktion von Fehlern in den Knoten. Hierfür wird am Anfang der Übertragung eines Data-Frames von Sender und Empfänger(n) nämlich festgelegt, ob ein Fehler auftreten wird und wann (Zeitpunkt im Frame) dies geschehen soll. Dabei ist dieser Fehler natürlich dem aktuellen Data-Frame zugeordnet. Wenn ein Knoten einen Fehler zu Zeitpunkt 10 meldet und ein Anderer wiederum zu Zeitpunkt 20 seinen Fehler melden sollte, so müsste der dafür aufgesetzte Timer unterbrochen werden. Wenn dies nicht geschieht, würde der zweite Knoten seinen Fehler bei der Sendewiederholung des Data-Frames melden, obwohl der Fehler für den

originalen Data-Frame bestimmt war. Die Unterbrechung eines Timers ist allerdings ein Aufwand, der durch folgenden Ansatz behoben werden kann.

II. Signalisierung zum Start der Übertragung

Jeder Knoten, der einen Fehler im Laufe der Übertragung entdeckt, ermittelt diesen schon vor der Übertragung eines Data-Frames. Also schicken alle Knoten, die einen Fehler entdecken werden, einen Error-Frame mit der genauen Position (Zeitpunkt im Frame) des Fehlers. Der Bus überprüft nur zur Anfangszeit der Übertragung auf eingegangene Error-Frames und wählt den frühesten Zeitpunkt eines Fehlers in diesen Frames aus. Somit kann es keine Überschneidung mit weiteren Data-Frames geben und Timer werden überflüssig. Für die Simulation ist dieser Ansatz gewählt.

4.5.3 Versendung eines Data-Frames mit Fehler beim Sender

Es gibt verschiedene Fehler, die in der Übermittlung einer Nachricht auftreten können.
Dem sendenden Knoten werden folgende Fehler zugeordnet:

- Bit Error
- *Acknowledgement Error* (der Sender erkennt ihn)
- Form Error

Versendung ohne ACK-Bit:
Der Bit Error und der Form Error sind Fehler, die bei der Übermittlung durch den Sender überwacht werden, der Acknowledgement Error wird später behandelt. Wenn Bit- oder Form-Error auftreten, so schickt der Sender einen Error-Frame.

Die Sendung einer Nachricht wird in Abb. 13 gezeigt. Es wird die Möglichkeit einer erfolgreichen und nicht erfolgreichen Übertragung veranschaulicht. Der Arbitrierungsvorgang wird nicht gezeigt und war für den sendenden Knoten erfolgreich.

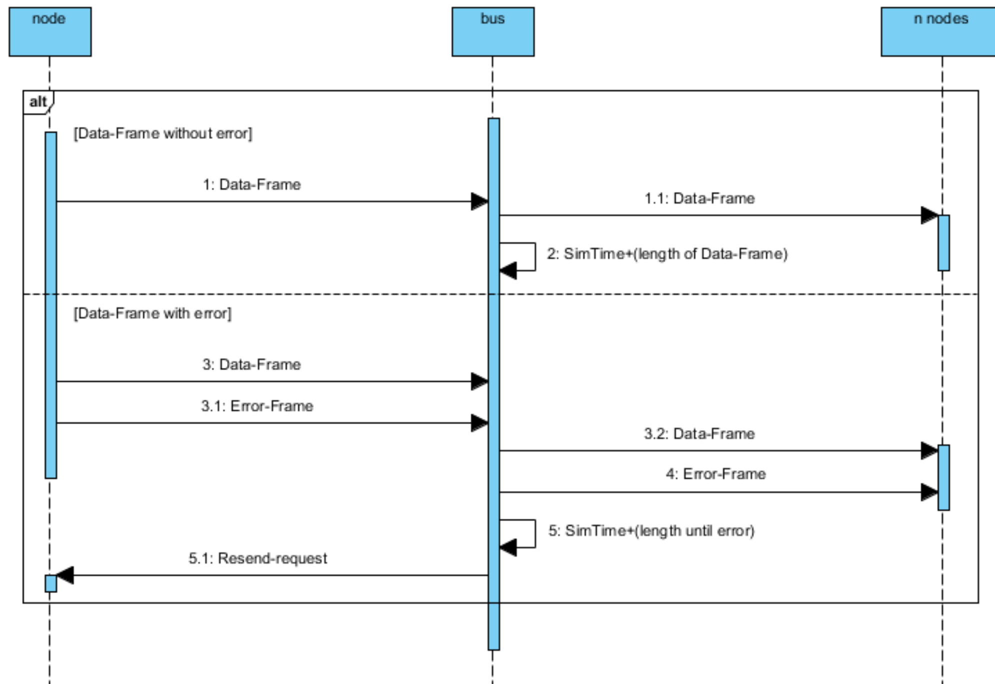

Abbildung 13: Nachrichtenübertragung mit Fehler beim Sender

1. Übertragung der Nachricht:
   Der Knoten schickt seine Nachricht an den Bus. Die Nachricht beinhaltet die Information, dass die Übermittlung ohne Fehler passiert, also kein Fehler durch den Sender verursacht wird.
2. Abschluss des Frames:
   Nach der Simulationszeit, die für die Versendung des Data-Frames nötig war, geht der Bus wieder in den Zustand „idle“ und verschickt eine Bestätigung der Übermittlung an den Sender (in der Abb. nicht dargestellt).

Fehlerfall:

3. Übertragung der Nachricht:
   Der Knoten übermittelt seine Nachricht. Unmittelbar danach verschickt er den Error-Frame. Der Bus überträgt die Nachricht trotzdem weiter an die anderen Knoten.
4. Weiterleitung des Error-Frames:
   Der Bus informiert alle Teilnehmer darüber, dass ein Fehler aufgetreten ist, in dem er den Error-Frame weiterleitet.
5. Anfrage für erneutes Senden:
   Der Bus addiert die Länge des Data-Frames abzüglich der noch nicht gesendeten Teile zu der Länge des aufgetretenen Error-Frames. Zu dem resultierenden Zeitpunkt informiert er den Sender, dass er den Data-Frame erneut schicken soll.

Versendung mit ACK-Bit:
Der dritte Fehler, der durch den Sender erkannt wird, ist der Acknowledgement Error. Er wird im Gegensatz zu den oben genannten Fehlern aber vom *Empfänger* verursacht. Er ist dem Sender deshalb nicht schon bei Versendung des Data-Frames bekannt.

Im Folgenden wird veranschaulicht, wie ein Fehler bei aktiviertem ACK-Bit auftritt. In einem CAN-Netzwerk ist dies dann der Fall, wenn kein Empfänger den Pegel auf dem Bus verändert. In der Simulation wird dieser Vorgang mit einer ACK-Nachricht von den Empfängern abgebildet. Ist alles in Ordnung schickt der Empfänger ein positives ACK. Sollte ein Fehler aufgetreten sein, dann schickt der Empfänger eine Nachricht mit negativem ACK. Der Bus leitet dies dann an den Sender-Knoten weiter, damit dieser seine Sendung erneut starten kann. Wichtig hierbei ist, dass ein negatives ACK einen rezessiven Pegel auf dem Bus darstellt. Dementsprechend reicht es aus, wenn ein einziger Knoten ein positives ACK schickt, um die Sendung als erfolgreich anzusehen. Abb. 14 zeigt den Ablauf ohne Arbitrierung.

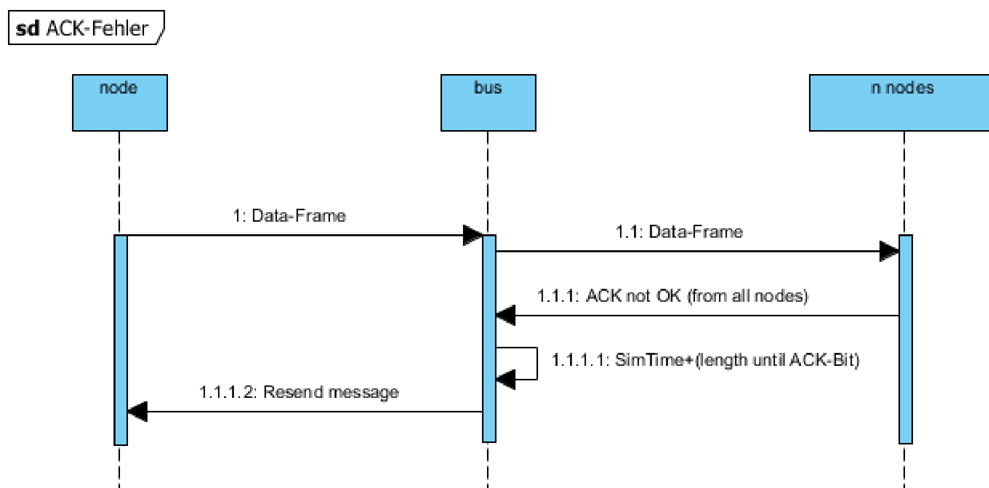

Abbildung 14: Acknowledgement-Error bei Nachrichtenübertragung

In Abbildung 14 ist nicht berücksichtigt, dass auch ein anderer Fehler bei Sender oder Empfänger auftreten kann.

4.5.4 Versendung eines Data-Frames mit Fehler beim Empfänger

Es gibt Fehler, die dem Empfänger zugeordnet werden. Diese sind:

- CRC Error
- *Acknowledgement Error* (der Empfänger verursacht ihn)
- Bit-Stuffing Error

Versendung ohne ACK-Bit:

Der CRC Error und Bit-Stuffing Error sind die beiden Fehler, die der Empfänger während der Übertragung eines Data-Frames erkennen kann. Der Empfänger muss also für diese Fehler dem Bus signalisieren, dass bei ihm ein Fehler aufgetreten ist. Dies tut er, nachdem er den Data-Frame empfangen hat. Der Acknowledgement Error wird auch vom Empfänger erkannt, wird allerdings separat behandelt.

Es kann vorkommen, dass mehrere Empfänger einen Fehler feststellen und diesen dem Bus signalisieren. In einem solchen Fall würde der Bus den am frühesten erkannten Fehler als Zeitpunkt für den Fehler wählen. In Abb. 15 werden die möglichen Fälle eines Fehleraufkommens betrachtet – die Arbitrierung ist bereits abgeschlossen.

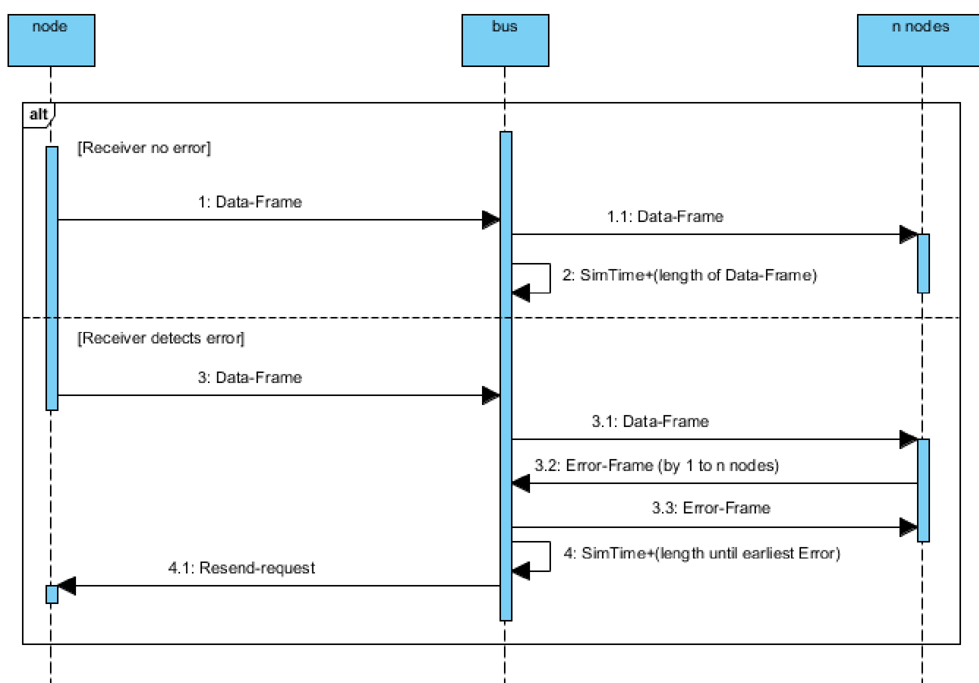

Abbildung 15: Fehler bei der Nachrichtenübertragung beim Empfänger

1. Übermittlung der Nachricht:
   Der Sender legt die Nachricht auf den Bus und dieser leitet sie an die anderen Knoten weiter.
2. Abschluss des Data-Frames:
   Es ist kein Fehler aufgetreten. Nach der Simulationszeit, die für die Versendung des Data-Frames nötig war, geht der Bus wieder in den Zustand „Idle“ und verschickt eine Bestätigung der Übermittlung an den Sender (in der Abb. nicht dargestellt).

Fehlerfall:

3. Übermittlung der Nachricht:
   Nach der Übermittlung der Nachricht erkennt mindestens einer der Empfänger einen Fehler und signalisiert dem Bus dies umgehend. In der Nachricht steht der Zeitpunkt, an dem der Fehler erkannt worden ist. Wenn mehrere Teilnehmer gleichzeitig einen Fehler in einem Frame erkennen, so wird der früheste Zeitpunkt der Fehlererkennung gewählt. Wenn der Sender ebenfalls einen Fehler erkannt hat, so wird dieser Fehlerzeitpunkt auch betrachtet (nicht im Diagramm gezeigt).
4. Anfrage für erneutes Senden:
   Der Bus addiert die Länge des Data-Frames abzüglich der noch nicht gesendeten Teile zu der Länge des aufgetretenen Error-Frames. Zu dem resultierenden Zeitpunkt informiert er den Sender, dass er den Data-Frame erneut schicken soll.

Versendung mit ACK-Bit:

Der Empfänger ist der Verursacher des Acknowledgement Errors. Der Empfänger schickt dafür nach Erhalt eines Data-Frames eine Nachricht „negatives ACK-Bit“ an den Bus. Wenn kein Empfänger ein positives ACK-Bit geschickt hat, signalisiert der Bus dem Sender, dass er den Data-Frame erneut senden muss. Ein Ablauf für einen solchen Fehler ist in Abbildung 14 zu sehen.

4.6 Konfigurationsmöglichkeiten

Die Simulation bietet mehrere unterschiedliche Konfigurationen. Dies bezieht sich zum einen auf grundsätzliche und erweiterte Funktionen, die auch einen Einfluss auf die Performanz haben, zum anderen auf die Angaben für den integrierten Stimuli-Generator. Während die Funktionen für das gesamte System – also alle Knoten – gelten, sind die Angaben im Stimuli-Generator für jeden Knoten einzeln bestimmt.

4.6.1 Grundsätzliche Funktionen

Die folgenden Funktionen werden von der Simulation immer unterstützt, es gibt keine Möglichkeit sie zu deaktivieren:

- **Arbitrierung**
  Ohne die Arbitrierung würde die prioritäts-gesteuerte Rechteverteilung ausfallen. Die Knoten wüssten nicht, wann sie senden dürfen.

- **Nachrichtenübermittlung**
  Die Simulation ist darauf ausgelegt, die Nachrichtenübermittlung zu simulieren, also ist diese natürlich unabdingbar.
- **Übermittlungsbestätigung**
  Für Erweiterungen der Simulation wird eventuell die Bestätigung gebraucht, deshalb kann auch sie nicht wegfallen.
- **Stimuli-Generator**
  Momentan ist der Stimuli-Generator ein integraler Bestandteil der Simulation, daher muss die Option „trafficGen“ aktiviert sein. Wenn Erweiterungen (zum Beispiel die Steuerung der Knoten durch externen Code) die Nutzung überflüssig machen, kann dies angepasst werden.
- **Externer Code**
  Es gilt das gleiche wie für den Stimuli-Generator. Ohne Erweiterung ist diese Option zu deaktivieren.

Grundfunktionen, die *beeinflusst*, aber *nicht deaktiviert* werden können:

- **Bandbreite**
  Sie bestimmt die Geschwindigkeit des Busses. Es sind Bandbreiten zwischen 10kbps(=10000) und 1Mbps(=1000000) möglich. Wäre ein Data-Frame 100Bits lang, würde die Versendung bei 10kbps wie folgt berechnet:

$$Übertragungszeit_{100/10kbps} = \frac{100Bit}{10000kbps} = 0.01s$$

- **CAN-Spezifikation**
  Gültig sind die Angaben 2.0A oder 2.0B.
- **Anzahl Message-Buffer**
  Es muss für die Knoten eine festgelegte Anzahl an Message-Buffern geben.
- **Bit-Stuffing-Modus**
  Die Stuffing Bits beeinflussen die Länge eines Data-Frames. Es muss eine der folgenden Optionen gewählt werden:
  - 0: Keine Stuffing Bits (Kürzeste Nachrichtenlänge)
  - 1: Worst Case (Maximale Länge mit Stuffing Bits)
  - 2: Verteilung: **percentage** entscheidet über die Häufigkeit von Stuffing Bits
  - 3: Original (Berechnung mit Bit-Reihenfolge)
- **Anzahl Knoten**
  Hier muss die exakte Anzahl an Knoten angegeben werden, dies ist für die Initialisierung des Netzwerkes zwingend notwendig.

Es gibt erweiterte Funktionen, die für die Ausführung der Simulation nicht notwendig sind. Sie sind modular zuschaltbar, haben aber direkten Einfluss auf die Performanz des Systems.

4.6.2 Erweiterte Funktionen

Die erweiterten Funktionen können aktiviert oder deaktiviert werden. Sie beeinflussen den Ablauf der Nachrichtenversendung. Der Einfluss dieser Einstellungen auf die Performanz des Systems wird in 7.2 untersucht. Die erweiterten Funktionen sind ACK-Bit, Fehler und Payload.

- **ACK-Bit**
  Das Aktivieren dieser Option hat zur Folge, dass jedem versendeten Data-Frame eine Bestätigung durch die Empfänger folgen muss.
  Dies passiert durch ein Paket, das stellvertretend für das ACK-Bit geschickt wird. Details zum Sendeablauf bei aktiviertem ACK-Bit in 4.5.1. Die Nachrichtenzahl erhöht sich deutlich, da nach der Verteilung des Data-Frames auf die Knoten von jedem Empfänger eine ACK-Nachricht generiert wird. Abb. 12 und 14 zeigen die Verläufe für positive oder negative ACK-Nachrichten.
- **Payload**
  Wie unter 4.4.1 beschrieben teilt sich der Data-Frame in zwei Teile auf: Overhead und *Payload*. Der Payload wird in einem Data-Frame an der Stelle des Datenfeldes übertragen (s. Abb. 1). Unabhängig von der Länge des Datenfeldes (für einzelne Nachrichten konfigurierbar) kann nun unterschieden werden, ob der Inhalt eines Datenfeldes von ankommenden Nachrichten richtig ausgelesen werden muss oder nur die Länge des Datenfeldes übergeben wird. Bei Deaktivierung muss eine Nachricht kein Inhalt im Datenfeld enthalten. Auf die Funktionalität der Simulation hat diese Option keine Auswirkung, da die Simulationszeit unter beiden Einstellungen gleich funktioniert.
  Einen Unterschied macht diese Option dann aus, wenn die Stuffing Bits des Frames originalgetreu berechnet werden sollen (s. 4.6.1).
- **Fehler**
  Die Aktivierung dieser Option hat zur Folge, dass alle Knoten zu einer gewissen Wahrscheinlichkeit (im Stimuli-Generator konfigurierbar) Fehler bei Data-Frame-Übertragungen entdecken werden. Wenn ein Fehler beim Lesen oder Schreiben einer Nachricht auftritt, so kann dies zu jedem Zeitpunkt innerhalb der Übermittlung der Daten auf den Bus geschehen. Wenn ein Knoten einen Fehler nach einigen Bits entdeckt und dementsprechend ein Error-Frame auf den Bus legt, muss die Sendung des Daten-Frames unterbrochen und wiederholt werden.
  Die Wiederholung eines Frames erzeugt einen zusätzlichen Nachrichtenaufwand und erhöht durch längere Belegung des Busses die Buslast. Mehrere fehlerhafte Knoten können die Effizienz und Buslast also sehr schnell verschlechtern.

4.6.3 Stimuli-Generator

Der Stimuli-Generator bestimmt für jeden Knoten Anzahl und Inhalt seiner Nachrichten.

- **Nachrichten**
  Für jeden Knoten kann eine beliebig hohe Anzahl an Nachrichten erstellt werden. Für jede einzelne Nachricht müssen dann folgende Parameter angegeben werden:
  - Nachrichten-ID (bestimmt die Priorität der Nachricht)
  - Periode (in Millisekunden)
  - Daten (bei aktiviertem Payload muss hier eine Zeichenkette angegeben werden, bei deaktiviertem Payload genügt eine „0“ als Platzhalter)
  - Modus (Data-Frame, Remote-Frame, Empfangener Data-Frame)
  - Datenlänge (Bei deaktiviertem Payload, ansonsten unwichtig)

  Bei Modus „Empfangener Data-Frame“ sind alle anderen Angaben unwichtig, da der Knoten nur die Nachrichten-ID registriert, um bei Empfang einer entsprechenden Nachricht diese zu verarbeiten.

  Die Anzahl an Message-Buffern beeinflusst die Initialisierung des Knotens. Wenn er nicht genügend Puffer für jede Nachricht zur Verfügung hat, so wird er einen Puffer allgemeingültig konfigurieren, so dass entweder alle Nachrichten-IDs versendet oder empfangen werden können.
- **Fehler**
  Für jeden Knoten wird das Auftreten von Fehlern individuell konfiguriert. Es gibt die Möglichkeit, eine Nachricht falsch zu lesen oder falsch zu schicken. Dies wird bei der Erstellung des Knotens (prozentual) festgelegt und hängt auch von der Konfiguration des ganzen Systems ab (ob Fehler aktiviert sind). Die angegebene Prozentzahl beeinflusst das Fehleraufkommen für Sende-, Empfangs- und ACK-Fehler gleichermaßen.
- **Sonstiges**
  Für jeden Knoten lässt sich ein Name bestimmen. Die Anzahl der konfigurierten Nachrichten muss angegeben werden, damit der Stimuli-Generator diese richtig initialisieren kann.

4.6.4 Verbindungen

Die Verbindungen der Netzwerke und der Knoten zu den Bussen werden vorkonfiguriert. Es gibt die Möglichkeit, das generische Netzwerk „Canbus“ zu verwenden, oder ein eigenes Netzwerk zu entwerfen. Zu beachten ist hier, dass die Verbindungen folgende Eigenschaften haben müssen:

- Jeder Knoten ist per inout-gate mit dem Bus verbunden
- Die Knoten sind untereinander nicht verbunden
- Zusätzliche Verbindungen (z.B. für Erweiterungen) werden an den Bus angelegt

5 Umsetzung

Dieses Kapitel beschreibt die Implementierung der Simulation. Anfangs wird der Aufbau des Programms anhand eines Klassendiagramms erläutert. Dessen Komponenten werden dann anschließend kurz erklärt und ggf. in die zugehörigen Teile des Konzepts eingeordnet.

5.1 Übersicht

Die beiden Hauptmodule der Simulation sind CAN_Node und CAN_Bus. Abb. 16 zeigt das Klassendiagramm.

CAN_Node:
Für jedes Netzwerk muss mindestens einer dieser Knoten vorhanden sein. Er agiert als Teilnehmer des Netzwerkes, das über einen CAN_Bus-Knoten verbunden wird. Die Nachrichten, die von CAN_Bus verarbeitet und weitergeleitet werden, sind von einem dieser Knoten generiert worden. Er ist nach dem in 4.2.1 gezeigten Teilnehmer realisiert worden.

Implementiert ist CAN_Node als simple cModule. Unter OMNeT++ hat dies zur Folge, dass er einige Methoden erbt, die für die Verarbeitung der Nachrichten nötig sind, oder bei Interaktion mit der graphischen Oberfläche aufgerufen werden (`handle_message`, `finish`). CAN_Node hat Zugriff auf die Objekte Buffer, Stats und MOB.

CAN_Bus:
Dieser Knoten agiert als zentraler Controller im Netzwerk. Er sorgt für die Vergabe von Senderechten und verteilt die ankommenden Nachrichten an andere Knoten (CAN_Node) weiter. Er entspricht dem in 4.2.2 vorgestellten Bus.

CAN_Bus ist ebenfalls als simple cModule implementiert, erbt damit die gleichen Methoden wie CAN_Node. CAN_Bus hat Zugriff auf das Objekt ID.

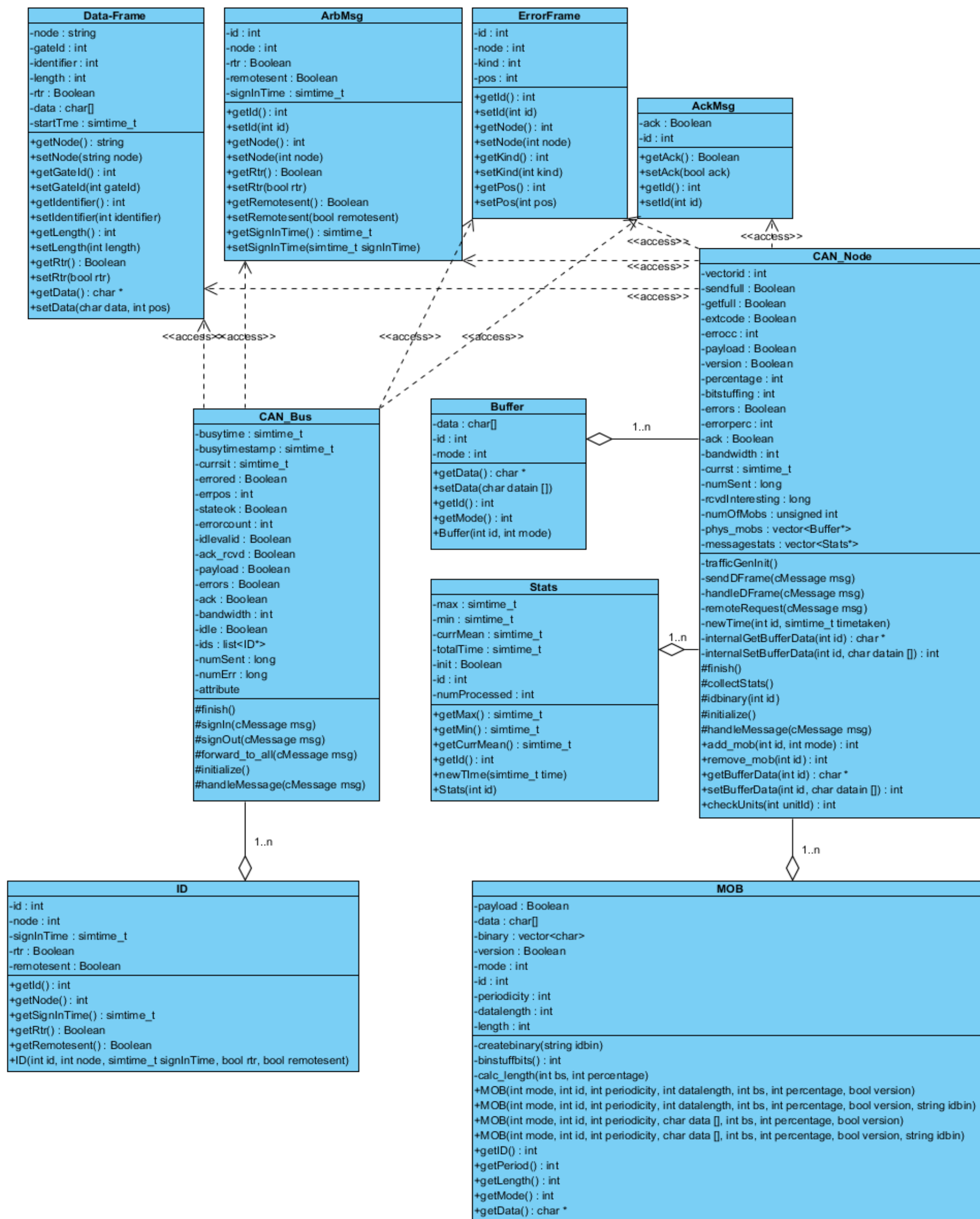

Abbildung 16: Klassendiagramm

5.2 Nachrichten und Informations-Container

Die Nachrichten der Simulation sind Objekte, die von der Klasse cMessage abgeleitet werden. OMNeT++ benutzt diese Klasse, um Nachrichten zwischen Modulen verschicken zu können. Die unter 4.4 gezeigten Nachrichtenarten werden mit den folgenden Klassen abgebildet:

- *Data-Frame*
- *ErrorFrame*

Das Nachrichten-Objekt *Data-Frame* wird als Data- oder Remote-Frame verschickt, unterschieden in der Variable rtr (vgl. 4.4.1 und 4.4.2). Zusätzlich ist der Data-Frame zur Steuerung der Kommunikation mit Attributen ausgestattet, die ihn einem sendenden Knoten zuordnen. Das Nachrichten-Objekt *ErrorFrame* ist keine Abbildung des realen Error-Frames. Seine Attribute ordnen den aufgetretenen Fehler einem Knoten zu und geben einen Zeitpunkt für den Fehler an. Die Länge eines Error-Frames ist für die Simulation auf 12 Bit festgelegt. Die beiden anderen Nachrichten-Objekte sind:

- *AckMsg*
- *ArbMsg*

Diese Nachrichten-Objekte werden von *CAN_Node*(s) und *CAN_Bus* geschickt, um die Arbitrierung zu steuern und Kommunikation zu gewährleisten, die sonst per Pegeländerung während einer Data-Frame-Übertragung geschieht. Eine *ArbMsg* wird für Informationen, die *CAN_Bus* für die Arbitrierungsliste (s. 4.3) benötigt, benutzt. Zusätzlich signalisiert eine *ArbMsg* einem *CAN_Node*, dass er senden darf. Eine *AckMsg* wird von *CAN_Node*(s) nach vollständigem Erhalt eines Data-Frames zur Bestätigung für den Sender verschickt (vgl. 4.5).

Die Informations-Container sind Objekte, die von *CAN_Node* und *CAN_Bus* zur Lagerung von Informationen benutzt werden. *CAN_Bus* benutzt Objekte der Klasse *ID*. Diese werden für den Arbitrierungsvorgang benutzt. Jede Nachrichten-ID, die von einem Knoten verschickt werden soll, wird von *CAN_Bus* in ein *ID*-Objekt geschrieben und in einer Liste verwahrt. Es ist auf diese Weise möglich, die Nachrichten-ID einem Knoten zuzuordnen. *CAN_Node* benutzt folgende Container:

- *MOB*
- *Stats*
- *Buffer*

MOB wird durch den Stimuli-Generator mit Informationen über zu sendende Nachrichten befüllt und durch *CAN_Node* immer dann ausgelesen, wenn diese Nachricht versendet werden soll. *Stats* dient zur Erfassung von Sendezeiten der Nachrichten. Wenn der Knoten auf den Empfang einer Nachricht konfiguriert ist, wird ein *Stats*-Objekt für diese Nachrichten-ID erstellt. *Buffer* ist ein Objekt, das extern ausgelesen (bei auf Empfang konfigurierten Nachrichten) oder verändert (bei auf Versendung konfigurierten Nachrichten) werden kann. Es enthält die Daten, die für diese Nachricht aktuell sind.

6 Qualitätssicherung

Für das Programm ist die Qualitätssicherung ein wichtiger Schritt, um Fehler in den Abläufen zu vermeiden. Die Tests beziehen sich auf die zentralen Funktionsblöcke der Simulation, die folgende Aspekte steuern:

- Die Arbitrierung
- Das Zeitverhalten während einer Nachrichten-Sendung
- Das Ein- und Auslesen von Daten
- Das Verhalten bei Fehlern
- Laufzeitfehler

Die ersten drei Testfälle sind ohne Fehler und ACK-Bit konfiguriert. Das heißt, dass die Sendung immer fehlerfrei und ohne Unterbrechungen funktionieren wird. Im dritten Testfall wird der Payload aktiviert. Für Testfall 4 werden Fehler eingeschaltet.

6.1 Testfall 1: Arbitrierung

Dieser Test überprüft, ob die Arbitrierung die niedrigste ID erkennt und dem zugehörigen Knoten die Sendeerlaubnis erteilt wird. Verwendet wird die Spezifikation CAN2.0B. Es kommunizieren fünf Knoten miteinander, die absteigend nach Priorität geordnet sind. Tabelle 1 zeigt, zu welchem Zeitpunkt die Knoten ihre Nachricht schicken wollen. Das heißt, dass sie ab diesem Zeitpunkt versuchen, ihre Nachricht mithilfe der Arbitrierung auf den Bus zu legen, wenn dieser frei wird.

Zeitpunkt(s)	Knoten 1	Knoten 2	Knoten 3	Knoten 4	Knoten 5
0.5	ID 1; ID 7				
0.51	ID 2	ID 10			
0.53	ID 3	ID 20	ID 100		
0.57	ID 4	ID 30	ID 200	ID 1000	
0.65	ID 5	ID 40	ID 300	ID 2000	ID 10000
0.651		ID 50			
0.652	ID 6				
0.654			ID 400		

Tabelle 1: Arbitrierungstest – Zeitpunkte für Sendewünsche von Knoten

Die IDs werden ihrer Priorität nach abgearbeitet. Die letzten drei Nachrichten (ID 50, ID 6, ID 400) werden nicht als letzte Nachrichten gesendet werden, da sie vorige IDs aufgrund ihrer höheren Priorität verdrängen. Sie sind in Tabelle 2 gelb unterlegt. Die Tabelle zeigt den erwarteten zeitlichen Ablauf, in dem die Nachrichten über den Bus verschickt werden. Der Bus verschickt die erste Sendeerlaubnis erst eine Bitzeit später, siehe hierzu 4.4.1 – Versendung ohne ACK-Bit. Die Bandbreite beträgt 100kbps. Die fett markierten Einträge aus Tabelle 2 werden in Abb. 17 als Events aus OMNeT++ gezeigt.

#	Zeitspanne(s)	Knoten 1	Knoten 2	Knoten 3	Knoten 4	Knoten 5
1	0.50001-0.50068	ID 1				
2	0.50069-0.50135	ID 7				
3	0.51001-0.51068	ID 2				
4	0.51069-5.51135		ID 10			
5	0.53001-0.53068	ID 3				
6	0.53069-0.53135		ID 20			
7	0.53136-0.53202			ID 100		
8	0.57001-0.57068	ID 4				
9	0.57069-0.57135		ID 30			
10	0.57136-0.57202			ID 200		
11	0.57203-0.57269				ID 1000	
12	0.65001-0.65068	ID 5				
13	0.65069-0.65135		ID 40			
14	0.65136-0.65202		ID 50			
15	0.65203-0.65269	ID 6				
16	0.65270-0.65336			ID 300		
17	0.65337-0.65403				ID 2000	
18	0.65404-0.65470			ID 400		
19	0.65471-0.65537					ID 1000

Tabelle 2: Arbitrierungstest – Erwarteter zeitlicher Ablauf der Nachrichten auf dem Bus

Zeitlicher Ablauf aus OMNeT++ (Stichproben):

```
** Event #24  T=0.50068  Canbus.node[0] (CAN_Node, id=3), on `SendingComplete' (ArbMsg, id=47)
Nachricht mit ID 1 vollständig verschickt
** Event #106  T=0.53135  Canbus.node[1] (CAN_Node, id=4), on `SendingComplete' (ArbMsg, id=159)
Nachricht mit ID 20 vollständig verschickt
** Event #171  T=0.57202  Canbus.node[2] (CAN_Node, id=5), on `SendingComplete' (ArbMsg, id=248)
Nachricht mit ID 200 vollständig verschickt
** Event #242  T=0.65202  Canbus.node[1] (CAN_Node, id=4), on `SendingComplete' (ArbMsg, id=340)
Nachricht mit ID 50 vollständig verschickt
```

** Event #256 T=0.65269 Canbus.node[0] (CAN_Node, id=3), on `SendingComplete' (ArbMsg, id=361)
Nachricht mit ID 6 vollständig verschickt
** Event #300 T=0.6547 Canbus.node[2] (CAN_Node, id=5), on `SendingComplete' (ArbMsg, id=425))
Nachricht mit ID 400 vollständig verschickt
** Event #314 T=0.65537 Canbus.node[4] (CAN_Node, id=7), on `SendingComplete' (ArbMsg, id=446)
Nachricht mit ID 10000 vollständig verschickt

Abbildung 17: Arbitrierungstest – Tatsächlicher zeitlicher Ablauf auf dem Bus

Die Stichproben zeigen, dass die Reihenfolge der Nachrichten in der Simulation in diesem Fall mit der erwarteten Reihenfolge übereinstimmt. Für die folgenden Testfälle wird nun also angenommen, dass die Arbitrierung korrekt funktioniert.

6.2 Testfall 2: Zeitverhalten

Das Zeitverhalten der Simulation spielt eine zentrale Rolle für die potentielle praktische Nutzung in der Analyse. Sollte das zeitliche Verhalten nicht stimmen, sind die Ergebnisse unbrauchbar.

Für diesen Test werden lediglich zwei Knoten im Netzwerk kommunizieren. Knoten 1 als Sender und Knoten 2 als Empfänger. Der Sender schickt seine Nachrichten ohne Unterbrechung hintereinander.
Die Nachrichten haben folgende (relevante) Eigenschaften:

- Nachricht 1: Datenlänge: 1 Bit; Gesamtlänge 55Bit
- Nachricht 2: Datenlänge: 0 Bit; Gesamtlänge 47Bit
- Nachricht 3: Datenlänge: 3 Bit; Gesamtlänge 71Bit
- Nachricht 4: Datenlänge: 0 Bit; Gesamtlänge 47Bit
- Nachricht 5: Datenlänge: 5 Bit; Gesamtlänge 87Bit
- Nachricht 6: Datenlänge: 0 Bit; Gesamtlänge 47Bit

Daraus ergibt sich folgender (erwarteter) zeitlicher Ablauf der Nachrichten (auch hier um eine Bitzeit versetzt) – Bandbreite 1Mbps:

- Nachricht 1: 0.050001 - 0.050056
- Nachricht 2: 0.050056 - 0.050103
- Nachricht 3: 0.050103 - 0.050174
- Nachricht 4: 0.050174 - 0.050221
- Nachricht 5: 0.050221 - 0.050308
- Nachricht 6: 0.050308 - 0.050355

** Event #23 T=0.050056 Canbus.node[0] (CAN_Node, id=3), on `SendingComplete' (ArbMsg, id=26)
Nachricht mit ID 1 vollständig verschickt
** Event #31 T=0.050103 Canbus.node[0] (CAN_Node, id=3), on `SendingComplete' (ArbMsg, id=38)
Nachricht mit ID 2 vollständig verschickt

** Event #39 T=0.050174 Canbus.node[0] (CAN_Node, id=3), on `SendingComplete' (ArbMsg, id=50)
Nachricht mit ID 3 vollständig verschickt
** Event #47 T=0.050221 Canbus.node[0] (CAN_Node, id=3), on `SendingComplete' (ArbMsg, id=62)
Nachricht mit ID 4 vollständig verschickt
** Event #55 T=0.050308 Canbus.node[0] (CAN_Node, id=3), on `SendingComplete' (ArbMsg, id=74)
Nachricht mit ID 5 vollständig verschickt
** Event #63 T=0.050355 Canbus.node[0] (CAN_Node, id=3), on `SendingComplete' (ArbMsg, id=86)
Nachricht mit ID 6 vollständig verschickt

Abbildung 18: Zeitverhaltenstest – OMNeT++ Events

Abb. 18 zeigt, dass die, von der Simulation ausgegebenen, Events in diesem Fall mit der vorher erwarteten zeitlichen Abfolge vollständig übereinstimmen. In den folgenden Testfällen wird also angenommen, dass das Zeitverhalten korrekt abgebildet wird.

6.3 Testfall 3: Payload

Der Sender kann in seinem Datenblock 0-8 Bytes an Daten verschicken. Die Simulation erlaubt daher für die Versendung Zeichenketten mit der Länge 0-8.

Der Testaufbau besteht erneut aus zwei Knoten, einem Sender und einem Empfänger. Der Sender verschickt nacheinander sechs Nachrichten, von denen für den Empfänger allerdings nur fünf interessant sind (sein Puffer ist auf Empfang für diese fünf Nachrichten gestellt). Die (relevanten) Eigenschaften der Nachrichten im Überblick:

- Nachricht 1: Daten: „Hallo“
- Nachricht 2: Daten: „dies“
- Nachricht 3: Daten: „ist“
- Nachricht 4: Daten: „ein“
- Nachricht 5: Daten: „guter“
- Nachricht 6: Daten: „Test“

Der Empfänger interessiert sich für Nachricht 1-4 und Nachricht 6.
In Abb. 19 werden die zugehörigen Events gezeigt, wobei der Sender (node[0]) und der Empfänger (node[1]) als Vergleich untereinander stehen.

** Event #8 T=0.055001 Canbus.node[0] (CAN_Node, id=3), on `message' (DataFrame, id=15)
Nachricht angekommen
Identifier = 1.
Daten: Hallo
** Event #9 T=0.055001 Canbus.node[1] (CAN_Node, id=4), on `message' (DataFrame, id=16)
Nachricht angekommen
Identifier = 1.
Daten: Hallo

```
** Event #19  T=0.056001  Canbus.node[0] (CAN_Node, id=3), on `message' (DataFrame, id=29)
Nachricht angekommen
Identifier = 2.
Daten: dies
** Event #20  T=0.056001  Canbus.node[1] (CAN_Node, id=4), on `message' (DataFrame, id=30)
Nachricht angekommen
Identifier = 2.
Daten: dies
** Event #30  T=0.057001  Canbus.node[0] (CAN_Node, id=3), on `message' (DataFrame, id=43)
Nachricht angekommen
Identifier = 3.
Daten: ist
** Event #31  T=0.057001  Canbus.node[1] (CAN_Node, id=4), on `message' (DataFrame, id=44)
Nachricht angekommen
Identifier = 3.
Daten: ist
** Event #41  T=0.058001  Canbus.node[0] (CAN_Node, id=3), on `message' (DataFrame, id=57)
Nachricht angekommen
Identifier = 4.
Daten: ein
** Event #42  T=0.058001  Canbus.node[1] (CAN_Node, id=4), on `message' (DataFrame, id=58)
Nachricht angekommen
Identifier = 4.
Daten: ein
** Event #52  T=0.059001  Canbus.node[0] (CAN_Node, id=3), on `message' (DataFrame, id=71)
Nachricht angekommen
Identifier = 5.
Daten: guter
** Event #53  T=0.059001  Canbus.node[1] (CAN_Node, id=4), on `message' (DataFrame, id=72)
Nachricht angekommen
Identifier = 5.
Diese Nachricht wird durch den Knoten nicht verarbeitet
** Event #63  T=0.060001  Canbus.node[0] (CAN_Node, id=3), on `message' (DataFrame, id=85)
Nachricht angekommen
Identifier = 6.
Daten: Test
** Event #64  T=0.060001  Canbus.node[1] (CAN_Node, id=4), on `message' (DataFrame, id=86)
Nachricht angekommen
Identifier = 6.
Daten: Test
```

Abbildung 19: Payload-Test – OMNeT++ Events, 5. Nachricht nicht verarbeitet

Der Empfänger verarbeitet also jeden ankommenden Data-Frame, liest aber nur bei IDs von Interesse den Inhalt aus. Die versendeten Daten kommen dazu korrekt beim Empfänger an. In diesem Test funktioniert die Datenübertragung in einer Nachricht also korrekt.

6.4 Testfall 4: Fehlerinjektion durch Stimuli-Generator

Den Anforderungen entsprechend kann der CAN-Bus Fehler bei den einzelnen Knoten verarbeiten. Für diesen Test werden zwei Aufbauten (per Stimuli -Generator) modelliert:

- Modell 1: Knoten 1 sendet fehlerfrei, Knoten 2 ist fehleranfällig (50%)
- Modell 2: Knoten 1 sendet fehleranfällig (50%), Knoten 2 empfängt fehlerfrei

Der Sender sendet im Zyklus drei Nachrichten, der Inhalt und die Eigenschaften der Nachrichten sind unwichtig. In beiden Modellen ist jeweils nur der Sender oder nur der Empfänger fehlerbehaftet. Da der Fehler eines Teilnehmers einen Fehler auf dem gesamten Bus zur Folge hat, sollte die gesamte Fehlerquote bei beiden Modellen bei etwa 50% liegen. Für aussagekräftige Ergebnisse sollte ein genügend großer Simulationszeitraum gewählt werden. Abb. 20 und 21 zeigen Ergebnisse nach kurzer und langer Simulationszeit.

Modell 1:

```
scalar Canbus.bus   #Simulated_Time     401.95
scalar Canbus.bus   %Busload     3.118005473317
scalar Canbus.bus   #Data-Frames     328253
scalar Canbus.bus   #Errors      164126
scalar Canbus.bus   %Errors      49.999847678468
```

Abbildung 20: Fehlertest – Fehlerhäufigkeit bei Empfangsfehlern - Kurze Zeit

```
scalar Canbus.bus   #Simulated_Time     2552.430001
scalar Canbus.bus   %Busload     3.119450091434
scalar Canbus.bus   #Data-Frames     2086016
scalar Canbus.bus   #Errors      1043776
scalar Canbus.bus   %Errors      50.036816592011
```

Abbildung 21: Fehlertest – Fehlerhäufigkeit bei Empfangsfehlern - Lange Zeit

```
** Event #52  T=0.01  Canbus.node[0] (CAN_Node, id=3), on `SendingPermission' (ArbMsg, id=66)   (1)
Send Permission achieved for mobId 1
** Event #53  T=0.01  Canbus.bus (CAN_Bus, id=2), on `message' (DataFrame, id=68)
Length: 47
** Event #54  T=0.01  Canbus.node[0] (CAN_Node, id=3), on `message' (DataFrame, id=70)
Nachricht angekommen
Identifier = 1.
** Event #55  T=0.01  Canbus.node[1] (CAN_Node, id=4), on `message' (DataFrame, id=71)
Nachricht angekommen
Identifier = 1.
** Event #56  T=0.01  Canbus.bus (CAN_Bus, id=2), on `geterror' (ErrorFrame, id=75)
** Event #57  T=0.01  Canbus.node[0] (CAN_Node, id=3), on `geterror' (ErrorFrame, id=78)
Fehler erkannt                                                                                 (2)
** Event #58  T=0.01  Canbus.node[1] (CAN_Node, id=4), on `geterror' (ErrorFrame, id=79)
Fehler erkannt
** Event #59  T=0.010012  Canbus.bus (CAN_Bus, id=2), on selfmsg `ErrResend' (ArbMsg, id=77)
** Event #60  T=0.010012  Canbus.bus (CAN_Bus, id=2), on selfmsg `ErrResendCompl' (ArbMsg, id=80)
** Event #61  T=0.010012  Canbus.node[0] (CAN_Node, id=3), on `SendingPermission' (ArbMsg, id=81)   (3)
Send Permission achieved for mobId 1
** Event #62  T=0.010012  Canbus.bus (CAN_Bus, id=2), on `message' (DataFrame, id=83)
Length: 47
** Event #63  T=0.010012  Canbus.node[0] (CAN_Node, id=3), on `message' (DataFrame, id=85)
```

Abbildung 22: Fehlertest – Erkannter Fehler bei Empfänger

Für das erste Modell (Abb. 20 und 21) ist ersichtlich, dass sich der prozentuale Fehlerwert an die 50% annähert. Abb. 22 zeigt den folgenden Fall:

- Absendung der Nachricht (1)
- Der Empfänger erkennt einen Fehler (2)
- Aufforderung vom Bus, die Nachricht erneut zu senden (3)

Modell 2:

```
scalar Canbus.bus   #Simulated_Time     2627.255103
scalar Canbus.bus   %Busload    3.119595919955
scalar Canbus.bus   #Data-Frames    2147334
scalar Canbus.bus   #Errors     1074541
scalar Canbus.bus   %Errors     50.040701632815
```

Abbildung 23: Fehlertest – Fehlerhäufigkeit bei Sendefehlern

```
** Event #6  T=0.005001  Canbus.node[0] (CAN_Node, id=3), on `SendingPermission' (ArbMsg, id=8)      (1)
Send Permission achieved for mobId 1
** Event #7  T=0.005001  Canbus.bus (CAN_Bus, id=2), on `senderror' (ErrorFrame, id=11)
** Event #8  T=0.005001  Canbus.bus (CAN_Bus, id=2), on `message' (DataFrame, id=10)
Length: 47
** Event #9  T=0.005001  Canbus.node[0] (CAN_Node, id=3), on `senderror' (ErrorFrame, id=13)
Fehler erkannt                                                                                        (2)
** Event #10  T=0.005001  Canbus.node[1] (CAN_Node, id=4), on `senderror' (ErrorFrame, id=14)
Fehler erkannt
** Event #11  T=0.005001  Canbus.node[0] (CAN_Node, id=3), on `message' (DataFrame, id=16)
Nachricht angekommen
Identifier = 1.
** Event #12  T=0.005001  Canbus.node[1] (CAN_Node, id=4), on `message' (DataFrame, id=17)
Nachricht angekommen
Identifier = 1.
** Event #13  T=0.005013  Canbus.bus (CAN_Bus, id=2), on selfmsg `ErrResend' (ArbMsg, id=12)
** Event #14  T=0.005013  Canbus.bus (CAN_Bus, id=2), on selfmsg `ErrResendCompl' (ArbMsg, id=22)
** Event #15  T=0.005013  Canbus.node[0] (CAN_Node, id=3), on `SendingPermission' (ArbMsg, id=23)    (3)
Send Permission achieved for mobId 1
** Event #16  T=0.005013  Canbus.bus (CAN_Bus, id=2), on `senderror' (ErrorFrame, id=26)
** Event #17  T=0.005013  Canbus.bus (CAN_Bus, id=2), on `message' (DataFrame, id=25)
Length: 47
```

Abbildung 24: Fehlertest – Erkannter Fehler bei Sender

Auch Modell 2 (Abb. 23) zeigt eine Annäherung an 50% Fehlerhäufigkeit. Abb. 24 zeigt den umgekehrten Fall zu Abb. 22:

- Der Sender erkennt einen Fehler (2)
- Aufforderung vom Bus, die Nachricht (aus (1)) erneut zu senden (3)

Für diesen Test bestätigt sich also zum einen, dass der Stimuli-Generator ausreichend genau die erwarteten Fehler injiziert, zum anderen, dass bei Auftreten eines Fehlers der Bus darauf reagiert, indem er den Knoten die Sendung der betroffenen Nachricht wiederholen lässt.

6.5 Laufzeitfehler

Die Simulation prüft beim Start die Konfiguration des Netzwerkes und die Eingaben des Stimuli-Generators. Hierzu gehören eine gültige Bandbreite, eine gültige CAN-Spezifikation, vollständige Angaben zu den Nachrichten, gültige IDs, Modus der Data-Frames, gültige Datenlängen, gültige Prozentangaben und gültiger Bit-Stuffing-Modus. Abb. 25 und 26 zeigen Fehler, die aus falschen Eingaben resultieren und den Abbruch der Simulation verursachen.

1. Allgemeine Konfigurationen (Bandbreite, CAN-Spezifikation, Bit-Stuffing):

Ungueltige Bandbreiteneinstellung
<!> Module (CAN_Node) Canbus.node[0] (id=3) during network initialization: Simulation stopped with endSimulation().

Keine gueltige Einstellung fuer Bitstuffing. Gueltiger Bereich: 0-3. Angegebener Wert: 4
<!> Module (CAN_Node) Canbus.node[0] (id=3) during network initialization: Simulation stopped with endSimulation().

Keine gueltige Einstellung fuer prozentuale Verteilung. Gueltiger Bereich: 0-100. Angegebener Wert: 123
<!> Module (CAN_Node) Canbus.node[0] (id=3) during network initialization: Simulation stopped with endSimulation().

Keine gueltige Einstellung fuer die Version. Gueltig: 2.0A oder 2.0B. Angegebener Wert: 80
<!> Module (CAN_Node) Canbus.node[0] (id=3) during network initialization: Simulation stopped with endSimulation().

Abbildung 25: Laufzeitfehler der allgemeinen Konfiguration

2. Stimuli-Generator Eingaben:

Daten sind zu Lang. Maximale Länge ist 8 Byte, diese Daten sind 10 Byte lang
<!> Module (CAN_Node) Canbus.node[0] (id=3) during network initialization: Simulation stopped with endSimulation().

ID 4536 ist nicht gueltig.
<!> Module (CAN_Node) Canbus.node[0] (id=3) during network initialization: Simulation stopped with endSimulation().
<!> Error in module (CAN_Node) Canbus.node[0] (id=3) during network initialization: Cannot schedule message (cMessage)0 to the past, t=-0.005.

Modus 5 ist nicht gueltig. Zahl zwischen 0 und 2 wählen.
<!> Module (CAN_Node) Canbus.node[0] (id=3) during network initialization: Simulation stopped with endSimulation().

Keine Dummy-Werte für Empfangsnachricht angegeben
<!> Module (CAN_Node) Canbus.node[1] (id=4) during network initialization: Simulation stopped with endSimulation().

Nicht genug Parameter für die Nachrichten von Node Knoten2 angegeben.
In omnetpp.ini unter Canbus.node[1].id ergänzen.
In omnetpp.ini unter Canbus.node[1].mode ergänzen.
In omnetpp.ini unter Canbus.node[1].periodicity ergänzen.
<!> Module (CAN_Node) Canbus.node[1] (id=4) during network initialization: Simulation stopped with endSimulation().

Mehr als ein Knoten will mit der selben ID senden
Da dies zu fortlaufenden Bit-Fehlern durch die versendenden Knoten führt, wird die Simulation angehalten
<!> Module (CAN_Bus) Canbus.bus (id=2) at event #10, t=0.003001: Simulation stopped with endSimulation().

Abbildung 26: Laufzeitfehler bei Falscheingabe in Stimuli-Generator

Da alle Fehler zum Stopp der Simulation führen, ist fast ausgeschlossen, dass das System falsch konfiguriert läuft. Es gibt weitere Laufzeitfehler, die durch die Konfiguration nicht beeinflusst werden und Abstürze der Simulation im Notfall verhindern.

7 Ergebnisse / Evaluierung

Ein Ziel der Bachelorarbeit ist die Darstellung eines vorgegebenen Modells und dessen korrekte und umfassende Analyse. Für diesen Zweck wird ein Modell von Philips zur Kommunikation zwischen einem Generator für Röntgengeräte (durch die sogenannte „AWS“ gesteuert) herangezogen. Diese Kommunikation verläuft in geregelten Zyklen. So werden die Teilnehmer des Netzes in vorher festgelegten Perioden Nachrichten verschicken.

Für die Evaluierung wird die Effizienz der Simulation im Hinblick auf verschiedene Konfigurationen des oben genannten Modells betrachtet.

7.1 Philips-Modell

Das Modell besteht aus acht miteinander kommunizierenden Knoten. Abb. 27 zeigt das komplette Netzwerk. Die einzelnen Knoten sind:

- AWS (Steuergerät)
- Generator
- Motoren x,y,z,alpha,beta und gamma (für die Ausrichtung des Röntgenstrahlers)

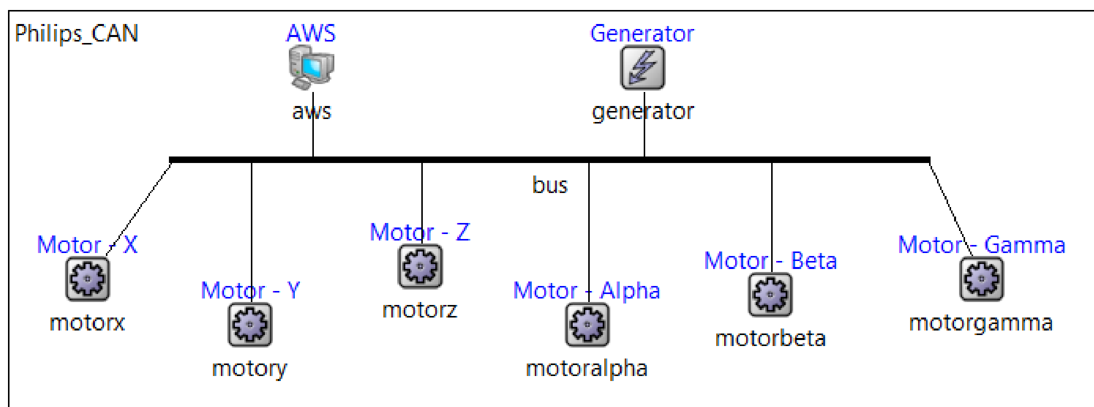

Abbildung 27: Philips-Modell

Die Kommunikation hat die folgenden Merkmale:

- Bandbreite 250kbps, keine Fehler, kein ACK-Bit, kein original Payload. Bit-Stuffing entspricht dem worst case (also werden die Frames immer die maximale Länge haben).

- Die AWS schickt alle 5ms die SOLL-Werte an die einzelnen Motoren (das entspricht sechs Nachrichten alle 5ms), die jeweils 8 Byte Daten enthalten.
- Alle 50ms melden die Motoren ihre IST-Werte an die AWS (das entspricht jeweils einer Nachricht pro Motor, also sechs Nachrichten alle 50ms), die jeweils 8 Byte Daten enthalten.
- Alle 1000ms möchte die AWS von allen anderen Knoten eine Rückmeldung über deren Zustand. Dafür schickt sie an jeden Knoten einen Remote-Frame (entspricht sieben Nachrichten alle 1000ms), auf den die anderen Knoten antworten (entspricht sieben Nachrichten alle 1000ms). Die Frames enthalten 1 Byte Daten.
- Der Generator schickt alle 1000ms einen großen Satz Daten. Er verschickt dafür sieben Nachrichten hintereinander (also sieben Nachrichten alle 1000ms), die jeweils 8 Byte Daten enthalten.

Da das Bit-Stuffing auf worst-case eingestellt ist, lässt sich die Länge der entsprechenden Nachrichten ermitteln (aufgeführt in Tabelle 3).

Nachrichtenbezeichnung	Länge des Frames	Anzahl/Periode
AWS-SOLL-Werte	135	6/5ms
Motoren-IST-Werte	135	6/50ms
AWS-Ping	65	14/1000ms
Generator-Daten	135	7/1000ms

Tabelle 3: Übersicht der Längen der Frames im Philips-Modell

Um die Auslastung des Busses zu errechnen, wird der Zeitraum zwischen t = 1.0 und t = 2.0 betrachtet, da sich alle Nachrichten mindestens einmal im Zyklus von 1000ms wiederholen, sind hier alle Nachrichten berücksichtigt. Nachfolgend alle verschickten Bits innerhalb dieses Zeitraums:

$$AWS_SOLL_Werte_{Zeit} = 135\ Bit\ \cdot 6\ \cdot\ \frac{1s}{0{,}005s} = 162000\ Bit$$

$$Motoren_IST_Werte_{Zeit} = 135\ Bit\ \cdot 6\ \cdot\ \frac{1s}{0{,}05s} = 16200\ Bit$$

$$AWS_Ping_{Zeit} = 65\ Bit\ \cdot 14 = 910\ Bit$$

$$Generator_Daten_{Zeit} = 135\ Bit\ \cdot 7 = 945\ Bit$$

Diese Anzahl an Bits muss innerhalb einer Sekunde verschickt werden. Da die Bandbreite 250kbps beträgt, können in der Sekunde 250000 Bit übertragen werden.
Für die Busauslastung muss also nur die Summe der beanspruchten Bits in Relation zu der gesamten Bandbreite gesetzt werden:

$$Busauslastung = \frac{162000+16200+910+945}{250000} = \frac{180055}{250000} = 0{,}72022 \triangleq 72{,}022\%$$

Neben den allgemeinen Statistiken des Busses (Anzahl der Nachrichten, Anzahl der Fehler, Buslast) gibt es auch Angaben zu den einzelnen Knoten. Diese betreffen die Sendezeit von empfangenen Nachrichten. Nach dem Durchlauf einer gewissen Anzahl von Zyklen erhält man aussagekräftige Ergebnisse. Abb. 28 zeigt einen Teil der Ergebnisse.

```
scalar Philips_CAN.bus  #Simulated_Time    718.701748
scalar Philips_CAN.bus  %Busload    72.035958369472
scalar Philips_CAN.bus  #Data-Frames    963018
scalar Philips_CAN.bus  #Errors     0
scalar Philips_CAN.bus  %Errors     0
scalar Philips_CAN.aws  #sent   866716
scalar Philips_CAN.aws  #interesting_received_messages  96302
scalar Philips_CAN.aws  15_min(in_ms)   0.125
scalar Philips_CAN.aws  15_jitter(ms)   3.346
scalar Philips_CAN.aws  15_max(in_ms)   3.471
scalar Philips_CAN.aws  15_avg(in_ms)   0.399665391285
scalar Philips_CAN.aws  16_min(in_ms)   0.26
scalar Philips_CAN.aws  16_jitter(ms)   3.375
scalar Philips_CAN.aws  16_max(in_ms)   3.635
scalar Philips_CAN.aws  16_avg(in_ms)   0.625263304328
scalar Philips_CAN.aws  17_min(in_ms)   0.395
scalar Philips_CAN.aws  17_jitter(ms)   3.375
scalar Philips_CAN.aws  17_max(in_ms)   3.77
scalar Philips_CAN.aws  17_avg(in_ms)   0.850692173894
scalar Philips_CAN.aws  18_min(in_ms)   0.53
scalar Philips_CAN.aws  18_jitter(ms)   4.156
scalar Philips_CAN.aws  18_max(in_ms)   4.686
scalar Philips_CAN.aws  18_avg(in_ms)   1.082882782589
scalar Philips_CAN.aws  19_min(in_ms)   1.475
scalar Philips_CAN.aws  19_jitter(ms)   3.375
scalar Philips_CAN.aws  19_max(in_ms)   4.85
scalar Philips_CAN.aws  19_avg(in_ms)   1.829303652154
scalar Philips_CAN.aws  20_min(in_ms)   1.61
scalar Philips_CAN.aws  20_jitter(ms)   3.375
scalar Philips_CAN.aws  20_max(in_ms)   4.985
scalar Philips_CAN.aws  20_avg(in_ms)   2.053999999981
```

Abbildung 28: Auszug der Ergebnisse der Simulation für das Philips-Modell

Mit 72,036% ist die Angabe über die Auslastung relativ genau und ist daher für Analyse-Zwecke brauchbar. Die Angaben zu den Sendezeiten der Nachrichten sind ebenso nachvollziehbar und der hohe Jitter durch zwischenzeitliche „Staus“ bei der Versendung der Nachrichten erklärbar. Insgesamt kann die Umsetzung eine relativ hohe Genauigkeit

und eine korrekte zeitliche Abfolge vorweisen. Durch die Angaben bei den Nachrichten lassen sich Schlüsse ziehen, wie die zeitliche Planung optimiert werden könnte. So müssten die Nachrichten, die hier noch gleichzeitig verschickt werden, etwas flexibler geplant werden. Ein zeitversetzter Start könnte die Verzögerung von manchen Nachrichten aber nicht verhindern, eine solche Konfiguration ist zudem in der Simulation nicht möglich.

7.2 Effizienz-Tests

Die Effizienz der Simulation unterscheidet sich je nach Konfiguration. Wenn ein Fehler beim Senden oder Empfangen zur Folge hat, dass andere Nachrichten nur mit größerer Verzögerung gesendet werden können und der Nachrichtenaufwand durch Fehler und ACK-Bit ansteigt, arbeitet die Simulation nicht mehr mit dem gleichen Durchsatz. Anhand des in 7.1 gezeigten Modells von Philips werden in diesem Abschnitt unterschiedliche Konfigurationen auf ihre Effizienz getestet. Alle Tests liefen auf einem System, Unterschiede durch verschiedene Konfigurationen werden also ausgeschlossen.

Für alle Tests werden bewirkte Änderungen der Paramater durch vorige Tests beibehalten.

Folgende Parameter werden auf ihre Effizienz untersucht:

- ACK-Bit
- Fehler
- Payload

7.2.1 Bestmögliche Effizienz

Um einen Ausgangspunkt für Vergleiche in der Effizienz zu bilden, wird als erstes der Durchsatz für die effizienteste Konfiguration ermittelt. Diese deaktiviert Fehler, ACK-Bits und Payload. Als Einheit für die Effizienz wird der durchschnittliche Wert von Simsec/sec (Simulations-Sekunden pro echter Sekunde) benutzt. Um einen gleichen Rahmen für alle Tests zu schaffen, wird die Simulation jeweils bis zur Simulationszeit t = 1000 im Expressmodus betrieben und anhand des maximalen und minimalen Wertes für Simsec/sec ein Durchschnitt gebildet.

Run #0: Philips_CAN	Event #21877504	T=736.449516
Msgs scheduled: 29	Msgs created: 30772025	
Ev/sec: 417959	Simsec/sec: 14.0593	

Abbildung 29: Minimaler Durchsatz bei der besten Effizienz

Run #0: Philips_CAN	Event #20108032	T=676.89468
Msgs scheduled: 27	Msgs created: 28283146	
Ev/sec: 454835	Simsec/sec: 15.3265	

Abbildung 30: Maximaler Durchsatz bei der besten Effizienz

Nach Abb. 29 und 30 beträgt der Durchschnitt für die bestmögliche Effizienz:

$$Durchsatz_{bestmöglich} \frac{14.0593 + 15.3265}{2} = 14.6929 Simsec/sec$$

7.2.2 Test 1: ACK-Bit

Dieser Testablauf untersucht den Unterschied von Aktivierung und Deaktivierung des ACK-Bits. Erwartungsgemäß sollte allein der zusätzliche Nachrichtenaufwand, der durch ein ACK-Bit nach Versendung einer Nachricht verursacht wird, zu sichtbaren Einbrüchen der Effizienz führen. Es wird wieder ein Durchschnitt bis zur Simulationszeit t = 1000 gebildet.

Run #0: Philips_CAN	Event #17572352	T=434.509284
Msgs scheduled: 35	Msgs created: 18738548	
Ev/sec: 444396	Simsec/sec: 10.9956	

Abbildung 31: Minimaler Durchsatz bei aktiviertem ACK-Bit

Run #0: Philips_CAN	Event #33384448	T=825.484504
Msgs scheduled: 28	Msgs created: 35599983	
Ev/sec: 466721	Simsec/sec: 11.5402	

Abbildung 32: Maximaler Durchsatz bei aktiviertem ACK-Bit

Nach Abb. 31 und 32 beträgt der Durchschnitt für die Effizienz bei aktiviertem ACK-Bit:

$$Durchsatz_{ACK_aktiviert} \frac{10.9956 + 11.5402}{2} = 11.2679 Simsec/sec$$

Die Effizienz ist damit durch Aktivierung des ACK-Bits um 23,3% gesunken.

7.2.3 Test 2: Payload

Die Aktivierung von Payload hat im Programm zur Folge, dass die Knoten sich genauer mit den ankommenden Nachrichten beschäftigen und bei der Initialisierung zusätzlicher Aufwand entsteht. Der Nachrichtenaufwand wird dadurch allerdings nicht steigen, deshalb ist ein merklicher Einbruch der Effizienz nicht zu erwarten.

Run #0: Philips_CAN	Event #32581376	T=805.63048
Msgs scheduled: 34	Msgs created: 34743632	
Ev/sec: 434877	Simsec/sec: 10.7565	

Abbildung 33: Minimaler Durchsatz bei aktiviertem Payload

Run #0: Philips_CAN	Event #35780608	T=884.73314
Msgs scheduled: 35	Msgs created: 38155168	
Ev/sec: 457817	Simsec/sec: 11.3296	

Abbildung 34: Maximaler Durchsatz bei aktiviertem Payload

Durchschnittlicher Durchsatz nach Abb. 34 und 35:

$$Durchsatz_{Payload_aktiviert} \frac{10.7565 + 11.3296}{2} = 11.04305 Simsec/sec$$

Damit sinkt der Durchsatz durch aktivierten Payload nur um 2,05%.

7.2.4 Test 3: Fehler

Die Aktivierung von Fehlern kann einen starken Einbruch im Durchsatz zur Folge haben. Eine Fehlerquote in der Simulation verursacht ein gesteigertes Nachrichtenaufkommen. Wenn beispielsweise zu 50% ein Fehler beim Senden oder Empfangen der Nachricht auftritt, so hat dies folgenden zusätzlichen Aufwand zur Folge:

$$Anzahl_{zusätzlNachr.} = \lim_{n\to\infty} \sum_{i=0}^{n} 50\% \cdot (0.5^{i}) = 100\%$$

Durch einen Fehler wird über den Bus ein Error-Frame an alle Knoten geschickt, also steigt der Nachrichtenaufwand noch weiter (obigem Beispiel entsprechend):

$$Anzahl_{FehlerNachr.} = 0.5 \cdot (Anzahl_{zusätzlNachr.} + Anzahl_{Nachrichten}) = 100\%$$

Durch eine Fehlerquote von 50% entstehen also 200% mehr Nachrichten auf dem Bus. Eine solch drastische Erhöhung hätte einen starken Einbruch der Effizienz zur Folge. Die allgemeinen Formeln zu der Nachrichtenanzahl sind wie folgt:

$$Anzahl_{zusätzlNachr.} = \lim_{n\to\infty} \sum_{i=0}^{n} x\% \cdot (\frac{x}{100}^{i})$$

$$Anzahl_{FehlerNachr.} = \frac{x}{100} \cdot (Anzahl_{zusätzlNachr.} + Anzahl_{Nachrichten})$$

Für das Philips Modell ist ein Fehleraufkommen von 50% nicht realisierbar, da dies die Auslastung des Busses über 100% steigern würde. Deshalb wird in einigen Schritten die Fehlerhäufigkeit leicht erhöht.

Als erster Schritt werden zwei Knoten eine Fehleranfälligkeit von 5% erhalten. Die kombinierte totale Wahrscheinlichkeit ergibt sich aus den Möglichkeiten, bei denen ein Fehler auftritt:

Fall 1 (beide Knoten entdecken einen Fehler):

$$P_{FehlerBeide} = 0.05 \cdot 0.05 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 = 0.0025 = 0.25\%$$

Fall 2 (ein Knoten entdeckt einen Fehler, gilt doppelt):

$$P_{FehlerEiner} = 0.05 \cdot 0.95 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 = 0.0475 = 4.75\%$$

Zusammen addiert ergibt sich die Gesamtwahrscheinlichkeit:

$$P_{Fehler0,05+0,05} = 4.75\% \cdot 2 + 0.25\% = \mathbf{9.75}\%$$

Steigerung des Nachrichtenaufkommens:

$$Anzahl_{zusätzlNachr.} = \lim_{n\to\infty} \sum_{i=0}^{n} 9.75\% \cdot (0.0975^{i}) \approx 10.805\%$$

$$Anzahl_{FehlerNachr.} = 0.0975 \cdot (Anzahl_{zusätzlNachr.} + Anzahl_{Nachrichten}) \approx 10.8035\%$$

Der gesamte Anstieg der Nachrichtenzahl beläuft sich also auf ca. 21,6% (vgl. Abb. 35). Diese Erhöhung wird sich in einem merkbaren Umfang auf die Effizienz niederschlagen.

```
scalar Philips_CAN.bus  #Simulated_Time     999.999972
scalar Philips_CAN.bus  %Busload    72.14357562002
scalar Philips_CAN.bus  #Data-Frames     1341948
scalar Philips_CAN.bus  #Errors     0
scalar Philips_CAN.bus  %Errors     0

scalar Philips_CAN.bus  #Simulated_Time     999.99956
scalar Philips_CAN.bus  %Busload    76.255543952327
scalar Philips_CAN.bus  #Data-Frames     1485775
scalar Philips_CAN.bus  #Errors     143848
scalar Philips_CAN.bus  %Errors     9.6816812774478
```

Abbildung 35: Effizienztests – Statistiken ohne/mit Fehler

Nun die Effizienz-Werte:

Run #0: Philips_CAN	Event #31427584	T=702.495264
Msgs scheduled: 35	Msgs created: 34219674	
Ev/sec: 400155	Simsec/sec: 8.95077	

Abbildung 36: Minimaler Durchsatz bei aktivierten Fehlern (1)

Run #0: Philips_CAN	Event #26828800	T=599.652336
Msgs scheduled: 34	Msgs created: 29212830	
Ev/sec: 425010	Simsec/sec: 9.50925	

Abbildung 37: Maximaler Durchsatz bei aktivierten Fehlern (1)

Der durchschnittliche Durchsatz nach Abb. 36 und 37:

$$Durchsatz_{Fehler_aktiviert(1)} \frac{8.95077 + 9.50925}{2} = 9.23001 Simsec/sec$$

Damit hat eine Gesamtfehlerquote von 9,75% bereits einen Rückgang der Effizienz um 16,42% zur Folge. Im nächsten Schritt wird ein zusätzlicher Knoten eine Fehlerwahrscheinlichkeit von 5% aufweisen und die bisherigen fehlerhaften Knoten steigern ihr Fehleraufkommen auf 10%. Damit ergibt sich folgende Gesamtwahrscheinlichkeit für Fehler (umgekehrte Rechnung):

$$P_{keineFehler} = 0.95 \cdot 0.9 \cdot 0.9 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 = 0.7695 \triangleq 76.95\%$$

$$P_{Fehlergesamt} = 100\% - P_{keineFehler} = \mathbf{23.05}\%$$

Bei den Effizienz-Tests ergeben sich folgende Werte:

Run #0: Philips_CAN	Event #50405376	T=963.440876
Msgs scheduled: 32	Msgs created: 56431705	
Ev/sec: 411242	Simsec/sec: 7.85242	

Abbildung 38: Minimaler Durchsatz bei aktivierten Fehlern (2)

Run #0: Philips_CAN	Event #46158848	T=882.51892
Msgs scheduled: 32	Msgs created: 51675193	
Ev/sec: 430673	Simsec/sec: 8.29695	

Abbildung 39: Maximaler Durchsatz bei aktivierten Fehlern (2)

Der durchschnittliche Durchsatz nach Abb. 38 und 39:

$$Durchsatz_{Fehler_aktiviert(2)} \frac{7.85242 + 8.29695}{2} = 8.074685 Simsec/sec$$

Die Effizienz sinkt also um weitere 13,52%. Im letzten Schritt werden zwei zusätzliche Knoten eine Fehlerwahrscheinlichkeit von 10% erhalten. Daraus ergibt sich

$$P_{keineFehler} = 0.95 \cdot 0.9 \cdot 0.9 \cdot 0.9 \cdot 0.9 \cdot 1 \cdot 1 \cdot 1 = 0.623295 \triangleq 62.3295\%$$

$$P_{Fehlergesamt} = 100 - P_{keineFehler} = \mathbf{37.6705}\%$$

Die Auslastung des Busses liegt inzwischen bei ca. 92,5%, deshalb ist dies die letzte Steigerung. Die Effizienz-Tests ergeben folgende Werte:

Run #0: Philips_CAN	Event #37917952	T=596.9399
Msgs scheduled: 35	Msgs created: 43722152	
Ev/sec: 406017	Simsec/sec: 6.34865	

Abbildung 40: Minimaler Durchsatz bei aktivierten Fehlern (3)

Run #0: Philips_CAN	Event #50476288	T=794.488376
Msgs scheduled: 62	Msgs created: 58205208	
Ev/sec: 436589	Simsec/sec: 6.88415	

Abbildung 41: Maximaler Durchsatz bei aktivierten Fehlern (3)

Der durchschnittliche Durchsatz nach Abb. 40 und 41:

$$Durchsatz_{Fehler_aktiviert(3)} \frac{6.34865 + 6.88415}{2} = 6.6164 Simsec/sec$$

Das entspricht einem weiteren Effizienzverlust von 18,06%. Durch vergleichsweise kleine Fehler der einzelnen Knoten lässt sich die Effizienz also sehr schnell stark verringern.

Im Vergleich zu der Konfiguration, in der ACK-Bit, Payload und Fehler deaktiviert sind, sinkt die Effizienz der Simulation um insgesamt ca. 55%. Demnach ist es sinnvoll, diese Parameter nur dann zu aktivieren, wenn man sie wirklich benötigt. Hohe Fehlerwahrscheinlichkeiten wirken sich am stärksten auf die Performanz aus.

8 Zusammenfassung und Ausblick

Diese Bachelorarbeit basiert auf einer Zusammenarbeit der HAW Hamburg mit Philips Healthcare (vgl. [5]). Die Röntgensysteme von Philips Healthcare werden durch Generatoren gesteuert, die per CAN-Bus mit allen angeschlossenen Komponenten kommunizieren. Durch den zyklischen Versand der CAN-Nachrichten sind die Aufstellung eines Sendeplans und damit verbundene Analysen zwar möglich, aber umständlich. Um die Analyse eines CAN-Netzwerkes zu automatisieren, stand die Realisierung einer dafür konzeptionierten Simulation im Fokus dieser Bachelorarbeit. Die Arbeitsgruppe „CoRE“ an der HAW, in dessen Umfeld diese Bachelorarbeit betreut wurde, verwendet für ihre Netzwerk-Simulationen die Entwicklungsumgebung „OMNeT++“, die deshalb auch für die Umsetzung dieser Simulation benutzt wurde.

Ziel war es, eine Simulation zu entwerfen, die allen Anforderungen an Ergebnissen und Funktionsweisen entspricht. Die Anforderungsspezifikation fasst zu diesem Zweck die Wünsche der HAW und die von Philips zusammen. Das darauffolgende Konzept baut auf die Anforderungen auf und entwickelt konkrete Ideen für deren Umsetzung. Es geht auf die Komponenten, die Funktionsweise ihrer Kommunikation mithilfe von Nachrichten, sowie die Konfigurationsmöglichkeiten des Systems ein. Eine anschließende Übersicht der Umsetzung erläutert kurz die Implementierung. Um fehlerfreie Abläufe und die korrekte Abbildung der CAN-Spezifikation auf die Simulation zu gewährleisten, wird in der Qualitätssicherung und Evaluierung das System grundlegend überprüft.

Die automatische Analyse eines CAN-Netzwerkes ist mit dieser Simulation nun problemlos möglich. Die Analyse zeigt neben allgemeinen Statistiken des Busses auch beliebige gewünschte Daten zu den einzelnen Nachrichtenübertragungen. Die Anforderungen, die an das System gestellt wurden, sind erfüllt. Die Qualitätssicherung belegt, dass die Funktionen der Simulation in diesen Testfällen die exakte Abbildung eines CAN-Netzwerkes sind. Die Analyse-Ergebnisse sind in Bezug auf die Nachrichten-Statistiken sehr genau, weisen aber in den prozentualen Angaben für Buslast und Fehlerquote darauf hin, dass sich diese Werte nicht exakt wie in der Berechnung verhalten. Die Gründe für diese Abweichungen sind die Implementierung einer Sicherheitsmaßnahme nach der Arbitrierung, sowie der verwendete Zufallsgenerator, der prozentuale Angaben nicht deckungsgleich umsetzt.

Ein Problem bei der Implementierung waren die Grundvoraussetzungen von OMNeT++. Es arbeitet bei der Kommunikation von einzelnen Netzwerk-Teilnehmern ausschließlich mit Nachrichten, bietet also keine Möglichkeiten für globale Variablen. Das führt zum einen

dazu, dass es für manche Funktionen eines CAN-Netzwerkes keine Möglichkeit der direkten Abbildung in diese Entwicklungsumgebung gibt. Die Arbitrierung und die Fehlererkennung sind deshalb auf dieses Nachrichtenmodell angepasst worden, arbeiten aber zuverlässig. Zum anderen hat es die oben genannte Sicherheitsmaßnahme nach der Arbitrierung zur Folge, die einen zeitlich korrekten Ablauf garantiert. Die hierdurch entstandene Ungenauigkeit bei der Angabe der Buslast ist mit nachträglichen Optimierungen der Sendeabläufe auf ein Minimum reduziert worden. Zu weiteren Schwierigkeiten während der Implementierung führten Timer-Abbrüche, die nichtdeterministische Fehler produzierten. Die gewünschte Funktionalität ist allerdings mithilfe von zusätzlichen Variablen erreicht worden.

Abgesehen von diesen Problemen eignet sich OMNeT++ hervorragend für die Abbildung des CAN-Netzwerkes. Die fehlende Möglichkeit der Kommunikation ohne Nachrichtenpakete wirkt sich allerdings negativ auf die Performanz des Systems aus. Ein bitweiser Datenaustausch der Komponenten wäre demnach ein sinnvoller Schritt der Weiterentwicklung für serielle Netzwerk-Modelle wie den CAN-Bus.

Das CAN-Protokoll selber befindet sich auf dem Scheideweg. Einerseits ist es nach wie vor weit verbreitet und ist mit höheren Protokollen sehr zugänglich gestaltet, andererseits fordern moderne Systeme, speziell sogenannte „Infotainment“-Systeme, eine Bandbreite, die der CAN-Bus nicht bieten kann. Nicht zuletzt aus diesem Grund werden andere Möglichkeiten gesucht, die gesamte Kommunikation auf andere Protokolle umzustellen.

Einen Ansatz hierfür bietet die bereits erwähnte Arbeitsgruppe „CoRE“, die sich hauptsächlich mit dem Kommunikations-Modell des Echtzeit-Ethernets im Automotive-Kontext beschäftigt. In Flugzeugen kommt eine ähnliche Technik (AFDX - vgl. [27]) teilweise bereits zum Einsatz. Es gibt auch andere Feldbus-Protokolle, die eine höhere Bandbreite unterstützen, beispielsweise FlexRay (vgl. [8]) oder MOST (vgl. [10]).

Auch der Erfinder des CAN-Protokolls versucht, mit dessen Weiterentwicklung einen Schritt in diese Richtung zu tun. So stellte Bosch im August 2011 das Modell „CAN with Flexible Data-Rate“ [28] vor. Dieses sieht einen Wechsel der Bitzeiten während der Übertragung vor. Zwischen der Arbitrierungs-Phase und dem CRC-Feld werden die Bitzeiten verkürzt und schaffen somit eine höhere Bandbreite. Zusätzlich wird das Datenfeld auf bis zu 64 Byte erweitert. Im Hinblick auf die vergleichsweise sehr hohen Bandbreiten in Ethernet-Netzwerken bleibt allerdings abzuwarten, ob dieser Ansatz weiter verfolgt wird.

Eine grundsätzliche Umstellung von Kommunikations-Systemen ist immer ein langwieriger Prozess. Sollte dieser Weg allerdings von führenden Unternehmen eingeschlagen werden, ist die langfristige Ablösung des CAN-Protokolls als de-facto-Standard im Automotive-Kontext kaum zu verhindern. Es liegt somit in der Hand der großen Unternehmen der Automobil-Industrie, wie die Zukunft des CAN-Busses aussieht.

Literaturverzeichnis

[1] Robert Bosch GmbH, Bosch-Autoelektrik und -Autoelektronik : Bordnetze, Sensoren und elektronische Systeme, 6. Hrsg., K. Reif, Hrsg., Wiesbaden: Vieweg + Teubner, 2011.

[2] IEC, *Digitale Datenkommunikation in der Leittechnik. Feldbus fuer industrielle Leitsysteme. Spezifikation des Protokolls der Anwendungsschicht (Application Layer),* BSI, Hrsg., 2004.

[3] ISO, *Road vehicles -- Controller area network (CAN) -- Part 1-5,* 2003;2004;2006;2007.

[4] J. Lunze, Automatisierungstechnik: Methoden für die Überwachung und Steuerung kontinuierlicher und ereignisdiskreter Systeme, München: Oldenbourg Wissenschaftsverlag, 2012.

[5] „Philips Healthcare," [Online]. Available: http://www.healthcare.philips.com/de_de/. [Zugriff am 28 März 2013].

[6] „CoRE Group," [Online]. Available: http://core.informatik.haw-hamburg.de/en/. [Zugriff am 28 März 2013].

[7] „EtherCAT Technology Group," [Online]. Available: http://www.ethercat.org/. [Zugriff am 27 März 2013].

[8] M. Rausch, FlexRay - Grundlagen, Funktionsweise, Anwendung, München: Hanser-Verlag, 2008.

[9] „LIN subbus," [Online]. Available: http://www.lin-subbus.org/. [Zugriff am 27 März 2013].

[10] „MOST," [Online]. Available: http://www.mostcooperation.com/home/index.html. [Zugriff am 27 März 2013].

[11] W. Lawrenz, Controller Area Network: Grundlagen, Design, Testtechnik, Berlin: VDE Verlag, 2011.

[12] Bosch, „CAN Specification - Version 2.0," Stuttgart, 1991.

[13] „CAN in Automation (CiA): Controller Area Network (CAN)," [Online]. Available: http://www.can-cia.org/. [Zugriff am 27 März 2013].

[14] „ODVA," [Online]. Available: http://www.odva.org/Home/ODVATECHNOLOGIES/DeviceNet/DeviceNetTechnologyOverview/tabid/72/lng/en-US/Default.aspx. [Zugriff am 27 März 2013].

[15] „CAN in Automation (CiA): Time-triggered CAN," [Online]. Available: http://www.can-

cia.org/index.php?id=166. [Zugriff am 27 März 2013].

[16] K. Etschberger, Controller-Area-Network: Grundlagen, Bausteine, Anwendungen, München/Wien: Hanser, 2002.

[17] M. di Natale, H. Zeng, P. Giusto und A. Ghosal, Understanding and Using the Controller Area Network Communication Protocol - Theory and Practice, New York/Dordrecht/Heidelberg/London: Springer, 2012.

[18] „OMNeT++ Network Simulation Framework," [Online]. Available: http://www.omnetpp.org. [Zugriff am 26 März 2013].

[19] T. Faison, Event-Based Programming: Taking Events to the Limit, Berkeley: Apress, 2006.

[20] A. S. Tanenbaum und D. J. Wetherall, Computer networks, Boston, Mass.: Pearson, 2011.

[21] L. Cheng und Y. Cai, „An interactive simulation framework for satellite design," in *Software Engineering and Science (ICSESS), 2012 IEEE 3rd International Conference*, Peking, 2012.

[22] W. Prodanov, M. Valle und R. Buzas, „A controller Area Network Bus Transceiver Behavioral Model for Network Design and Simulation," in *Industrial Electronics, IEEE Transactions, Volume 56, Issue: 9*, 2009.

[23] X. Hong und L. Chuan-Gou, „Modeling and simulation analysis of CAN-bus on bus body," in *Computer Application and System Modeling (ICCASM), 2010 International Conference, Volume 12, S. V12-205 – V12-208*, 2010.

[24] J. Hao, J. Wu und C. Guo, „Modeling and simulation of CAN network based on OPNET," in *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference, S. 577-581*, 2011.

[25] F. Li, L. Wang und C. Liao, „CAN(Controller Area Network) Bus Communication System Based on Matlab/Simulink," in *Wireless Communications, Networking and Mobile Computing, 2008. WiCOM '08. 4th International Conference, S. 1-4*, 2008.

[26] L. Pinho, F. Vasques und E. Tovar, „Integrating inaccesssibility in response time analysis of CAN networks," in *Factory Communication Systems, 2000. Proceedings. 2000 IEEE International Workshop, S. 77-84*, 2000.

[27] SYSGO AG, April 2013. [Online]. Available: http://www.sysgo.com/products/safety-critical-ethernetafdx/. [Zugriff am 28 April 2013].

[28] Robert Bosch GmbH, August 2011. [Online]. Available: http://www.bosch-semiconductors.de/media/pdf_1/canliteratur/can_fd.pdf. [Zugriff am 28 April 2013].

Abbildungsverzeichnis

Tabelle 1: Arbitrierungstest – Zeitpunkte für Sendewünsche von Knoten 48
Tabelle 2: Arbitrierungstest – Erwarteter zeitlicher Ablauf der Nachrichten auf dem Bus ... 49
Tabelle 3: Übersicht der Längen der Frames im Philips-Modell 57

Abbildung 1: Arbitrierung mit 3 Knoten 11
Abbildung 2: Data-Frame (CAN2.0A) 12
Abbildung 3: Bit-Stuffing Sequenz 14
Abbildung 4: Übertragung einer Nachricht in Beispielnetz 15
Abbildung 5: Beispielnetzwerk mit 4 angeschlossenen Knoten 24
Abbildung 6: Arbitrierungsansatz 1 – Knoten fragt den Bus nach Zustand 27
Abbildung 7: Arbitrierungsansatz 2 – Der Bus fragt die Knoten nach Sendewünschen 28
Abbildung 8: Arbitrierungsansatz 3 – Beispielhafter Ringverbund mit 2 Knoten 29
Abbildung 9: Arbitrierungsansatz 4 – Der Bus schickt seinen Zustand an die Knoten 30
Abbildung 10: Arbitrierungsansatz 5 – Knoten schickt seinen Sendewunsch an den Bus 31
Abbildung 11: Versendung einer Nachricht ohne ACK und ohne Fehler 34
Abbildung 12: Versendung einer Nachricht mit ACK und ohne Fehler 35
Abbildung 13: Nachrichtenübertragung mit Fehler beim Sender 38
Abbildung 14: Acknowledgement-Error bei Nachrichtenübertragung 39
Abbildung 15: Fehler bei der Nachrichtenübertragung beim Empfänger 40
Abbildung 16: Klassendiagramm 46
Abbildung 17: Arbitrierungstest – Tatsächlicher zeitlicher Ablauf auf dem Bus 50
Abbildung 18: Zeitverhaltenstest – OMNeT++ Events 51
Abbildung 19: Payload-Test – OMNeT++ Events, 5. Nachricht nicht verarbeitet 52
Abbildung 20: Fehlertest – Fehlerhäufigkeit bei Empfangsfehlern - Kurze Zeit 53
Abbildung 21: Fehlertest – Fehlerhäufigkeit bei Empfangsfehlern - Lange Zeit 53
Abbildung 22: Fehlertest – Erkannter Fehler bei Empfänger 53
Abbildung 23: Fehlertest – Fehlerhäufigkeit bei Sendefehlern 54
Abbildung 24: Fehlertest – Erkannter Fehler bei Sender 54
Abbildung 25: Laufzeitfehler der allgemeinen Konfiguration 55
Abbildung 26: Laufzeitfehler bei Falscheingabe in Stimuli-Generator 55
Abbildung 27: Philips-Modell 56
Abbildung 28: Auszug der Ergebnisse der Simulation für das Philips-Modell 58
Abbildung 29: Minimaler Durchsatz bei der besten Effizienz 59
Abbildung 30: Maximaler Durchsatz bei der besten Effizienz 59
Abbildung 31: Minimaler Durchsatz bei aktiviertem ACK-Bit 60

Abbildung 32: Maximaler Durchsatz bei aktiviertem ACK-Bit 60
Abbildung 33: Minimaler Durchsatz bei aktiviertem Payload 60
Abbildung 34: Maximaler Durchsatz bei aktiviertem Payload... 60
Abbildung 35: Effizienztests – Statistiken ohne/mit Fehler.. 62
Abbildung 36: Minimaler Durchsatz bei aktivierten Fehlern (1)... 62
Abbildung 37: Maximaler Durchsatz bei aktivierten Fehlern (1) 62
Abbildung 38: Minimaler Durchsatz bei aktivierten Fehlern (2)... 63
Abbildung 39: Maximaler Durchsatz bei aktivierten Fehlern (2) 63
Abbildung 40: Minimaler Durchsatz bei aktivierten Fehlern (3)... 64
Abbildung 41: Maximaler Durchsatz bei aktivierten Fehlern (3) 64

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den ________________ ____________________________