Smart Mechatronics

Hochschule Bochum
Bochum University
of Applied Sciences

Masterthesis

zur Erlangung des akademischen Grades
Master of Science (M.Sc.)

Modellbasierte Entwicklung und Verifikation einer modularen Antriebsplattform mit MATLAB/Simulink

Autor: Florian Wagner
florian0.wagner@hs-bochum.de
Matrikelnummer: 010205535

Erstgutachter: Prof. Dr.-Ing. Arno Bergmann
Zweitgutachter: M.Sc. Kevin Leiffels

Abgabedatum: 16.02.2017

Eidesstattliche Erklärung

Eidesstattliche Erklärung zur Abschlussarbeit:
»Modellbasierte Entwicklung und Verifikation einer modularen Antriebsplattform mit MATLAB/Simulink«

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

$Unterschrift:$ $Ort, Datum:$

Inhaltsverzeichnis

Inhaltsverzeichnis iv

Abkürzungsverzeichnis v

Symbolverzeichnis vii

1 Einleitung 1
- 1.1 Aufgabenstellung und Motivation 1
- 1.2 Gliederung der Arbeit 2

2 Grundlagen Software-Test 3
- 2.1 Teststufen 3
  - 2.1.1 Komponententest 3
  - 2.1.2 Integrationstest 4
  - 2.1.3 Systemtest 4
- 2.2 Testverfahren 4
  - 2.2.1 Statische Testverfahren 5
  - 2.2.2 Dynamische Testverfahren 6
- 2.3 Fundamentaler Testprozess 12
- 2.4 Modellbasierter Test 13
  - 2.4.1 Modellbasierter Testprozess 14
  - 2.4.2 Teststrategie für den Modellbasierten Test 17

3 Anforderungserhebung 20
- 3.1 Umfeldmodellierung 21
- 3.2 Anwendungsszenarien 22
- 3.3 Systemanforderungen 23

4 Systementwurf 24
4.1 Wirkstruktur . . . 24
4.2 Designspezifikation . . . 28
4.3 Mikrocontroller . . . 29
4.4 Software-Toolkette . . . 31

5 Modellierung mit Simulink 32
5.1 Modellstruktur . . . 32
5.2 Schnittstellen . . . 34
5.3 Funktionsmodellierung . . . 35
5.3.1 Regelungskonzept . . . 36
5.3.2 Betriebsmodi . . . 37
5.3.3 Geschwindigkeitsmodus . . . 39
5.3.4 Traktionskontrolle . . . 40
5.3.5 Bremsfunktion . . . 42
5.3.6 Überwachungsfunktion . . . 42
5.4 Streckenmodellierung . . . 43

6 Modellbasierter Test mit Simulink 44
6.1 Überblick . . . 44
6.2 Testfallbeschreibung . . . 47
6.2.1 Anforderungsnachverfolgung . . . 48
6.2.2 Modellüberdeckungsanalyse . . . 49
6.2.3 Blackbox-Test . . . 52
6.2.4 Whitebox-Test . . . 54
6.2.5 Back-to-Back-Test . . . 56
6.3 Testdatenerstellung . . . 58
6.4 Testrahmenerstellung . . . 60
6.5 Testdurchführung . . . 63
6.6 Testauswertung und -dokumentation . . . 65
6.7 Anforderungen und Beschränkung der Toolkette . . . 68
6.7.1 Beschränkungen für die Modellstruktur . . . 68
6.7.2 Limitierungen und Fehler im modellbasierten Testprozess . . . 69

7 Verifikation **73**
7.1 Verifikation Motorcontroller-Funktionsmodell 73
7.1.1 Komponententest 74
7.1.2 Integrationstest 81
7.2 Systemverifikation Motorcontroller EpOS 83
7.2.1 Systemintegrationstest 83
7.2.2 Abnahmetest und Validierung 88

8 Beurteilung der Toolkette **89**

9 Fazit **91**

Abbildungsverzeichnis **II**

Tabellenverzeichnis **III**

Literatur **IV**

A Anhang **VI**
A.1 Simulink Project auf Daten-CD VI
A.2 Sonstiger Inhalt auf Daten-CD VII

Abkürzungsverzeichnis

ADC Analog Digital Converter

ANF Anforderung

ASR Antriebsschlupfregelung

CAN Controller Area Network

CONSENS CONceptual design Specification technique for the ENgineering of complex Systems

DMC Digital Motor Control

DSP Digitaler Signalprozessor

eCAP Enhanced Capture Module

EMV Elektromagnetische Verträglichkeit

EpOS Entwicklungsplattform Ortsfrequenzfilter-Sensor

ePWM Enhanced Pulse Width Modulator

FOC Field-Oriented Control

GPIO General Purpose Input Output

GUI Graphical User Interface

HIL Hardware-in-the-Loop

ISTQB International Software Testing Qualifications Board

MBD Model-Based Design, Modellbasierte Entwicklung

MBT Model-Based Testing, Modellbasierter Test

MCM Motorcontrollermodul

MIL Model-in-the-Loop

PIL Processor-in-the-Loop

PMSM Permanentmagneterregte Synchronmaschine

PoC Point of Control

PoO Point of Observation

SIL Software-in-the-Loop

SUT System Under Test, Testobjekt

TI Texas Instruments

Symbolverzeichnis

Symbol	Bedeutung	Einheit
$i_{\mathsf{s,q,w}}$	Führungsgröße drehmomentbildende Statorstromkomponente	A
n	Motordrehzahl	min^{-1}
n_{in}	Drehzahlwunsch vor Rampe	min^{-1}
n_{max}	Maximale Drehzahl	min^{-1}
n_{out}	Drehzahlwunsch nach Rampe	min^{-1}
n_{ref}	Referenzdrehzahl	min^{-1}
n_{remote}	Fernbedienungssignal Fahrer	1
$n_{\mathsf{remote\ max}}$	Maximales Fernbedienungssignal	1
n_{w}	Führungsgröße Motordrehzahl	min^{-1}
S	Reifenschlupf	1
ϑ	Polradwinkel	rad
U_{min}	Entladeschlussspannung	V

1 Einleitung

Gegenstand dieser Arbeit ist die modellbasierte Entwicklung und der modellbasierte Test einer modularen Antriebsplattform für Synchronmaschinen. Die vorliegende Masterarbeit wurde in Kooperation mit der Firma Smart Mechatronics GmbH am Institut für Systemtechnik der Hochschule Bochum durchgeführt. In einem im Vorfeld abgeschlossenen Projekt konnte eine modellbasiert entwickelte Regelungs- und Steuerungssoftware auf einem Mikrocontroller von Texas Instruments sowie entsprechender Leistungselektronik auf einem Longboard in Betrieb genommen werden [1, 2].

1.1 Aufgabenstellung und Motivation

In diesem Projekt wird der modellbasierte Test als Teil des gesamten modellbasierten Entwicklungsprozesses am Institut für Systemtechnik untersucht. Hierfür wird der Testprozess der von Simulink angebotenen Toolkette evaluiert und an einer neu zu entwickelnden E-Skateboard-Version durchgeführt. Dabei wird die vorhandenen Regelungs- und Steuerungssoftware funktional weiterentwickelt.

Modellbasierte Entwicklung bietet im Vergleich zu handgeschriebenem Code den Vorteil, dass der entwickelte Algorithmus der Systemsoftware bereits im Modell simuliert, analysiert und verifiziert werden kann, bevor dieser auf einer Hardware in Betrieb genommen wird [3].

Innerhalb der abgeschlossenen Projektarbeit von Kevin Leiffels konnten diese Vorteile nur teilweise genutzt werden, da das entwickelte Simulink-Modell über eine direkte Anbindung an hardwarespezifische Blöcke einer Texas Instruments Bibliothek für die Generierung von hardwarespezifischem Code verfügte. Durch diese Hardwareschnittstellen innerhalb des Modells und durch das Fehlen eines Streckenmodells war es nicht möglich, das Systemmodell zu simulieren sowie die Regelung durch closed-loop Simulationen mit einem Streckenmodell bereits modellbasiert zu testen und zu verifizieren.[1]

Aus diesem Grund wird eine Modellstruktur entwickelt, die sowohl eine direkte Hardwareanbindung zur automatischen Codegenerierung bereitstellt, als auch die Möglichkeit zur Anbindung eines Streckenmodells für Systemsimulationen bietet. Des Weiteren wird die Modellarchitektur so umstrukturiert, dass eine Modellverifikation mit entsprechenden Simulink-Tools durchgeführt werden kann.

Neben funktionalen Weiterentwicklungen wie der Erweiterung des Antriebssystems auf ein Vierradantrieb mit Einzelradansteuerung und Traktionskontrolle wird eine modulare Antriebsplattform angestrebt. Dadurch ist es möglich, die gekapselte modellbasiert entwickelte Regelungs- und Steuerungssoftware für andere Projekten zu adaptieren und in unterschiedliche Systeme wie bspw. dem SolarCar der Hochschule Bochum [4] zu integrieren.

Das zu entwickelnde Implementierungsmodell wird durch Anwendung des modellbasierten Testprozesses auf Modellebene verifiziert, bevor es mit einer parallel entwickelten Leistungselektronik (siehe [5]) zu einem Wechselrichter eingebettet und am Beispiel eines elektrischen Mountainboards in Betrieb genommen wird. Die Leistungsfähigkeit des erprobten modellbasierten Testprozesses wird im Hinblick auf weitere Projekte des Instituts für Systemtechnik sowie der Smart Mechatronics GmbH beurteilt.

1.2 Gliederung der Arbeit

Diese Arbeit ist in folgende Kapitel gegliedert:

1. Grundlagen Software-Test
2. Anforderungserhebung
3. Systementwurf
4. Modellierung
5. Modellbasierter Test
6. Verifikation
7. Beurteilung der Toolkette
8. Fazit

2 Grundlagen Software-Test

In diesem Kapitel werden die für die Umsetzung nötigen Grundlagen aus dem Bereich Softwaretest dargestellt und in Zusammenhang zum modellbasierten Test gebracht.

2.1 Teststufen

Software-Tests werden in folgende Teststufen klassifiziert:

- Komponententest
- Integrationstest
- Systemtest

Die Einstufung dieser Teststufen ist nicht nur eine zeitliche Unterteilung der Testaktivitäten. Vor allem werden in den technisch unterschiedlichen Teststufen verschiedene Ziele verfolgt, die nur durch Verwendung unterschiedlicher Testverfahren und Testwerkzeuge erreicht werden können. [6]

2.1.1 Komponententest

Im Komponententest werden Algorithmen einzelner Softwarebausteine, welche funktionale Einheiten (Module) darstellen, erstmalig systematisch getestet [7]. Als Testbasis für die Erstellung von Testfällen mittels funktionsorientierten Testverfahren werden die komponentenspezifischen Anforderungen herangezogen. Testfälle nach strukturorientierten Testverfahren werden durch Analayse der Programmstruktur der jeweiligen Komponente erstellt [6].

Auf Modulebene ist der Einsatz sogenannter Blackbox- und Whitebox-Testverfahren möglich. Die Erläuterung der Testverfahren erfolgt in Abschnitt 2.2.

Da die Komplexität eines Algorithmus auf Modulebene geringer ist als auf Systemebene und über klare Schnittstellen angesprochen wird, kann ein Modul mit relativ wenigen Testfällen umfassend getestet werden. Dies gilt als Voraussetzung für die anschließenden Integrationstests, um die Testfälle dort auf das integrierte Zusammenwirken ausrichten zu können. [7]

Durch die Isolierung des Moduls von anderen Softwarebausteinen des Systems können komponentenexterne Einflüsse beim Test ausgeschlossen werden, wodurch Fehlerwirkungen dem getesteten Modul zugeordnet werden können [6].

2.1.2 Integrationstest

Beim Integrationstest wird das Zusammenspiel zweier oder mehrerer Module geprüft. Dadurch wird ermittelt, ob die Schnittstellen der Module korrekt spezifiziert und implementiert sind. Je nach Strategie erfolgt die Integration top-down, also von den aufrufenden zu den aufgerufenen Komponenten oder umgekehrt (bottom-up). Für den Integrationstest werden vorwiegend Blackbox-Verfahren eingesetzt. [8]

2.1.3 Systemtest

Der Systemtest stellt die Funktionsfähigkeit des gesamten Systems entsprechend der Anforderungsspezifikation sicher. Daher handelt es sich beim Systemtest um ein rein funktionales Blackbox-Verfahren. Durch das Erstellen von Testfällen für den Systemtest schon während der Anforderungserhebung können Anforderungen an ein System präziser spezifiziert werden. [8]

Oft wird durch einen sogenannten Systemintegrationstest das Zusammenspiel des zu testenden Systems mit Systemen in seiner Umgebung geprüft [8]. Ein Beispiel dafür ist die Einbettung des zu testenden Softwaresystems mit entsprechender Hardware und Mechanik zu einem mechatronischen Gesamtsystem.

2.2 Testverfahren

Es gibt verschiedene Ansätze zur Klassifizierung der unterschiedlichen Software-Testverfahren. In dieser Arbeit werden die Testverfahren nach ihrer Testreferenz klassifiziert. Diese Klassifizierung orientiert sich an aktueller Primärliteratur [7, 6, 9, 10].

Dabei erfolgt eine Beschränkung auf die Testverfahren mit der höchsten Praxisrelevanz, die innerhalb dieser Arbeit zum Einsatz kommen.

Abbildung 2.1 zeigt die Klassifizierung der Testverfahren. Diese werden zunächst in statische und dynamische Techniken unterteilt. Die weitere Unterteilung erfolgt anhand der bereits genannten Testreferenz des jeweiligen Testverfahrens. Die Testreferenz ist unterhalb der Testverfahren vermerkt.

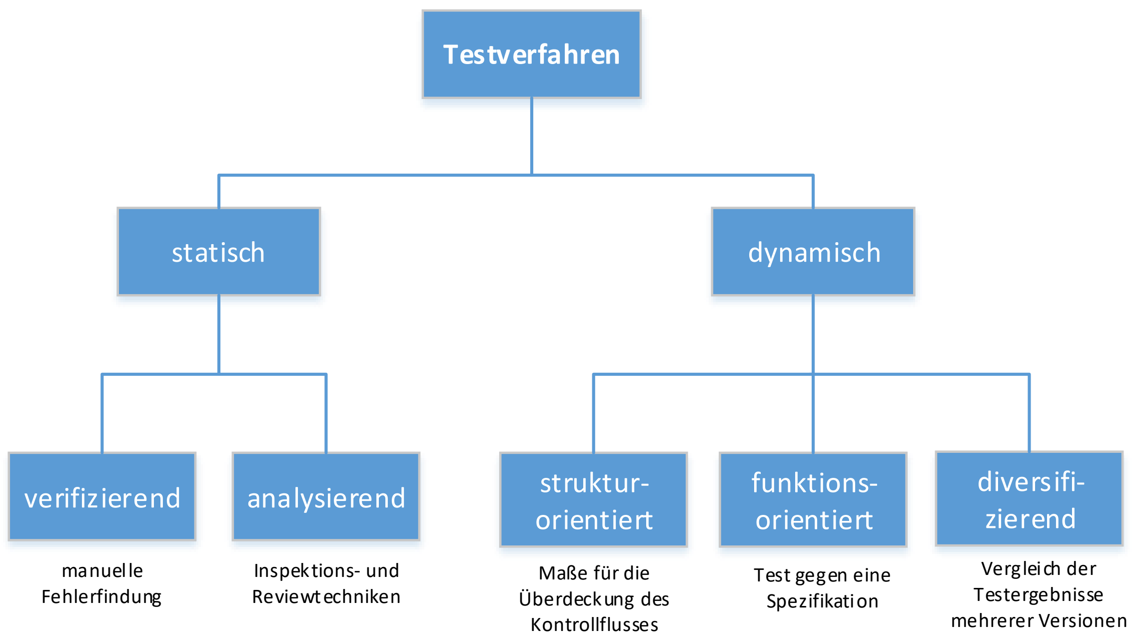


Abbildung 2.1: Klassifizierung von Software-Testverfahren

2.2.1 Statische Testverfahren

Bei statischen Testverfahren wird die zu testende Software im Gegensatz zu dynamischen Testverfahren nicht ausgeführt. Prinzipiell ist dafür keine Computerunterstützung notwendig, jedoch können statische Testverfahren sowohl von Personen, als auch von bestimmten Werkzeugen durchgeführt werden. [6]

Computergestütze statische Testverfahren werden nicht durch die zu evaluierende Simulink-Toolkette abgedeckt und sind daher nicht Inhalt dieser Arbeit.

2.2.2 Dynamische Testverfahren

In diesem Abschnitt werden Testtechniken eingeführt, die während der Evaluierung und Modellverifikation eingesetzt werden.

Bei dynamischen Testverfahren wird das Testobjekt auf einem Computer mit Eingabedaten versehen und zur Ausführung gebracht.

Da in den unteren Teststufen (Komponenten- und Integrationstest) ein lauffähiges Programm meist nicht vorliegt, muss das Testobjekt, wie in Abbildung 2.2 dargestellt, in einen Testrahmen eingebettet werden. Dadurch entsteht ein ablauffähiges Programm. [6]

Das Testobjekt kann über Schnittstellen mit Platzhaltern verbunden werden, die das Ein- und Ausgabeverhalten aufzurufender Programmteile simulieren. Der Testrahmen versorgt das Testobjekt mit den Testfällen entsprechenden Eingabedaten (Point of Control, PoC). Außerdem gibt es Schnittstellen zwischen Testobjekt und Testrahmen, um die Möglichkeit zur Analyse und Darstellung der Testdaten zu gewährleisten. Das Testobjekt versorgt den Testrahmen mit den Testausgaben (Point of Observation, PoO), wodurch die Ergebnisse untersucht werden können. Der Testrahmen muss vom Tester entwickelt bzw. auf die Schnittstellen des Testobjektes angepasst werden. [6]

Dynamische Testtechniken werden nach unterschiedlichen Kriterien klassifiziert. Eine Einordnungsmöglichkeit ist die Klassifizierung nach verwendeter Testreferenz bzw. Testbasis. Mithilfe der Testreferenz wird die Testvollständigkeit und die Korrektheit der Testergebnisse beurteilt. [9]
Die Testtechniken werden entsprechend eingeordnet in:

- Funktionsorientierte Tests
- Strukturorientierte Tests
- Diversifizierende Tests

Diese dynamischen Testtechniken besitzen eine hohe praktische Bedeutung. Als Minimalanforderungen an dynamische Tests werden von zahlreichen Standards funktionsorientierte Tests in allen Testphasen gefordert. Der sogenannte Zweigüberdeckungstest wird als minimale Testtechnik für den strukturorientierten Modultest angesehen. Bei

Versionsentwicklung muss auf die Wiederholbarkeit der Testfälle geachtet werden, sodass in diesem Fall ein systematischer Regressionstest aus der Klasse der diversifizierenden Tests als notwendig erachtet wird. [9]

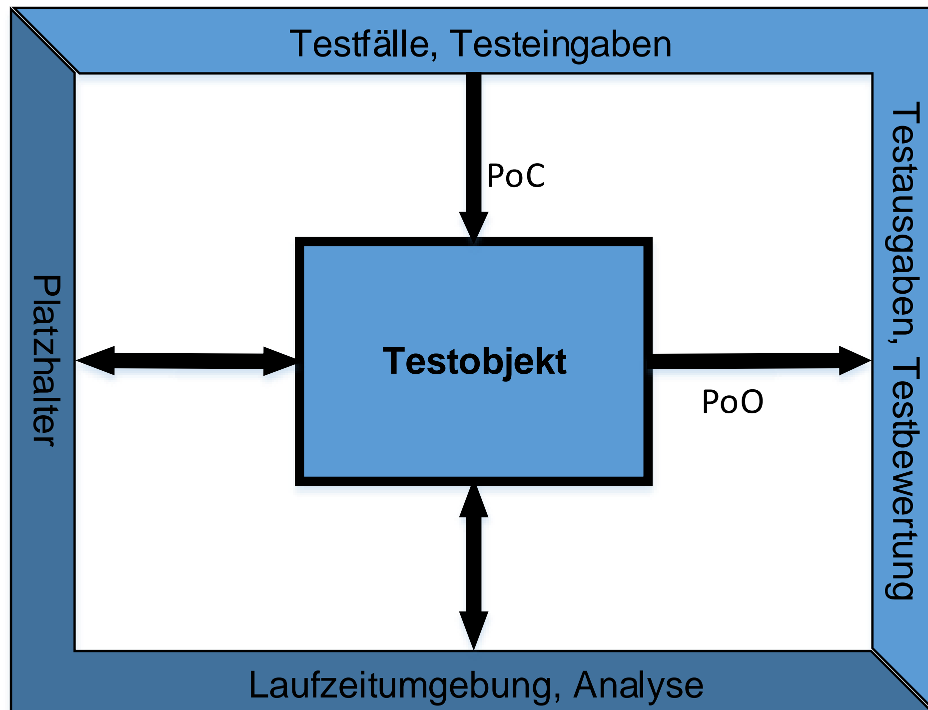


Abbildung 2.2: Testrahmen Blackbox-Verfahren

Funktionsorientierte Tests

Funktionsorientierte Testtechniken gehören zu den Blackbox-Verfahren. Sie bewerten die Testvollständigkeit und die Korrektheit der Testergebnisse anhand von Spezifikationen [9]. Als Testbasis bzw. Testreferenz dienen die funktionalen Anforderungen. Es steht die Prüfung des spezifizierten Ein- und Ausgabeverhaltens, also die Funktionalität des Testobjekts, im Mittelpunkt [6].

Das Testobjekt wird bei Blackbox-Verfahren als schwarzer Kasten angesehen. Da die Testfälle anhand der Spezifikationen ermittelt werden, sind keine Informationen über den Programmtext und den inneren Aufbau notwendig. Der PoO und der PoC liegen außerhalb des Testobjekts, wie in Abbildung 2.2 dargestellt. Das Verhalten des Testobjekts wird somit von außen beobachtet und gesteuert. Blackbox-Verfahren eignen

sich für den Einsatz in allen Teststufen. Wenn Modelle zur Spezifikation der Software oder einzelner Komponenten genutzt werden, können Testfälle systematisch aus den Modellen abgeleitet werden. [6]

Ein vollständiger Test mit allen möglichen Eingabewerten und deren Kombinationen ist aufgrund der meist hohen Anzahl an Kombinationsmöglichkeiten unrealistisch und bedingt durch begrenzte Ressourcen nicht sinnvoll. Um systematisch eine sinnvolle Auswahl an möglichen Testfällen zu finden, sind folgende Verfahren verbreitet:

- Funktionale Äquivalenzklassenbildung:
  Werte mit äquivalentem Verhalten werden einer Klasse zugeteilt.

- Grenzwertanalyse:
  Untersuchung der Grenzbereiche der zugeteilten Äquivalenzklassen.

- Ursache-Wirkungs-Analyse:
  Untersuchung von Systemen deren Ausgangsverhalten von Eingabewertkombinationen abhängt.

- Zustandstest:
  Untersuchung von Systemen mit zustandsabhängigem Ausgangsverhalten.

Beim Äquivalenzklassentest wird angestrebt, mit jedem Testfall eine ganze Klasse gleichartiger Fehler aufzudecken. Dafür müssen Wertebereiche von Ein- und Ausgaben in Klassen zerlegt werden. Alle Werte einer Klasse ergeben ein äquivalentes Verhalten des Testobjekts. [8] Das bedeutet:

- Wenn eine Wertekombination der Äquivalenzklasse einen Fehler aufdeckt, wird erwartet, dass auch jede andere Wertekombination der Äquivalenzklasse diesen Fehler aufdeckt.

- Wenn eine Wertekombination aus der Äquivalenzklasse keinen Fehler aufdeckt, wird erwartet, dass auch keine andere Wertekombination aus der Äquivalenzklasse einen Fehler aufdeckt.

Durch Ergänzung einer Grenzwertanalyse werden die Grenzen und die Umgebung der Äquivalenzklassen während des Erstellens von Testfällen beachtet, da es häufig in den Grenzbereichen zu Fehlern kommt [8].

Äquivalenzklassenbildung und die Grenzwertanalyse sind geeignet, wenn Software ohne Zustandsabhängigkeit zu testen ist (gedächtnislose Software) und für die Eingaben und Ausgaben eine Fallunterscheidung durchgeführt werden kann (unabhängige Äquivalenzklassen) [9].

Bei einer Ursache-Wirkungs-Analyse wird jede Ursache als eine Bedingung beschrieben, die aus Kombinationen von Eingabewerten besteht. Die Eingabewerte werden über logische Operatoren verknüpft. [6]
Eine Ursache-Wirkungs-Analyse ist geeignet, wenn Software ohne Zustandsabhängigkeit zu testen ist und für die Eingaben und Ausgaben keine Fallunterscheidung durchgeführt werden kann (abhängige Äquivalenzklassen) [9].

Bei einer Zustandsanalyse wird das Ausgangsverhalten mithilfe einer Zustandsmaschine beschrieben. Der Zustand eines Systems beinhaltet die Informationen, die sich aus den bisherigen Eingabewerten ergeben haben und die noch benötigt werden, um die Systemreaktion auf noch folgende Eingaben zu bestimmen. [6]
Eine Zustandsanalyse ist geeignet, wenn Software mit Zustandsabhängigkeit zu testen ist (gedächtnisbehaftete Software). Die Reaktion wird nicht nur von den Eingaben, sondern auch vom Zustand bestimmt [9].

Zusammenfassend kann durch die Erstellung funktionsorientierter Testfälle eine vollständige Abdeckung der Spezifikationen überprüft werden. Sie geben jedoch keine Garantie auf vollständige Abdeckung der Programmstruktur.

Strukturorientierte Tests

Strukturorientierte Testtechniken gehören zu den Whitebox-Verfahren. Sie bewerten die Testvollständigkeit anhand der Abdeckung von Strukturelementen des Programmcodes [9]. Auch die Struktur abstrakter Modelle der Software kann als Testbasis dienen [6].

Die Erstellung der Testfälle erfolgt aus der Programmstruktur bzw. Modellstruktur. Wie in Abbildung 2.3 dargestellt, liegt der PoO innerhalb des Testobjekts, da dessen innerer Ablauf analysiert wird. Da ein Eingriff in den Ablauf des Testobjekts möglich ist, liegt der PoC auch innerhalb des Testobjekts.

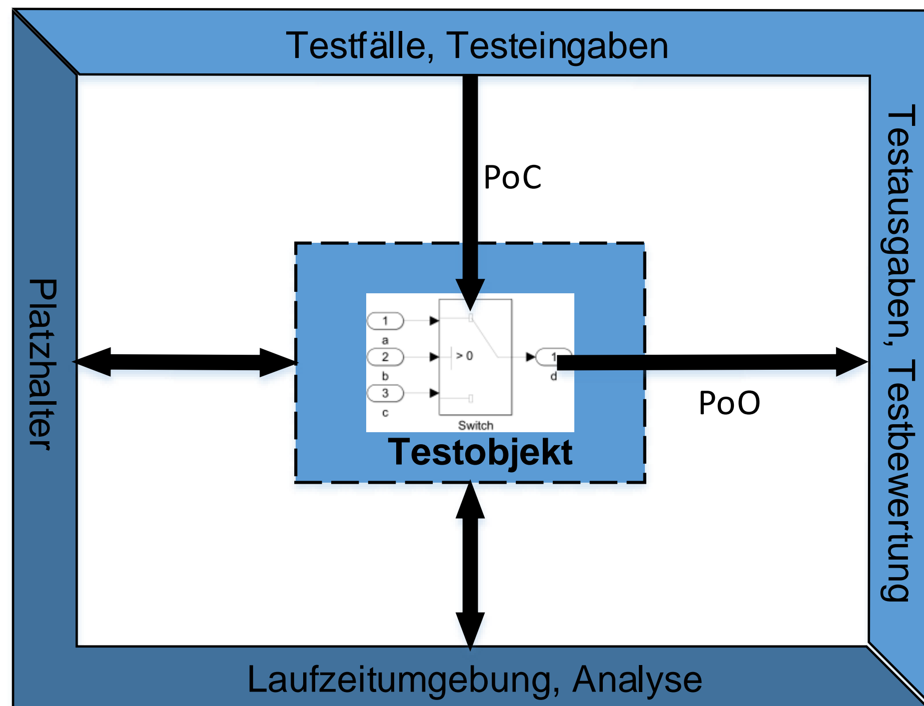


Abbildung 2.3: Testrahmen Whitebox-Verfahren

Ziel der Whitebox-Verfahren ist das Erreichen eines vorgegebenen Überdeckungsgrades des Programms durch systematisch abgeleitete Testfälle. Die Testfälle werden basierend auf der Programmlogik ermittelt, damit alle Programmteile im Idealfall mindestens einmal zur Ausführung gebracht werden. Um die erstellten Testfälle nach der Ausführung bewerten zu können, muss auch bei den Whitebox-Verfahren zusätzlich die Spezifikation als Testbasis berücksichtigt werden. Whitebox-Testverfahren lassen sich auf den unteren Teststufen wie Komponenten- und Integrationstest anwenden, sind jedoch für einen Systemtest wenig sinnvoll. [6]

Wenn die Testbasis als gerichteter Graph vorliegt (siehe Grundbegriffe der Graphentheorie in [8]), sind folgende Überdeckungskriterien, die auf die Abdeckung von Kontrollstrukturelementen zielen, üblich [9]:

- Anweisungs-/Ausführungsüberdeckung (Statement-/Execution Coverage):
  Anteil der vom Test berührten Knoten im Graphen.

- Zweig-/Entscheidungsüberdeckung (Branch-/Decision Coverage):
  Anteil der vom Test berührten Kanten/Transitionen im Graphen.

- Bedingungsüberdeckung (Condition Coverage):
  Anteil der vollständig durchlaufenden Bedingungen als *true* oder *false*.

Zusammenfassend kann durch die Erstellung struktureller Testfälle eine vollständige Abdeckung der vorhandenen Programmstruktur überprüft werden. Sie geben aber keine Garantie auf vollständige Abdeckung der Spezifikationen. Fehlender Programmcode kann dadurch nur durch Zufall erkannt werden.

Diversifizierende Tests

Diversifizierende Testtechniken gehören zu den Blackbox-Verfahren. Sie bewerten die Korrektheit von Testergebnissen durch den Vergleich mehrerer Versionen der zu testenden Software [9]. Die durch Fehlerkorrektur oder Weiterentwicklung geänderte oder ergänzte Software muss durch Wiederholung von Tests geprüft werden. Hierfür gibt es folgende Verfahren:

- Fehlernachtest:
  Überprüfung einer beabsichtigten Fehlerkorrektur.
- Regressionstest:
  Überprüfung auf Seiteneffekte durch eine Fehlerkorrektur.
- Back-to-Back-Test:
  Vergleich unterschiedlich implementierter Softwareversionen auf Äquivalenz.

Ein Fehlernachtest ist der Test nach einer Fehlerbehebung, um den Nachweis zu erbringen, dass der vormalige Fehler korrigiert wurde und das gewünschte Verhalten vorliegt [6].

Ein Regressionstest ist der erneute Test eines bereits getesteten Programms nach einer Änderung, um nachzuweisen, dass durch die vorgenommene Modifikation keine neuen Defekte eingebaut oder bisher maskierte Fehlerzustände freigelegt wurden [6].

Beim Back-to-Back-Test werden mehrere Versionen einer Software, die basierend auf derselben Spezifikation entwickelt worden sind, mit gleichen Eingabedaten gegeneinander getestet und verglichen, um einen Nachweis über Gleichheit oder Unterschiede zu erbringen [9].

Diversifizierende Tests finden demnach in allen Teststufen statt und umfassen funktionale, nicht funktionale und strukturelle Tests [6].

2.3 Fundamentaler Testprozess

Als Testen wird der gesamte Prozess bezeichnet, ein Programm auf systematische Weise auszuführen, um die korrekte Umsetzung der Anforderungen nachzuweisen und um Fehlerwirkungen aufzudecken. Abbildung 2.4 zeigt den fundamentalen Testprozess nach ISTQB-Standard [6]. Dabei wird der Testprozess als Abfolge einzelner Phasen mit jeweils zugehörigen Bestandteilen dargestellt.

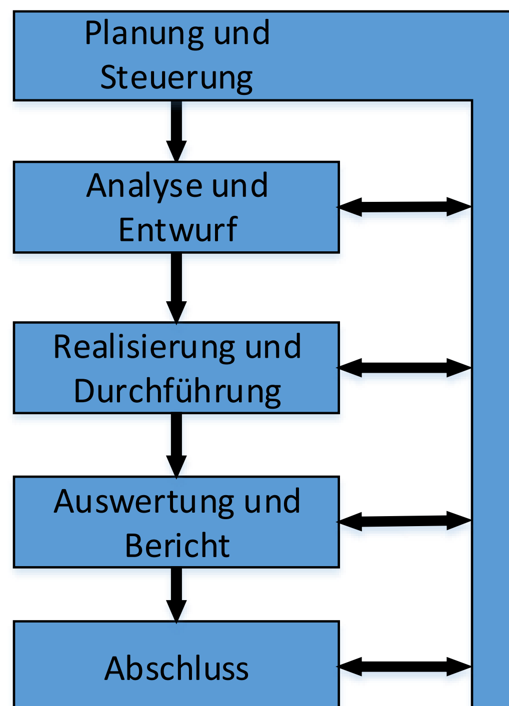


Abbildung 2.4: Testprozess nach ISTQB

Der Testprozess wird nicht als eine der Entwicklung nachgelagerte, sondern als parallel zu den Phasen des Entwicklungsprozesses stattfindende Tätigkeit verstanden. Dies ist eine Grundlage für den erfolgreichen Einsatz von Modellierungstechniken im Test. [8]

In der Phase Testplanung werden Ziele und Vorgehensweisen des Tests festgelegt. Es entsteht ein Testkonzept, welches Teststrategie, Aufwandsabschätzung und Rollenzuweisung beinhaltet. Durch die Teststeuerung werden die Aktivitäten des Testplans initiiert, der Fortschritt kontrolliert und ggf. Gegenmaßnahmen eingeleitet. [8]

Während der Testanalyse und dem Testentwurf werden die Testziele durch Anwendung der in Abschnitt 2.2 beschriebenen Testverfahren zu abstrakten Testfällen detailliert. [8]

Innerhalb der Testrealisierung werden die abstrakten Testfälle zu konkreten Testfällen formuliert. Dazu muss für die Testdurchführung zuerst die Testumgebung aus Testrahmen und Testdaten aufgebaut werden, wie in Abschnitt 2.2.2 erläutert. Die eigentliche Testdurchführung ist stark mit der folgenden Auswertephase verzahnt. [8]

Die Durchführung wird von unmittelbarer Erstellung von Testprotokollen begleitet. Um eine Wiederholung der Testfälle zu ermöglichen, müssen die Ein- und Ausgänge sowie die Vor- und Nachbedingungen dokumentiert werden. Wichtigstes Ergebnis der Auswertung sind die Fehlermeldungen innerhalb der getesteten Software. Durch die Erstellung eines Testberichts wird der Bezug der Testergebnisse zur Testreferenz hergestellt, indem Informationen zur Testabdeckung der jeweiligen Anforderung bereitgestellt werden. Dafür muss eine durchgängige Nachverfolgbarkeit von den Anforderungen bis zu den Testergebnissen und Fehlermeldungen sichergestellt werden. Dies wird durch eine durchgängige Werkzeugkette erreicht, die das Verknüpfungen von Artefakten sowie die Auswertung der entsprechenden Informationen ermöglicht. [8]

Als Artefakt wird im Zusammenhang dieser Arbeit ein Produkt bezeichnet, welches als Zwischen- oder Endergebnis in der Softwareentwicklung entsteht [6].

2.4 Modellbasierter Test

„Modellbasiertes Testen - derzeit ist wohl kaum ein anderer Begriff im Bereich des systematischen Testens von Software so verknüpft mit Innovation und Produktivität, aber auch falschen Hoffnungen und enttäuschenden Erwartungen.“ [8]

Die Kernidee von modellbasiertem Testen ist sowohl das Erreichen von Zeit- und Ressourcenersparnissen bei der Erstellung und Wartung von Tests, als auch eine Erhöhung der Testabdeckung und somit der Testqualität [8].

Modellbasiertes Testen entwickelt sich akutell von der Grundlagenforschung hin zu praktischen industriellen Anwendungen. Oft genügen im Vergleich zur Implementierung schon geringe Erweiterungen oder Anpassungen an vorhandene Modelle, um aus

ihnen für den Test relevante Artefakte „auf Knopfdruck“ erzeugen zu können, wobei der „Knopf“ erst konzipiert und realisiert werden muss. [8]

In der Primärliteratur existieren verschiedene Definitionen für modellbasiertes Testen (siehe [8]). Ein Grund dafür sind verschiedene Modellierungstechniken für verschiedene Domänen, in denen Software zum Einsatz kommt. Besonders zwischen den Bereichen eingebettete Systeme und informationsverarbeitende Systeme gibt es grundlegende Unterschiede [11].

Für die Entwicklung eingebetteter Software hat sich die mathematische Modellbildung zur Realisierung der Funktionalität (Regelung, Steuerung, Überwachung) sowie zur Simulation des Verhaltens realer physikalischer Systeme (Strecke) etabliert. Die Modellierung erfolgt häufig mit blockorientierten Modellierungs- und Simulationswerkzeugen wie MATLAB/Simulink. MATLAB/Simulink ist sowohl im akademischen als auch im industriellen Umfeld weit verbreitet und wird auch innerhalb dieser Arbeit verwendet. [12]

Aus diesem Grund beschreiben die folgenden Abschnitte den modellbasierten Test ausschließlich im Zusammenhang mit modellbasierter Entwicklung.

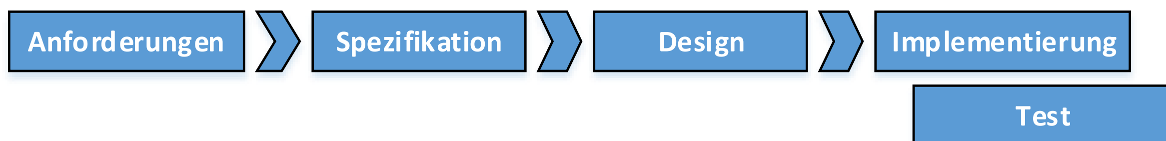
2.4.1 Modellbasierter Testprozess

Modellbasiertes Testen wird für diese Arbeit definiert als „Entwicklungsbegleitender Testprozess im Rahmen der Modellbasierten Entwicklung, der eine Kombination unterschiedlicher, sich gut ergänzender Testverfahren umfasst und dabei das ausführbare Modell als reichhaltige Informationsquelle für den Test benutzt“ [12].

Bei modellbasierter Entwicklung kann bereits während der Spezifikationsphase ein ausführbares Modell der Steuerungs- und Regelungssoftware (Funktionsmodell) als auch ein Modell des umgebenden Systems (Streckenmodell) in der Beschreibungssprache des Modellierungs- und Simulationswerkzeugs erstellt und im Verbund simuliert werden. Das Streckenmodell wird im weiteren Entwicklungsverlauf schrittweise durch das reale System und dessen Umgebung ersetzt. Das Funktionsmodell dient als Basis für die Implementierung der eingebetteten Software durch Codegenerierung. [12]
Das Funktionsmodell spezifiziert dabei die Funktionen, gibt ein Design für die Umsetzung der Funktionen vor und ist Grundlage der Implementierung mittels Codegenerie-

rung. In der Praxis werden diese verschiedene Anforderungen an das Funktionsmodell dadurch realisiert, dass es eine evolutionäre Weiterentwicklung (Modellevolution) von einem frühen Systemverhaltensmodell über ein physikalisches Modell bis zu einem Implementierungsmodell gibt [12]. Diese Weiterentwicklung des Funktionsmodells ist in Abbildung 2.5 dargestellt.

Traditionelle Softwareentwicklung:



Modellbasierte Softwareentwicklung:

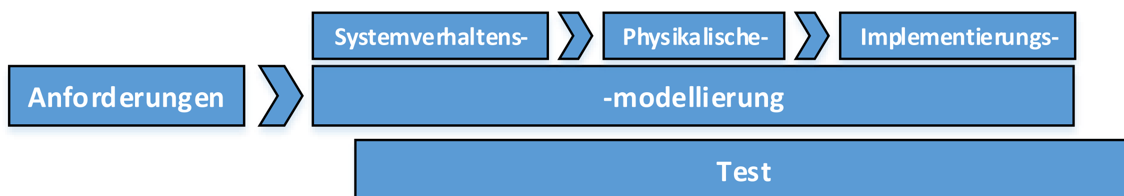


Abbildung 2.5: Traditionelle und modellbasierte Softwareentwicklung

Systemverhaltensmodelle beschreiben bestimmte Systemeigenschaften wie das generelle Ein- und Ausgabeverhalten auf einer sehr hohen Abstraktionsstufe und können auch informelle Anteile wie textuell beschriebene Übergangsreaktionen beinhalten [13].

Im physikalischen Modell werden Algorithmen für die gewünschte Steuerungs-, Regelungs- und Überwachungsfunktionalität auf Grundlage physikalischer Größen erstellt [13].

Im weiteren Verlauf wird das physikalische Modell auf die Zielplattform zugeschnitten. Das plattformspezifische Implementierungsmodell dient dann als Basis für die automatische Codegenerierung [13].

Im Vergleich zur traditionellen Softwareentwicklung ist außerdem ein stärkeres Zusammenwachsen der Spezifikations-, Design- und Implementierungsphase gegeben.

Durch das frühzeitige Vorliegen ausführbarer Funktionsmodelle können diese mit unterschiedlichen Techniken simuliert, erprobt und geprüft werden. Auftretende Fehler können dadurch früh erkannt und kostengünstig beseitigt werden, was zu einer Qualitätssteigerung in der frühen Entwicklungsphase führt. Die hohe Qualität durch Simulation und Test auf Modellebene wird durch die automatische Codegenerierung bis zur Systemintegration auf der Zielplattform übertragen. [13]

Neben der bereits etablierten interaktiven Modellsimulation hat sich der modellbasierte Test als spezielle Ausprägung zum systematischen Testen eingebetteter Software eingereiht [12].

Voraussetzung für erfolgreiches modellbasiertes Testen ist die Adaption der in Abschnitt 2.2 vorgestellten Testverfahren an das gegebene Umfeld der modellbasierten Entwicklung.

Während der modellbasierten Entwicklung ergeben sich im Gegensatz zur klassischen Softwareentwicklung mehrere ausführbare Artefakte zu einer Funktionalität, wie in Abbildung 2.6 dargestellt.

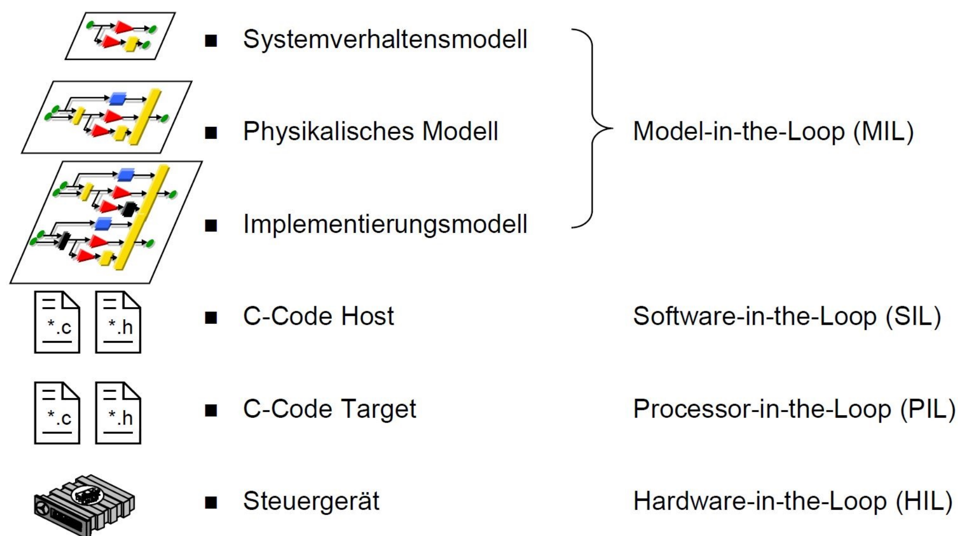


Abbildung 2.6: Testartefakte beim modellbasierten Test [14]

Wenn diese Artefakte mit den unterschiedlichen Ausführungsplattformen (z.B. Host-PC, Evaluierungsboard, Steuergerät) und Umgebungen (z.B. simulierbares Umgebungsmodell, Fahrzeug) kombiniert werden, resultieren eine Vielzahl von Testmöglich-

keiten und somit neue Herangehensweisen an den Software-Test. Durch eine Teststrategie müssen für einen gründlichen Test der Funktionalität nicht alle Testmöglichkeiten ausgeschöpft werden. [13]

Der Testprozess in der modellbasierten Entwicklung wird ähnlich dem in Abbildung 2.4 gezeigten ISTQB-Standard in folgende fünf Testaktivitäten unterteilt [12]:

- Beschreibung der Testfälle
- Auswahl konkreter Testdaten
- Testrahmenerstellung
- Testdurchführung
- Testauswertung und -dokumentation

Die einzelnen Schritte innerhalb der im Folgenden aufgeführten Teststrategie greifen in alle Aktivitäten des Testprozesses ein.

2.4.2 Teststrategie für den Modellbasierten Test

In [12] wird eine Teststrategie für den modellbasierten Test vorgestellt, welche die ausführbaren Modelle in den Mittelpunkt der analytischen Qualitätssicherung rückt. Durch Kombination von funktionsorientierten und strukturorientierten Testverfahren wird im Gegensatz zu der etablierten interaktiven Modellsimulation (Ad-Hoc-Test, Zufallstest) eine höhere Fehleraufdeckungswahrscheinlichkeit gewährleistet. Die Teststrategie für den modellbasierten Test geht dabei in folgenden zwei Punkten über eine allgemeine Teststrategie (vgl. [6]) hinaus:

- Sie berücksichtigt Testverfahren bei denen das ausführbare Modell der Software als Testreferenz bzw. Testbasis dient.
- Sie berücksichtigt die verschiedenen Artefakte bzw. Modellevolutionen des Testobjekts (vgl. Abbildung 2.6), die im Rahmen modellbasierter Entwicklung auftreten.

Der Schwerpunkt der Teststrategie liegt in der systematischen Erstellung von Testfällen aus den funktionalen Anforderungen, den Schnittstellen und dem ausführbaren Modell

der eingebetteten Software. Ergänzend wird ein geeignetes Strukturtestkriterium auf Modellebene definiert, anhand dessen die Qualität in Form der Modellüberdeckung der so ermittelten Tests bewertet werden kann. [3]

Diese Teststrategie wird als Grundlage für die in dieser Arbeit zu evaluierende Simulink-Toolkette festgelegt. Sie lässt sich durch folgende Agenda darstellen:

Nr.	**Schritt**	**Validierungsbedingung**
1	Systematischer Funktionstest auf Modellebene (Blackbox-Test)	ausreichende Anforderungsüberdeckung ausreichende Wertebereichsüberdeckung
2	Monitoring der Modellüberdeckung	ausreichende Modellstrukturüberdeckung (Model Coverage)
3	Strukturtest auf Modellebene (Whitebox-Test)	ausreichende Modellstrukturüberdeckung (Model Coverage)
4	Back-to-Back-Tests in der Software und dem eingebetteten System	Funktionale Äquivalenz zwischen Modell und Software bzw. eingebettetem System

Tabelle 2.1: Eingesetzte Teststrategie für den modellbasierten Test nach [12]

Die sich aus den Schritten 1 bis 4 ergebende Teststrategie mit den im Testprozess verankerten Testaktivitäten ist in Abbildung 2.7 schematisch dargestellt.

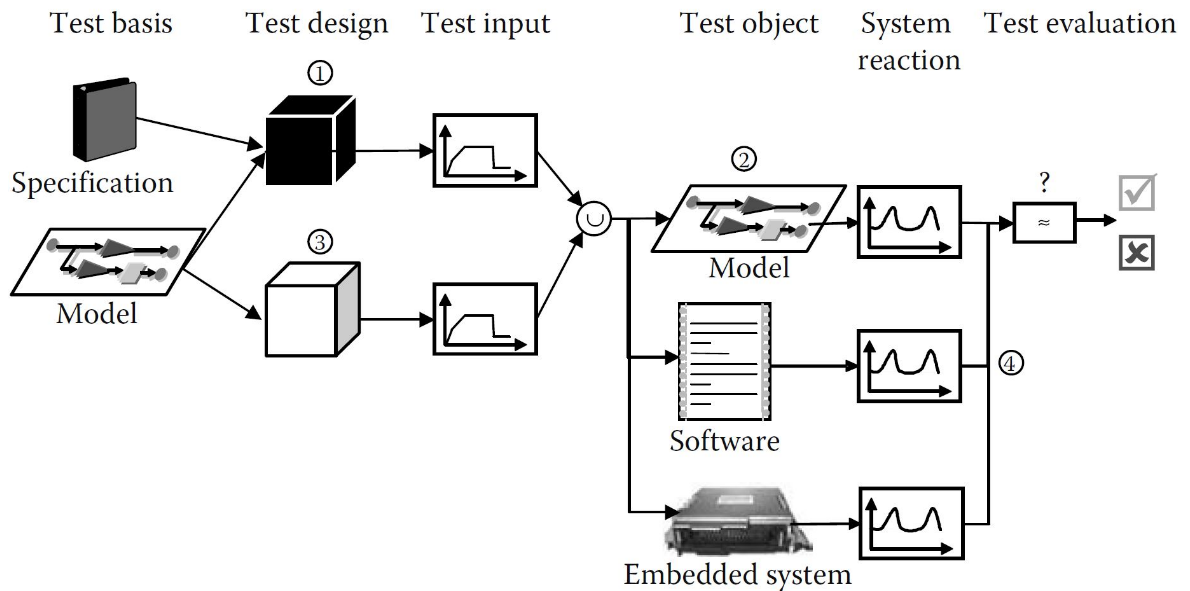


Abbildung 2.7: Teststrategie für den modellbasierten Test [14]

Bei der Umsetzung einer solchen Teststrategie in der Praxis ergibt sich das Problem, eine Testumgebung bereitzustellen, die es erlaubt, das System mit den verschiedenen Entwicklungsstufen Model-in-the-Loop (MIL), Software-in-the-Loop (SIL), Processor-in-the-Loop (PIL) und Hardware-in-the-Loop (HIL) (vgl. Abbildung 2.6) durchgängig mit denselben Testfällen zu testen [13].
Die Simulink-Toolkette versucht dieser Problematik Rechnung zu tragen und wird im weiteren Verlauf dieser Arbeit hinsichtlich der Leistungsfähigkeit untersucht. Dabei werden in Kapitel 6 und Kapitel 7 die einzelnen Schritte des modellbasierten Testprozesses und der Teststrategie mit den entsprechenden Simulink-Tools am Beispiel des Implementierungsmodells (MIL-Simulation) für die eingebettete Software des Motorcontrollers EpOS evaluiert und durchgeführt.

3 Anforderungserhebung

Die Anforderungserhebung sowie der Systementwurf erfolgen auf Grundlage der CONSENS®-Methode. Hierbei handelt es sich um eine Spezifikationstechnik zum systematischen Entwickeln von Systemarchitekturen mechatronischer Systeme, die von der Smart Mechatronics GmbH am Institut für Systemtechnik geschult wird. [15]
In Abbildung 3.1 ist die Einordnung von CONSENS® innerhalb des V-Modells dargestellt. Das Einsatzgebiet von CONSENS® liegt im Bereich Requirements Engineering und Systems Engineering.

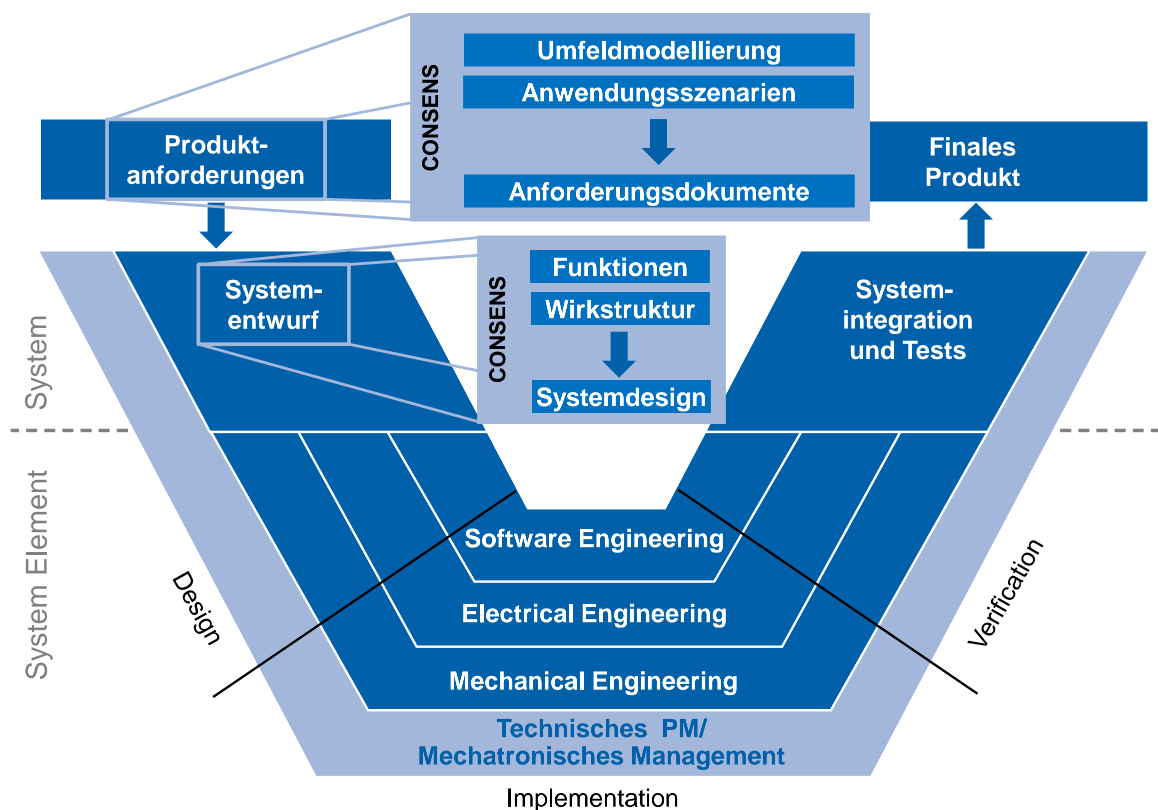


Abbildung 3.1: Einordnung von CONSENS® im V-Modell [15]

Die Aspekte Umfeldmodellierung und Anwendungsszenarien werden genutzt, um Dokumente mit Produktanforderungen wie einem Lastenheft zu erstellen. Durch Beschreibung der zu erbringenden Funktionalität sowie Erstellung einer Wirkstruktur erfolgt der Systementwurf. [15]

Der in diesem Projekt durchgeführte Workflow ist in Abbildung 3.2 dargestellt. Es werden die sechs Schritte Umfeldmodellierung, Anwendungsszenarien, Anforderungsdokumente, Wirkstruktur/Systemarchitektur, Review Systemarchitektur und Designspezifikation durchlaufen.

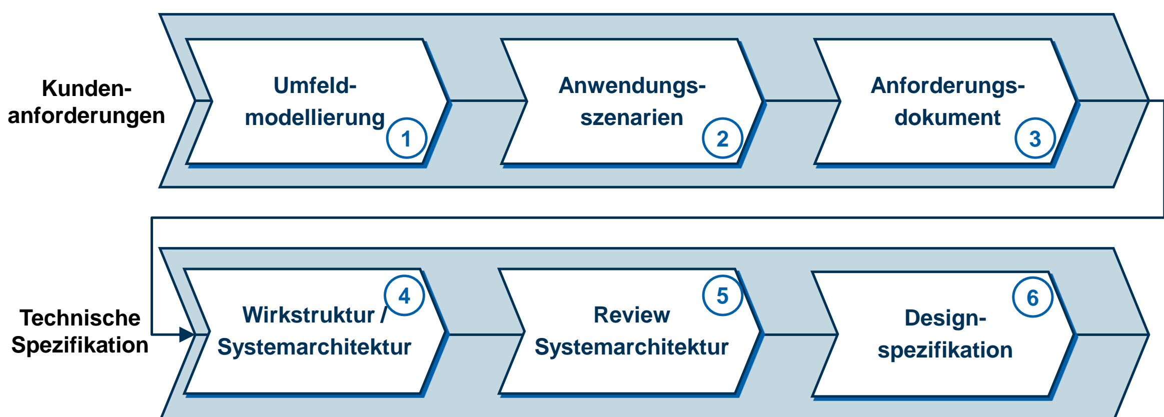


Abbildung 3.2: Durchgeführter Worfkflow nach CONSENS® [15]

3.1 Umfeldmodellierung

Innerhalb der Umfeldmodellierung wird das System E-Mountainboard als Blackbox betrachtet, wobei das Systemumfeld definiert wird und dessen Schnittstellen zum System spezifiziert werden. Dadurch wird die Entwicklungsaufgabe eingegrenzt. [15]

Abbildung 3.3 zeigt das für das E-Mountainboard entwickelte Umfeldmodell. Systemelemente sind blau eingefärbt, Umfeldelemente werden gelb dargestellt. Die Schnittstellen zwischen Systemelement und Umfeldelementen sind durch gerichtete Pfeile repräsentiert. Die unterschiedlichen Bedeutungen können der Legende in Abbildung 3.3 entnommen werden.

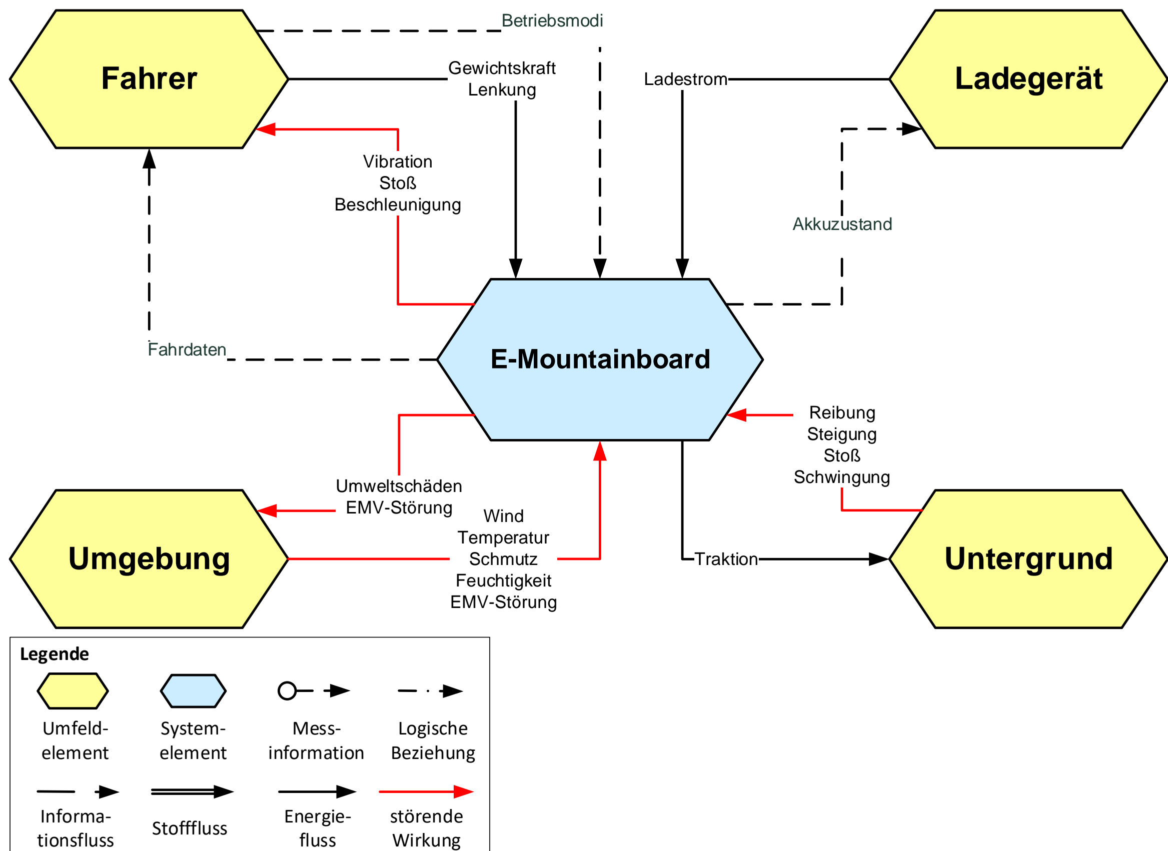


Abbildung 3.3: Umfeldmodell E-Mountainboard EpOS

Das zu entwickelnde E-Mountainboard ist durch Energie-, Informations- sowie Störungsaustausch von seinen Umfeldelementen Fahrer, Umgebung, Untergrund und Ladegerät abgegrenzt.

3.2 Anwendungsszenarien

Mithilfe von allgemeinen Beschreibungen möglicher Nutzungssituationen des Systems folgt eine Nutzungsanalyse, durch welche die Funktionalität des Systems festgelegt wird. Ein sogenanntes Anwendungsszenario beinhaltet eine Vorbedingung, eine Beschreibung des Systemverhaltens und eine Nachbedingung. [15]

Die mit der CONSENS®-Methode aufgestellten Anwendungsszenarien des E-Mountainboards sind dieser Arbeit angehängt (siehe Anhang A.1.1).

3.3 Systemanforderungen

Auf Grundlage von Umfeldmodellierung und Anwendungsszenarien sind die Anforderungen gegen das E-Mountainboard in einem Lastenheft festgehalten. Normalerweise wird ein Lastenheft vom Auftraggeber erstellt und an den Auftragnehmer übergeben [16]. Die darin enthaltenden Anforderungen beschreiben die Eigenschaften und Funktionalitäten des zu entwickelnden Produkts [17]. Innerhalb dieses Projekts ist das Lastenheft in Absprache mit dem Auftraggeber vom Auftragnehmer erstellt worden. Die Freigabe des Lastenheftes erfolgte durch den Auftraggeber.

Jede Anforderung hat zur Nachverfolgbarkeit eine eindeutige Anforderungsnummer (ANF_01 bis ANF_35) sowie einen beschreibenden Titel. Die Anforderungsbeschreibung ist lösungsneutral gehalten, um den Systementwurf und die Implementierung nicht einzuschränken. Einige Anforderungen wie die zu verwendende Mikrocontroller-Familie und die Anzahl der Motoren enthalten jedoch seitens des Auftraggebers bewusst Lösungsvorgaben, um eine gewünschte Systemtopologie zu gewährleisten.

Das Lastenheft beinhaltet außerdem einen Verifikationsplan, der die Verifikation jeder Anforderung behandelt. Dadurch wird sichergestellt, dass keine nicht verifizierbaren Anforderungen an das Produkt gestellt werden. Jede Anforderung verfügt über Testhinweise für die spätere Systemverifikation sowie für die Verifikation mittels modellbasierter Testfälle auf Modellebene.

Da das Lastenheft im Anhang diese Arbeit enthalten ist (siehe Anhang A.1.2), wird an dieser Stelle auf eine Beschreibung der Anforderungen verzichtet.

4 Systementwurf

Der Systementwurf erfolgt mithilfe von CONSENS®. Aus den im folgenden Abschnitt entwickelten Wirkstrukturen werden die Designspezifikationen für das Implementierungsmodell abgeleitet. Außerdem erfolgt die Auswahl des Mikrocontrollers sowie der zu verwendenden Produkte der Software-Toolkette.

4.1 Wirkstruktur

Innerhalb der Wirkstruktur wird das System als Whitebox betrachtet. Durch die Darstellung der inneren Zusammenhänge werden mehrere mögliche Systemarchitekturen entwickelt, aus denen das Projektteam eine auswählt. Für die Beschreibung der Systemarchitektur wird die aus Abbildung 3.3 bekannte Symbolik von CONSENS® genutzt. [15]

Aufgrund der verschiedenen Domänen Software Engineering, Electrical Engineering und Mechanical Engineering im vorliegenden Projekt ist es nötig, die Gesamtwirkstruktur auf der Systemebene E-Moutainboard in Wirkstrukturen domänenspezifischer Subsysteme zu detaillieren.

Da sich diese Ausarbeitung mit der modellbasierten Entwicklung und Verifikation der Antriebsplattform befasst, wird die Wirkstruktur des Systems E-Mountainboard zunächst in die Wirkstruktur des Subsystems Elektronik und anschließend in die Wirkstruktur des Subsystems Motorcontrollermodul detailliert. Die entstandene Systemarchitektur liefert den Ausgangspunkt für die Erstellung von Designspezifikationen bzw. Modellanforderungen an den zu entwickelnden Motorcontroller.

In Abbildung 4.1 ist die Wirkstruktur des Gesamtsystems E-Mountainboard mit den Umfeldelementen Fahrer, Ladegerät und Untergrund sowie deren Schnittstellen aus Energiefluss, Informationsfluss, Störfluss und mechanischen Verbindungen dargestellt.

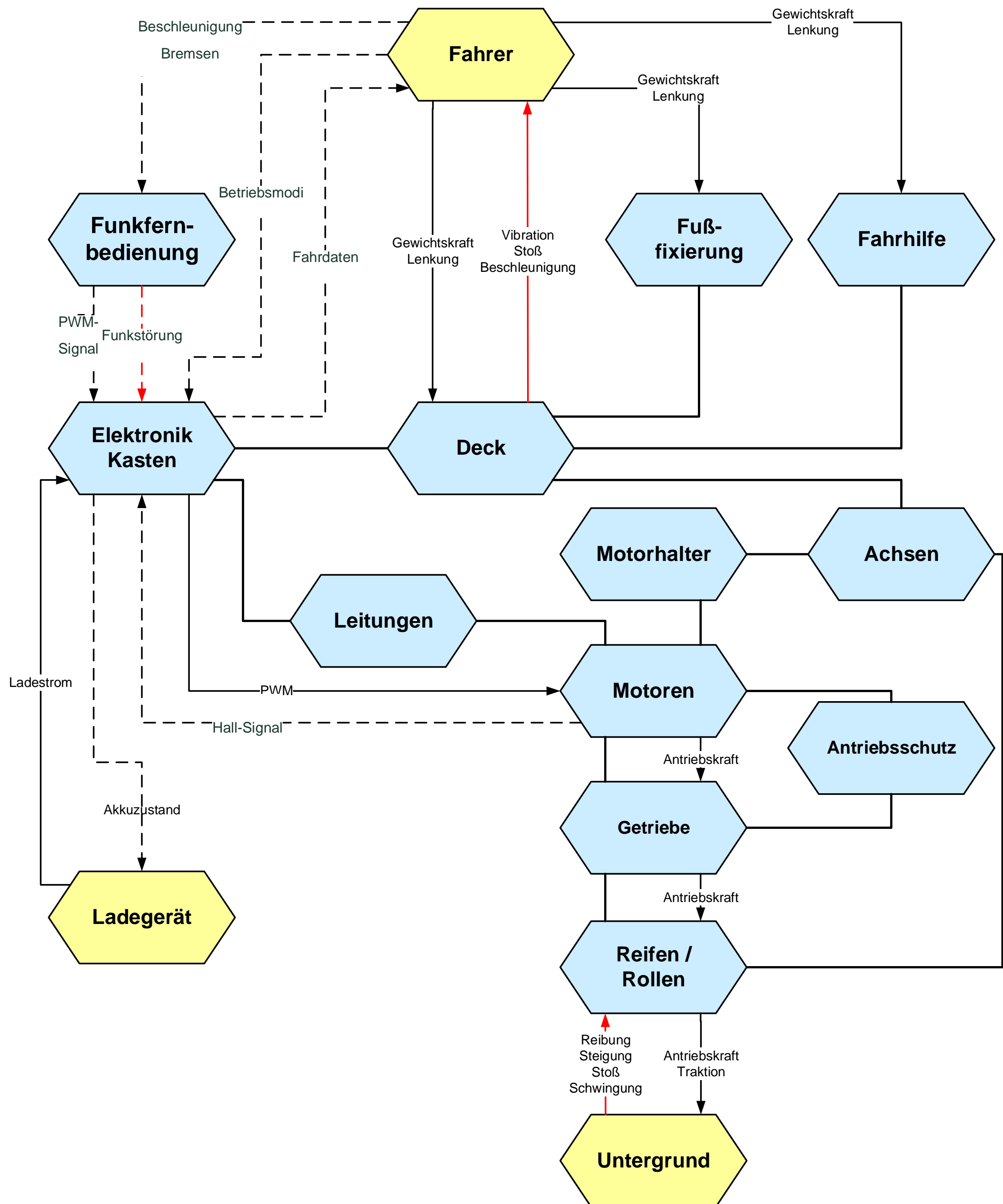


Abbildung 4.1: Wirkstruktur E-Mountainboard EpOS

Diese Gesamtwirkstruktur dient als Ausgangsbasis und wird domänenübergreifend genutzt.

Abbildung 4.2 stellt die Wirkstruktur des Systemelements Elektronik-Kasten aus Abbildung 4.1 dar. Die Systemarchitektur zeigt, dass die Elektronik über die drei Module Motorcontrollermodul I, Motorcontrollermodul II und Interfacemodul verfügt, welche über einen Akku mit elektrischer Energie versorgt werden.

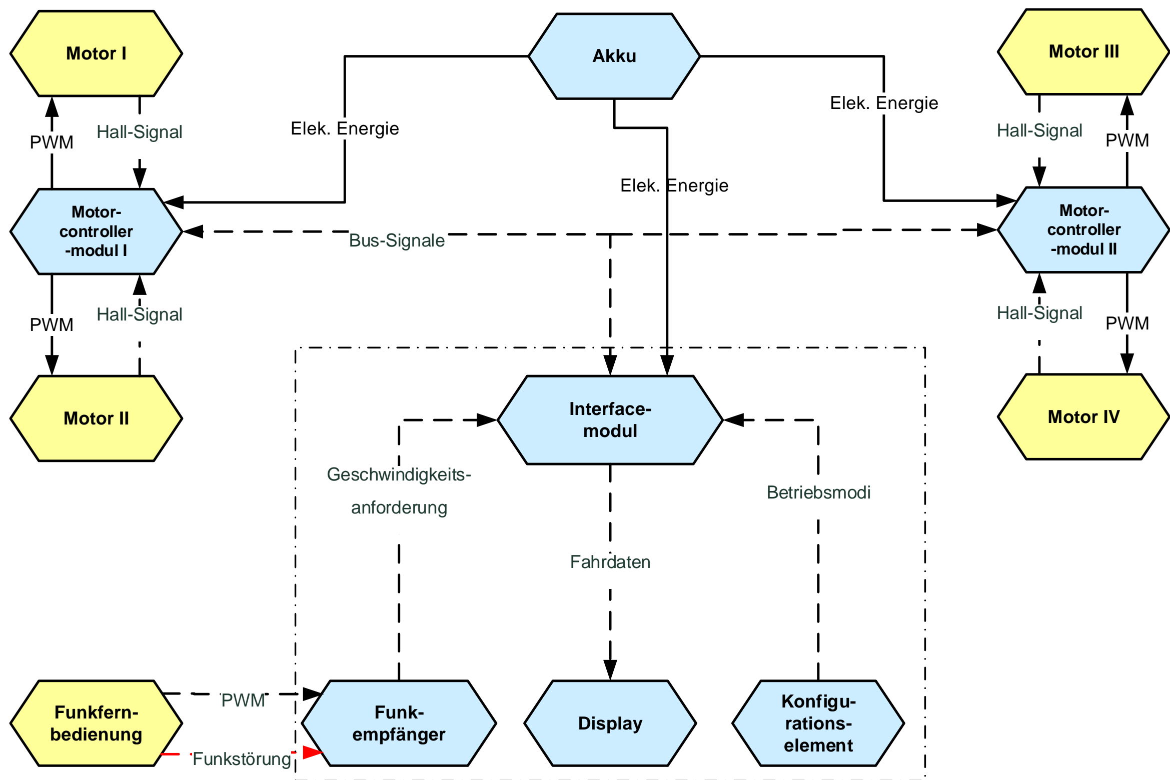


Abbildung 4.2: Wirkstruktur Elektronik

Die Notwendigkeit zu einem Systementwurf mit zwei Motorcontrollermodulen ist der zur Verfügung stehenden Hardware-Peripherie des verwendeten Mikrocontroller-Derivats geschuldet. Mit diesem ist es nicht möglich, vier separat angesteuerte Synchronmotoren zu regeln. Eine ausführliche Problembeschreibung erfolgt in Abschnitt 4.3.

Die Motorcontrollermodule haben jeweils Schnittstellen zu zwei Motoren, die als Umfeldelemente dargestellt sind. Das Interfacemodul besitzt eine Schnittstelle zum Umfeldelement Funkfernbedienung.

Aufgrund der Forderung nach einer Modularität der Antriebsplattform in Abschnitt 1.1 werden die Aufgaben, die nicht der Motorregelung dienen, auf das Interfacemodul ausgegliedert. Dazu gehört die Einstellung der Betriebsmodi und der Parametrierung, die Auswertung und Aufbereitung des Funkempfängersignals der Geschwindigkeitsanforderung sowie die Anzeige von Fahrdaten über ein Display.

Da es einer Kommunikation der drei Module untereinander bedarf, werden sie über ein Bus System vernetzt. Als Bus System wird ein CAN-Bus gewählt, da dieser sowohl in der Automobilindustrie als auch in der Automatisierungstechnik etabliert ist und vom verwendeten Mikrocontroller-Derivat unterstützt wird. Dadurch kann die zu entwickelnde Antriebsplattform wie in Abschnitt 1.1 angestrebt als Traktionswechselrichter in verschiedene Fahrzeugsysteme mit CAN-Bus integriert werden.

Die modellbasiert zu entwickelnde Regelungs- und Steuerungssoftware des Motorcontrollers muss somit über CAN-Schnittstellen verfügen, die in Abschnitt 5.2 beschrieben werden.

In Abbildung 4.3 ist die Wirkstruktur der Systemelemente Motorcontrollermodul aus Abbildung 4.2 mit den jeweiligen Umfeldelementen Interfacemodul, Motoren, Akku und Programmierschnittstelle dargestellt.

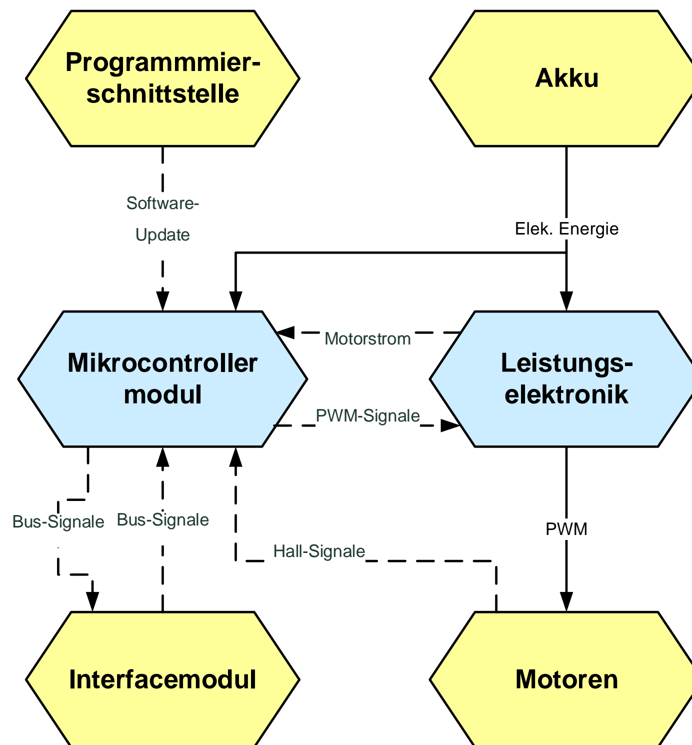


Abbildung 4.3: Wirkstruktur Motorcontrollermodul

Das Systemelement Mikrocontrollermodul muss eingangsseitig CAN-Bus-Signale, Hall-Signale und Motorstromsignale verarbeiten und ausgangsseitig entsprechende PWM-Signale für das Systemelement Leistungselektronik bereitstellen sowie Informationen über den Betriebszustand auf den CAN-Bus senden.

Innerhalb des Motorcontroller-Funktionsmodells sind die entsprechenden Schnittstellen für Ein- und Ausgangsoperationen vorzusehen. Eine detaillierte Beschreibung der Modellstruktur und dessen Schnittstellen erfolgt in Kapitel 5.

4.2 Designspezifikation

Die Designspezifikationen der Software und somit die Anforderungen an das in Simulink zu entwickelnde Implementierungsmodell des Motorcontroller werden auf Grundlage der in Abbildung 4.2 und 4.3 dargestellten Systemarchitekturen von Elektronik und Motorcontrollermodul aufgestellt. Dabei werden die Designspezifikationen für das Implementierungsmodell aus den Anforderungen des Lastenhefts abgeleitet.

Die Modellanforderungen erhalten eine eindeutige Anforderungsnummer (SW_ANF_X) und sind jeweils einer Anforderung aus dem Lastenheft (ANF_X) untergeordnet.

Das Modellanforderungsdokument und die jeweiligen Signalbeschreibungen sind dieser Arbeit angehängt (siehe Anhang A.1.3 und A.1.4).

4.3 Mikrocontroller

Zur Regelung von vier Synchronmaschinen mit Hallsensorik werden benötigt [1]:

- 24 PWM-Kanäle zur Ansteuerung von je drei Halbbrückenschaltungen
- 12 CAP-Module zur Interrupterzeugung von je drei Hallsignalen
- 8 ADC-Kanäle für die Auswertung von je zwei Phasenströmen pro Maschine

Eine Vorgabe vom Auftraggeber ist die Verwendung eines Texas Instruments (TI) Mikrocontrollers der C2000-Familie, da das Institut für Systemtechnik über Erfahrung mit dieser Mikrocontroller-Familie verfügt. Außerdem basiert auch das im abgeschlossenen Projekt entwickelte E-Skateboard auf einem TI C2000 Mikrocontroller und die TI C2000-Familie hat eine gute Anbindung an die zu verwendende *Embedded Coder*-Toolbox zur automatischen Codegenerierung von MATLAB/Simulink.

Tabelle 4.1 zeigt eine Gegenüberstellung der benötigten Hardware-Peripherie einzelner TI C2000 Mikrocontroller-Derivate [18, 19, 20].

		TI C2000 Derivat		
		F28069	**F28335**	**F28075**
Periphere	**eCAP**	3	6	6
	ePWM	16	18	24
	ADC	16	16	17
	GPIO	54	88	97

Tabelle 4.1: Vergleich möglicher TI C2000 Derivate

Im abgeschlossenen Projekt wurde das Implementierungsmodell zur Regelung von zwei Synchronmaschinen mit Hallsensorik für einen F28069-Prozessor entwickelt. Da dieses

Derivat drei eCAP-Module besitzt, mussten die fehlenden drei Hallsignal-Interrupts mithilfe externer Software-Interrupts implementiert werden. Daher wurde empfohlen das Implementierungsmodell bei Weiterentwicklung auf ein F28335-Derivat zu portieren, damit alle Subsysteme der Hallsensorauswertung identisch modelliert werden können. [1]

Da in dieser Arbeit eine Ansteuerung von vier Synchronmaschinen mit Hallsensorik realisiert wird, reicht die vorhandene Peripherie des F28335-Derivats nicht aus, um zwölf Halbbrückenschaltungen und zwölf Hallsignale anzusteuern bzw. auszuwerten (siehe Tabelle 4.1).

Ein TI C2000-Mikrocontroller mit dafür ausreichender Peripherie ist der F28075. Dieser verfügt über die 24 benötigten ePWM-Kanäle sowie über sechs eCAP-Module. Ein TI C2000-Mikrocontroller mit mehr als sechs eCAP-Modulen ist nicht verfügbar, sodass weiterhin sechs Hallsignal-Interrupts mit den implementierten externen Software-Interrupts erzeugt werden müssen.

Da es sich beim F28075 um ein neueres Derivat der TI C2000-Familie handelt, wird dieser zur Projektlaufzeit noch nicht vom *Embedded Coder Support Package for TI C2000 Processors* der Simulink-Toolkette unterstützt. Entsprechend fehlt die Simulink-Library mit den benötigten Blöcken der Hardware-Peripherie, sodass eine direkte Anbindung des Implementierungsmodells an die Zielplattform nicht möglich ist.

Zusammenfassend ist es mit der aktuell verfügbaren Hardware an TI C2000-Prozessoren und der angebotenen Simulink-Toolkette nicht möglich, vier Synchronmaschinen mit einem Mikrocontroller zu regeln.

Aus diesem Grund fällt die Entscheidung zu einem Systementwurf mit zwei separaten Motorcontrollermodulen mit je einem Mikrocontroller wie in Abschnitt 4.1 beschrieben bzw. in Abbildung 4.2 dargestellt.

Als Zielplattform werden zwei F28069 Mikrocontroller, die auf einer sogenannten controlCARD verbaut sind, ausgewählt. Es handelt sich um das gleiche Derivat, das auch im abgeschlossenen Projekt verwendet wurde. Dadurch kann die Anbindung an die Zielplattform weitestgehend aus dem vorhandenen Implementierungsmodell übernommen werden. Außerdem kann das vorhandene E-Skateboard System während der Weiterentwicklung weiterhin als Testplattform dienen. Die Verwendung der bereits implemen-

tierten externen Software-Interrupts zur Auswertung der Hallsignale wird dabei nicht als ein Nachteil gesehen, der die hier aufgeführten Vorteile des F28069 aufhebt.

4.4 Software-Toolkette

In Tabelle 4.2 sind die in dieser Arbeit verwendeten Einzelprodukte der gesamten Toolkette zur modellbasierten Entwicklung und zum modellbasierten Test aufgelistet. Dabei werden die Softwarepakete MATLAB/Simulink der Firma MathWorks und Code Composer Studio der Firma Texas Instruments verwendet.

Die Einrichtung und Inbetriebnahme der Toolkette ist [1] sowie Anhang A.1.5 zu entnehmen.

Hersteller	**Produktbezeichnung**	**Version**
MathWorks	MATLAB R2016a	9.0
	Simulink	8.7
	Embedded Coder	6.10
	Fixed-Point Designer	5.2
	Simscape	4.0
	Simulink Report Generator	5.0
	Simulink Test	2.0
	Simulink Verification and Validation	2.0
	TI C2000 Support from Embedded Coder	16.1.2
Texas Instruments	Code Composer Studio v5	5.5.0
	controlSUITE	3.4
	C2000 Compiler	6.2

Tabelle 4.2: Verwendete Software-Toolkette

5 Modellierung mit Simulink

Auf Grundlage des Systementwurfs erfolgt die Funktionsmodellierung des Implementierungsmodells sowie die Streckenmodellierung in Simulink.

Zunächst wird eine Modellstruktur entwickelt, die es ermöglicht hardwarespezifischen C-Code für den TI C2000 F28069 Mikrocontroller zu generieren, aber auch Simulationen und modellbasierte Tests durch Komponententests einzelner Subsysteme sowie Integrationstests mit angebundenem Streckenmodell gewährleistet. Nur so kann der gesamte modellbasierte Testprozess der Simulink-Toolkette am Implementierungsmodell durchgeführt werden.

In der Modellierungsphase wird das neu strukturierte Modell funktional erweitert, verbessert und dem Systementwurf entsprechend modifiziert. Der modellbasierte Test ist Teil der Implementierung bzw. erfolgt zeitlich parallel dazu und wird in Kapitel 6 beschrieben.

5.1 Modellstruktur

Das vorhandene Implementierungsmodell hat durch die Nutzung von Bibliotheken des *TI C2000 Support for Embedded Coder* eine direkte Anbindung an die Hardware-Peripherie des TI C2000 F28069 Mikrocontroller (siehe [1]). Dadurch kann direkt aus dem Implementierungsmodell hardwarespezifischer C-Code generiert werden.

Dies hat den Vorteil, dass der erzeugte C-Code nicht manuell an einen bestimmten Mikrocontroller angepasst werden muss, wie dies bei der Erzeugung von generischem C-Code aus einem Simulink-Modell der Fall ist. Jedoch kann das Implementierungsmodell aufgrund der angebundenen TI Library-Blöcke nicht simuliert werden, wodurch ein entscheidender Vorteil der modellbasierten Entwicklung nicht genutzt werden kann.

Außerdem ist aktuell kein Streckenmodell vorhanden, um das Verhalten des Motorcontroller-Funktionsmodells mit der Systemumgebung simulieren und analysieren zu können.

Es wird daher eine Modellstruktur entwickelt, in der die Hardwareanbindung außerhalb der Funktionsmodellierung vorgenommen werden kann und die es gleichzeitig erlaubt, ein Streckenmodell des umgebenden Systems an das Motorcontroller-Funktionsmodell anzubinden. In Abbildung 5.1 ist ein sogenanntes Minimalbeispiel der entwickelten Modellstruktur des gesamten Implementierungsmodells EpOS dargestellt.

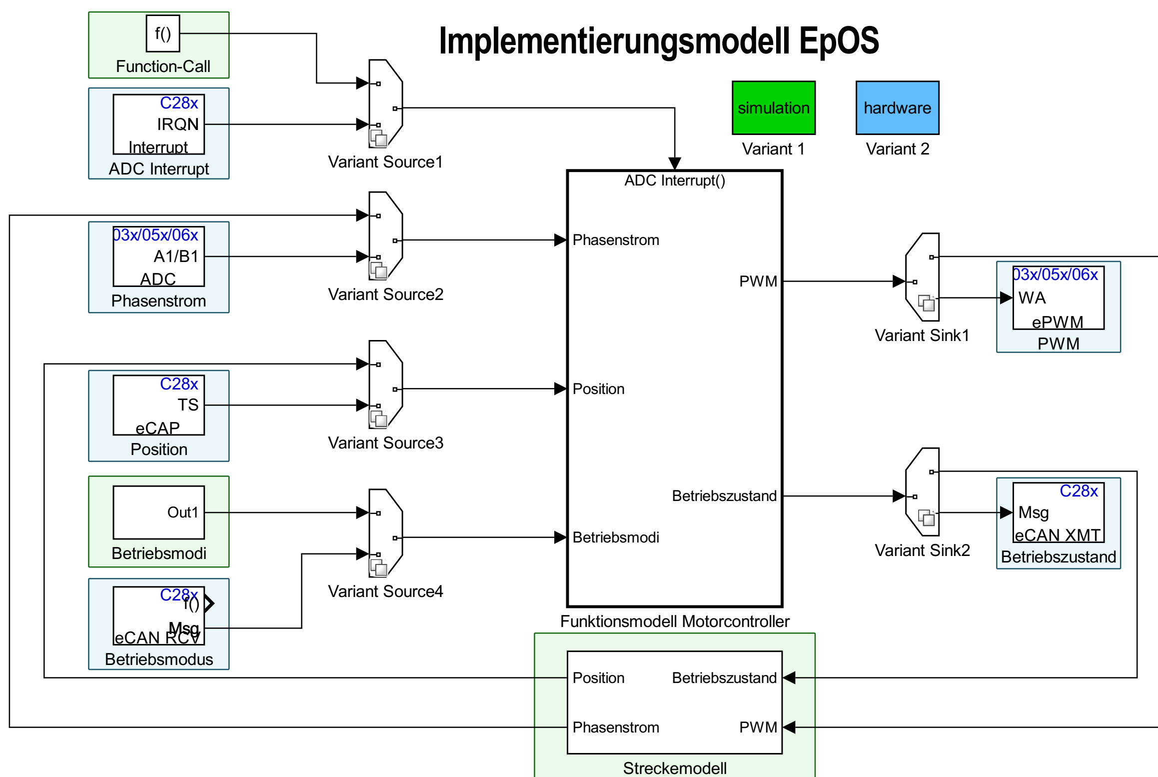


Abbildung 5.1: Minimalbeispiel der Implementierungsmodellstruktur

Das Funktionsmodell des Motorcontrollers ist frei von hardwarespezifischen Library-Blöcken und kann somit simuliert werden. Die Ein- und Ausgänge können sowohl an ein Streckenmodell, als auch an die entsprechende Hardware-Peripherie des C2000 F28069 angebunden werden. Dadurch ist es möglich, closed-loop Simulationen im Zuge des modellbasierten Tests durchzuführen und den C-Code des getesteten Funktionsmodells unverändert auf die Zielhardware zu übertragen.

Dies wird durch zwei verschiedene Modell-Varianten gewährleistet. Durch die Nutzung von *Variant Source*- und *Variant Sink*-Blöcken wird über die Bedienelemente *simulation* und *hardware* die Auswahl der Modell-Variante getroffen. Mittels eines hinterlegten Scripts wird dadurch im *Variant Manager*-Tool eine Steuervariable gesetzt, die den Signalpfad der *Variant Source*- und *Variant Sink*-Blöcke umschaltet. Die nicht ausgewählte Modell-Variante wird auskommentiert und in den darauf folgenden Aktivitäten (Simulation bzw. Codegenerierung) nicht beachtet.

Die eingangs- und ausgangsseitigen Schnittstellen des Funktionsmodells werden je nach ausgewählter Modell-Variante entweder vom Streckenmodell oder der Zielhardware verarbeitet.

5.2 Schnittstellen

Das umstrukturierte Motorcontroller-Funktionsmodell verfügt über die eingangsseitigen Schnittstellen Phasenstrom, Position und Betriebsmodi sowie die ausgangsseitigen Schnittstellen PWM und Betriebszustand (siehe Abbildung 5.1). Die Schnittstellen der modellbasiert zu entwickelnden Steuerungs- und Regelungssoftware des Motorcontrollers sind durch den Systementwurf festgelegt und werden auf Grundlage der Wirkstruktur des Motocontrollermoduls in Abbildung 4.3 bzw. Abschnitt 4.1 modelliert.

Bei Modell-Variante *simulation* kommen die Eingangssignale Phasenstrom und Position des Motorcontroller-Funktionsmodells aus dem PMSM-Modell der Streckenmodellierung, welche in Abschnitt 5.4 beschrieben wird. Die Eingangssignale für die verschiedenen Betriebsmodi werden innerhalb Simulink über eine GUI als Variablen im *Base Workspace* gesetzt. Die Betriebsmodi sind in Abschnitt 5.3.2 beschrieben.

Für die Modell-Variante *hardware* müssen die Eingangs- bzw. Ausgangssignale des Funktionsmodells an der Hardware gemessen und eingestellt bzw. für die Hardware des Motorcontrollermoduls bereitgestellt werden, welche parallel zu dieser Arbeit entwickelt wird (siehe [5]). Hierfür werden die Library-Blöcke des *TI C2000 Support for Embedded Coder* genutzt.

Wie in Abbildung 5.1 dargestellt, werden die gemessenen Phasenströme bzw. Hallsignale der Motoren mittels ADC-Modulen bzw. eCAP-Modulen und die geforderten PWM-Tastverhältnisse zur Ansteuerung der Leistungsschalter mittels ePWM-Modulen

verarbeitet und müssen passend zum Motorcontroller-Funktionsmodell skaliert werden. Diese Aufgaben werden vom C2000 F28069 Mikrocontroller verarbeitet und entsprechend als Eingangsoperationen (ADC und eCAP) bzw. als Ausgangsoperationen (ePWM) im Implementierungsmodell EpOS behandelt (siehe Anhang A.1.6).

Die Einstellung der Betriebsmodi sowie die Verarbeitung der Betriebszustände werden nicht vom C2000 F28069 Mikrocontroller gelöst, sondern sind Abbildung 4.2 entsprechend auf ein Interfacemodul ausgelagert. Die Kommunikation zwischen den in dieser Arbeit entwickelten Motorcontrollermodulen und dem Interfacemodul erfolgt über einen CAN-Bus. Dafür wird das eCAN-Modul des C2000 F28069 verwendet.

Innerhalb des Implementierungsmodells werden die Nachrichten auf dem CAN-Bus mittels *eCAN Receive*-Block empfangen und mittels *eCAN Transmit*-Block auf den CAN-Bus gesendet. Eine Übersicht aller CAN-Nachrichten und deren Konfiguration ist dieser Arbeit angehängt (siehe Anhang A.1.8).

5.3 Funktionsmodellierung

Hauptbestandteil des Motorcontroller-Funktionsmodells ist die Regelung von zwei permanentmagneterregten Synchronmaschinen (PMSM) mit Hallsensorik. Die Wirkstruktur in Abbildung 4.2 sieht einen Motorcontroller pro Achse des E-Mountainboards vor, welcher die beiden PMSM, die ihr Antriebsmoment jeweils auf einen Reifen übertragen, in Einzelradansteuerung regelt.

Es werden zwei separate Implementierungsmodelle für die beiden Motorcontrollermodule entwickelt, die über das gleiche Funktionsmodell verfügen. Lediglich die CAN-Nachrichten der Betriebszustände haben andere IDs, um die Daten dem jeweiligen Motor zuordnen zu können und eine Kommunikation der beiden Motorcontroller untereinander zu gewährleisten. Auch die Regelungsalgorithmen für die beiden Motoren innerhalb eines Motocontrollermoduls sind identisch aufgebaut, sodass im weiteren Verlauf im Singular berichtet wird.

Die Basis für die Funktionsmodellierung ist das vorhandene Implementierungsmodell des abgeschlossenen E-Skateboard Projektes (siehe [1]). Daher werden im Folgenden ausschließlich die funktionalen Erweiterungen und Modifikationen knapp erläutert.

Der Schwerpunkt der Ergebnisdokumentation dieser Arbeit liegt im modellbasierten Test des entwickelten Motorcontroller-Funktionsmodells in Kapitel 6 sowie in der anschließenden Modellverifikation in Abschnitt 7.1.

5.3.1 Regelungskonzept

Als Regelungskonzept der PMSM kommt die feldorientierte Regelung (FOC), auch Vektorregelung genannt, zum Einsatz. Die feldorientierte Regelung hat sich aufgrund sehr guter Regelungseigenschaften etabliert und gilt heutzutage als Industriestandard bei Drehfeldmaschinen für drehzahl- und positionsspezifische Aufgaben. Durch die feldorientierte Regelung ist es möglich, den Statorstrom in Betrag, Frequenz und Phase zu regeln, wodurch die gleiche Regelungsstrategie wie bei einer fremderregten Gleichstrommaschine resultiert [21]. Die Funktionsweise der FOC wird nicht innerhalb dieser Arbeit erläutert und ist [1] oder [22] zu entnehmen.

Zur Übersicht ist in Abbildung 5.2 ein Blockschaltbild der Gesamtstruktur des Motorcontroller-Funktionsmodells dargestellt.

Abbildung 5.2: Gesamtstruktur des Motorcontroller-Funktionsmodells

Es handelt sich um eine kaskadierte Regelung mit innerem Stromregelkreis durch FOC, der seine Führungsgröße vom äußeren Drehzahlregler erhält. Die Führungsgröße n_w

des äußeren Drehzahlregelkreises wird dem Drehzahlregler von außen vorgegeben und ist abhängig von den Faktoren Funkfernbedienungssignal, Geschwindigkeitsmodus, Beschleunigungsrampe und Traktionskontrolle.

Über eine Funkfernbedienung wird über die Hebelstellung eine gewünschte Geschwindigkeit vorgegeben. Das Signal der Funkfernbedienung wird, wie in Abbildung 4.2 dargestellt, vom Interfacemodul verarbeitet und auf den CAN-Bus gesendet. Die CAN-Nachricht wird vom Implementierungsmodell empfangen und zum Signal n_{remote} verarbeitet. Abhängig vom eingestellten Geschwindigkeitsmodus, welcher in Abschnitt 5.3.3 erläutert ist, wird die angeforderte Geschwindigkeit n_{in} berechnet.

Aufgrund einer Beschleunigungsrampe wird die vom Fahrer angeforderte Geschwindigkeit n_{in} nicht direkt als Führungsgröße n_{w} auf den Drehzahlregler gegeben, sondern je nach Beschleunigungsmodus (siehe Abschnitt 5.3.2) unterschiedlich schnell als n_{out} in variablen Intervallen einer Rampenfunktion inkrementiert. Solange die Traktionskontrolle, welche in Abschnitt 5.3.4 erläutert ist, keinen Schlupf detektiert, wird das Signal n_{out} als Führungsgröße n_{w} auf den Drehzahlregler geschaltet.

Detektiert die Traktionskontrolle einen definierten Schlupf zwischen aktueller Motorgeschwindigkeit n und Referenzgeschwindigkeit n_{ref}, wird n_{ref} als Führungsgröße n_{w} auf den Drehzahlregler geschaltet.

5.3.2 Betriebsmodi

Über ein intuitives Konfigurationsmenü des Interfacemoduls können verschiedene Betriebsmodi eingestellt werden, um das Fahrverhalten des E-Mountainboards an unterschiedliche Fahrer- und Energieverbrauchslevel anzupassen. Die möglichen Konfigurationen im Implementierungsmodell EpOS sind:

- Funkfernbedienungssignal
- Antriebsachse Front/Heck/Allrad
- Motorfreilauffunktion aktivieren/deaktivieren
- Bremsfunktion aktivieren/deaktivieren
- Traktionskontrolle aktivieren/deaktivieren

- maximale Geschwindigkeit
- maximale Beschleunigung
- maximale Bremsverzögerung

Die Betriebseinstellungen werden über den CAN-Bus empfangen und sind dadurch im Implementierungsmodell beliebig erweiterbar. Das Implementierungsmodell kann somit der Forderung nach einer Modularität der Antriebsplattform gerecht werden. Es können beispielsweise folgende Betriebseinstellungen über den CAN-Bus mit verglichen zur Implementierung geringem Aufwand ergänzt werden, um den Motorcontroller gekapselt in ein anderes System zu integrieren:

- Einlesen der Motordrehrichtung
- Einlesen der Polpaarzahl einer PMSM
- Einlesen des Hallsensor Offset
- Einlesen der Getriebeuntersetzung
- Einlesen des Reifendurchmessers
- Parametrierung der Regler
- Wechsel zwischen Geschwindigkeitsregelung und Drehmomentenregelung

In der Modell-Variante *simulation* werden die Signale der verschiedenen Betriebseinstellungen innerhalb Simulink über eine GUI abgefragt und über das *Base Workspace* für die Simulation eingelesen. Daher ist es möglich das Verhalten des Motorcontroller-Funktionsmodells mit angeschlossenem Streckenmodell mit allen Betriebsmodi zu simulieren, zu analysieren und innerhalb der modellbasierten Tests zu nutzen.

5.3.3 Geschwindigkeitsmodus

Die eingestellte maximale Geschwindigkeit n_{max} wird, wie in Abschnitt 5.3.2 beschrieben, über den CAN-Bus empfangen. Die Maximalgeschwindigkeit kann auf die Werte 5 $\frac{\text{km}}{\text{h}}$, 10 $\frac{\text{km}}{\text{h}}$, 15 $\frac{\text{km}}{\text{h}}$, 20 $\frac{\text{km}}{\text{h}}$, 25 $\frac{\text{km}}{\text{h}}$ und 30 $\frac{\text{km}}{\text{h}}$ eingestellt werden.

Durch das Subsystem Geschwindigkeitsmodus (siehe Komponente in Abbildung 5.2), das im Motorcontroller-Funktionsmodell durch das Subsystem *Max Speed through Speedmode* umgesetzt ist, wird das Signal der Funkfernbedienung n_{remote} so verarbeitet, dass die angeforderte Geschwindigkeit n_{in} am Ausgang des Subsystems die eingestellte maximale Geschwindigkeit n_{max} nicht überschreitet. In Abbildung 5.3 ist die Funktion des Subsystems grafisch dargestellt.

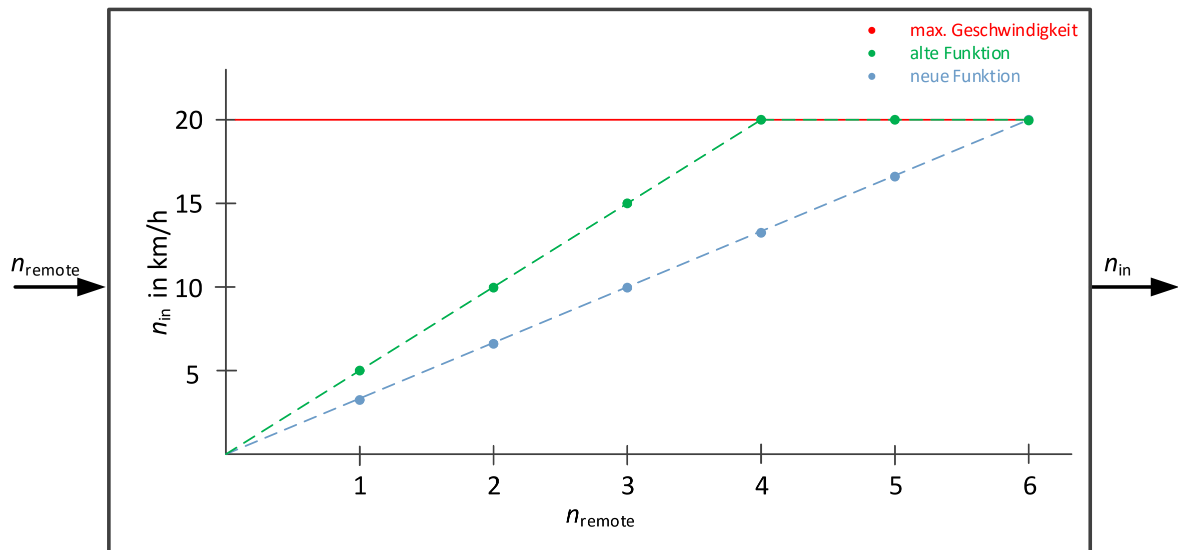


Abbildung 5.3: Funktion des Subsystems Geschwindigkeitsmodus

Die gewählte maximale Geschwindigkeit n_{max} beträgt für dieses Beispiel 20 $\frac{\text{km}}{\text{h}}$. Im alten Implementierungsmodell wurde die Funktion über einen *switch case Operator* gelöst, welcher das Signal n_{remote} auf das eingestellte n_{max} begrenzt. Dadurch wird der Signalbereich n_{remote} nur bei einem n_{max} von 30 $\frac{\text{km}}{\text{h}}$ vollständig zur Dosierung der angeforderten Geschwindigkeit n_{in} ausgenutzt. Je kleiner n_{max} eingestellt ist, desto weniger kann vom Signalbereich n_{remote} genutzt werden.

Wie in Abbildung 5.3 in grün dargestellt, wird bei der alten Funktion bspw. bei einem n_{max} von 20 $\frac{\text{km}}{\text{h}}$ das letzte Drittel von n_{remote} abgeschnitten bzw. befindet sich im Sätti-

gungsbereich. Außerdem ist die Geschwindigkeitsanforderung n_{in} unabhängig von n_{max} nur in 5 $\frac{\text{km}}{\text{h}}$-Schritten dosierbar.

In der neuen Funktion stellt sich gemäß Gleichung 5.1 die angeforderte Geschwindigkeit n_{in} als lineare Gleichung dar, die den gesamten Signalbereich von n_{remote} unabhängig vom gewählten n_{max} nutzt.

$$n_{\text{in}} = n_{\text{remote}} \cdot \frac{1}{n_{\text{remote max}}} \cdot n_{\text{max}} \tag{5.1}$$

Das Funkfernbedienungssignal n_{remote} wird normiert und mit der eingestellten Maximalgeschwindigkeit n_{max} multipliziert. Dadurch ist die Geschwindigkeitsanforderung n_{in} bei kleiner werdendem n_{max} genauer dosierbar. Dies ist in Abbildung 5.3 in blau für ein n_{max} von 20 $\frac{\text{km}}{\text{h}}$ dargestellt.

Ein detaillierter Vergleich der beiden unterschiedlich implementierten Funktionen des Subsystems *Max Speed through Speedmode* erfolgt in Abschnitt 6.2.5 im Zuge eines Back-to-Back-Tests.

5.3.4 Traktionskontrolle

Eine Traktionskontrolle, auch Antriebsschlupfregelung (ASR) genannt, wirkt im Antriebsfall (Beschleunigung) eines Fahrzeugs und hat zum Ziel die Fahrstabilität sowie den Vortrieb des Fahrzeugs durch Ausnutzung des maximal möglichen Reibwerts an allen Antriebsrädern zu verbessern [23]. Das Subsystem Traktionskontrolle (siehe Komponente in Abbildung 5.2) wirkt daher einem Durchdrehen der angetriebenen Räder entgegen. Dazu ist eine Referenzgeschwindigkeit n_{ref} nötig, mit welcher die Drehzahl des angetrieben Rades n verglichen werden kann.

Im Front- oder Heck-Antriebsmodus kann die Drehzahl der nicht angetrieben Räder als n_{ref} herangezogen werden, da diese keinen Radschlupf aufweisen. Im Allradantriebsmodus gibt es im Fahrbetrieb kein Rad, bei dem ein Durchdrehen ausgeschlossen werden kann. Eine Traktionskontrolle für Fahrzeuge mit Allradantrieb benötigt zur zuverlässigen Bestimmung von n_{ref} weitere Sensoren [23].

Aktuell ist bis auf die Motorsensorik keine weitere Sensorik zur Bestimmung von n_{ref} am E-Mountainboard vorhanden. Aufgrund der Modularität des Motorcontrollers kann das E-Mountainboard aber in Folgeprojekten durch Sensoren zur Ermittlung der Fahr-

zeuggeschwindigkeit erweitert werden, deren Daten über den CAN-Bus empfangen und von der Traktionskontrolle des Motorcontroller-Funktionsmodells verarbeitet werden können. Denkbar ist die Integration des am Institut für Systemtechnik entwickelten Ortsfrequenzfilter-Sensors, welcher zur berührungslosen, optischen Messung von Geschwindigkeiten eingesetzt wird.

Im Allradantriebsmodus ist bei unterschiedlichen Reibwerten aufgrund wechselnder Fahrbahnbedingungen das gleichzeitige Durchdrehen der Räder einer Achse wahrscheinlicher, als der Bodenhaftungsverlust eines Rades von Vorder- und Hinterachse. Bei Kurvenfahrten ist es dagegen wahrscheinlicher, dass die kurvenäußeren Räder aufgrund der geringeren auf sie wirkenden Gewichtskraft die Bodenhaftung verlieren. Daher wird in der aktuellen Konfiguration des E-Mountainboards als Referenzgeschwindigkeit n_{ref} jeweils die Drehzahl des diagonalen Rades gewählt.

Das im Motorcontroller-Funktionsmodell umgesetzte Subsystem *Traction Control* berechnet den Schlupf gemäß Gleichung 5.2:

$$S = \frac{n - n_{\mathsf{ref}}}{n_{\mathsf{ref}}} \tag{5.2}$$

Bei aktivierter Traktionskontrolle wird ein Schlupfsignal *slip* auf den Ausgang des Subsystems geschaltet, sobald S größer 25 % ist und erst wieder deaktiviert, wenn S einen Wert unter 5 % erreicht. Dies ist mit der maximalen Haftreibung zwischen Reifen und Fahrbahn begründet, welche laut Reibwert-Schlupf-Diagramm auf trockenem Asphalt bei einem Schlupf zwischen 15 % und 20 % erreicht wird. [23]

Das Schlupfsignal *slip* sorgt dafür, dass anstatt dem angeforderten Geschwindigkeitssignal n_{out} die Referenzgeschwindigkeit n_{ref} als Führungsgröße n_{w} auf den Drehzahlregler geschaltet wird, wie in Abbildung 5.2 dargestellt.

Dadurch wird die Stellgröße $i_{\mathsf{s,q,w}}$ des Drehzahlreglers, welche aufgrund der kaskadierten Regelkreisstruktur gleichzeitig die Führungsgröße des Stromreglers ist, zu Null gesetzt, sodass der Motor des schlupfbehafteten Rades kein Drehmoment erzeugt bis die Referenzgeschwindigkeit n_{ref} erreicht ist.

5.3.5 Bremsfunktion

Das Abbremsen des E-Mountainboards erfolgt durch den generatorischen Betrieb der PMSM. Die Führungsgröße der drehmomentbildenden Statorstromkomponente $i_{\mathsf{s,q,w}}$ wird 90° nacheilend zum Rotorfeld gestellt, wodurch sich die PMSM im generatorischen Betrieb befindet und ein Bremsmoment über das Getriebe auf die Räder überträgt. Dadurch wird die abgebaute kinetische Energie in elektrische Energie umgewandelt und kann in den Akku des E-Mountainboards zurückgespeist werden. Ein solcher Bremsvorgang wird als Rekuperationsbremse bezeichnet.

Die Stärke der Bremsverzögerung kann durch den Fahrer dosiert werden, da dem Stromregler abhängig vom Funkfernbedienungssignal eine entsprechende Amplitude von $i_{\mathsf{s,q,w}}$ als Führungsgröße vorgegeben wird.

Im normalen Fahrbetrieb erfolgt jedoch kein aktiver rekuperierender Bremsvorgang, wenn die Führungsgröße n_{w} des Drehzahlreglers kleiner ist, als die aktuelle Drehzahl n der PMSM. Dadurch soll ein „natürliches“ Fahrverhalten des E-Mountainboards gewährleistet werden.

5.3.6 Überwachungsfunktion

Das Motorcontroller-Funktionsmodell misst aus Sicherheitsgründen während des Betriebs über Auswertung eines ADC-Moduls kontinuierlich die Spannung des Akkus. Ab einer Grenzspannung U_{min} kleiner 22 V erfolgt softwareseitig eine Unterspannungsabschaltung. In diesem Zustand kann ein Bremsvorgang eingeleitet werden, jedoch ist keine weitere Beschleunigung möglich. Dadurch wird einem „harten“ Abschalten des Akkus vorgebeugt, bei dem die gesamte Elektronik von der Versorgungsspannung getrennt wird, wodurch dem Fahrer auch keine Bremsfunktion zur Verfügung stünde.

Außerdem werden die Drehzahlen und Phasenströme aller vier PMSM auf den CAN-Bus gesendet, um die Möglichkeit der Überwachung zu gewährleisten.

5.4 Streckenmodellierung

Die Entwicklung des Streckenmodells dient dazu, das Verhalten des Motorcontroller-Funktionsmodells in der Modell-Variante *simulation* durch closed-loop Simulationen mit angeschlossenem Streckenmodell zu untersuchen und zu analysieren. Dadurch ist es unter anderem möglich, die Reglerparameter mittels der Simulationsergebnisse experimentell einzustellen, bevor in der Modell-Variante *hardware* die PMSM mit entsprechender Leistungselektronik in Betrieb genommen wird. Außerdem können realistische Signalverläufe aus dem Verhalten der PMSM als Stimuli für die modellbasierten Tests generiert werden.

Zur Modellierung des Streckenmodells werden *Simscape*-Modellierungsblöcke verwendet. *Simscape* ermöglicht die Beschreibung physikalischer Systeme innerhalb der *Simulink*-Umgebung, ohne dass die zugrundeliegenden Gleichungen abgeleitet und implementiert werden müssen.

Innerhalb des Streckenmodells (siehe Abbildung 5.1 und Anhang A.1.6) ist eine Wechselrichter-PMSM-Kombination sowie die Messung und Skalierung der physikalischen Signale passend zum Motorcontroller-Funktionsmodell implementiert.

Im Wechselrichtermodell werden die vom FOC-Algorithmus berechneten Tastverhältnisse der PWM-Kanäle der jeweils entsprechenden Versorgungsspannung zugeordnet, die mittels idealer Spannungsquellen auf die Motorphasen geschaltet wird.

Das Motormodell besteht neben der eigentlichen PMSM aus einem idealen dreiphasigen Stromsensor, einem idealen Drehgeber, einem idealen Drehmomentsensor, dem Trägheitsmoment und der Reibung des Rotors sowie entsprechender Lagerung zur mechanischen Referenz. Das *Simsape* PMSM-Modell sowie deren Beschaltung wird entsprechend der Motorparameter des Datenblatts parametriert. Es ist darauf zu achten, dass der Polradwinkel im *Simsape* PMSM-Modell anders definiert ist als in den TI DMC Library-Blöcken zur Park Transformation.

Der Verlauf der externen Last wird über einen *Signal Builder* vorgegeben. Es besteht die Möglichkeit das Streckenmodell zur Nachbildung eines realistischen Lastverlaufs um ein Längsdynamikmodell des E-Mountainboards zu erweitern.

6 Modellbasierter Test mit Simulink

In diesem Kapitel wird der modellbasierte Testprozess und die Teststrategie für den modellbasierten Test aus Abschnitt 2.4 in den Kontext der Simulink-Umgebung gebracht. Es werden dabei die Toolboxen genannt, die für den jeweiligen Schritt des Testprozesses innerhalb der Simulink-Umgebung verwendet werden. Der gesamte Testprozess mit der Simulink-Toolkette wird Schrittweise am Beispiel einzelner Subsysteme des entwickelten Implementierungsmodells EpOS aus Kapitel 5 erläutert.

6.1 Überblick

Für einen vollständigen modellbasierten Test mit Simulink werden folgende Toolboxen benötigt:

- *Simulink Verification and Validation*
- *Simulink Test*
- *Simulink Report Generator*
- *Simulink Design Verifier*

Simulink Verification and Validation bietet Tools zur Überprüfung von Modellierungsstandards, zur Anforderungsnachverfolgung und zur Modellüberdeckungsanalyse [24].

Mit *Simulink Test* ist es möglich, systematische, simulationsbasierte Tests von Modellen zu erstellen, zu verwalten und auszuführen [25].

Der *Simulink Report Generator* ist nötig, um begleitende Dokumente des modellbasierten Entwicklungs- und Testprozesses zur Anforderungsnachverfolgung, Modellüberdeckungsanalyse, Testauswertung, usw. automatisch zu erzeugen.

Durch Nutzung des *Simulink Design Verifier* können u.a. automatisch Testfälle für eine geforderte Modellüberdeckung generiert werden. Die Funktionalitäten der *Simulink Design Verifier*-Toolbox können für das entwickelte Implementierungsmodell EpOS nicht evaluiert und genutzt werden, da der *Simulink Design Verifier* keine *s-functions* verarbeiten kann, welche innerhalb der Library-Blöcke des *TI C2000 Support for Embedded Coder* genutzt werden. Außerdem darf das Modell ausschließlich aus nicht virtuellen Subsystemen bestehenden (*Atomic Units*), damit der *Simulink Design Verifier* genutzt werden kann. Dies ist jedoch für *Function-Call* getriggerte Subsysteme, die im Implementierungsmodell EpOS benötigt werden, nicht möglich. [26]

Im Folgenden werden die verwendeten Toolboxen *Simulink Verification and Validation*, *Simulink Test* und *Simulink Report Generator* vorgestellt.

In Abbildung 6.1 ist ein Überblick des gesamten Testprozesses innerhalb der Simulink-Umgebung dargestellt. Das Testobjekt wird mit einem oder mehreren Testfällen simuliert, wobei die Ergebnisse mit den Anforderungen verglichen werden. Jeder Testfall beinhaltet Eingänge zum Testobjekt, erwartete Ausgänge und Testbewertungskriterien. Durch den Entwurf von Testfallabläufen kann die Reichweite der getesteten Anforderungen erhöht werden. [25]

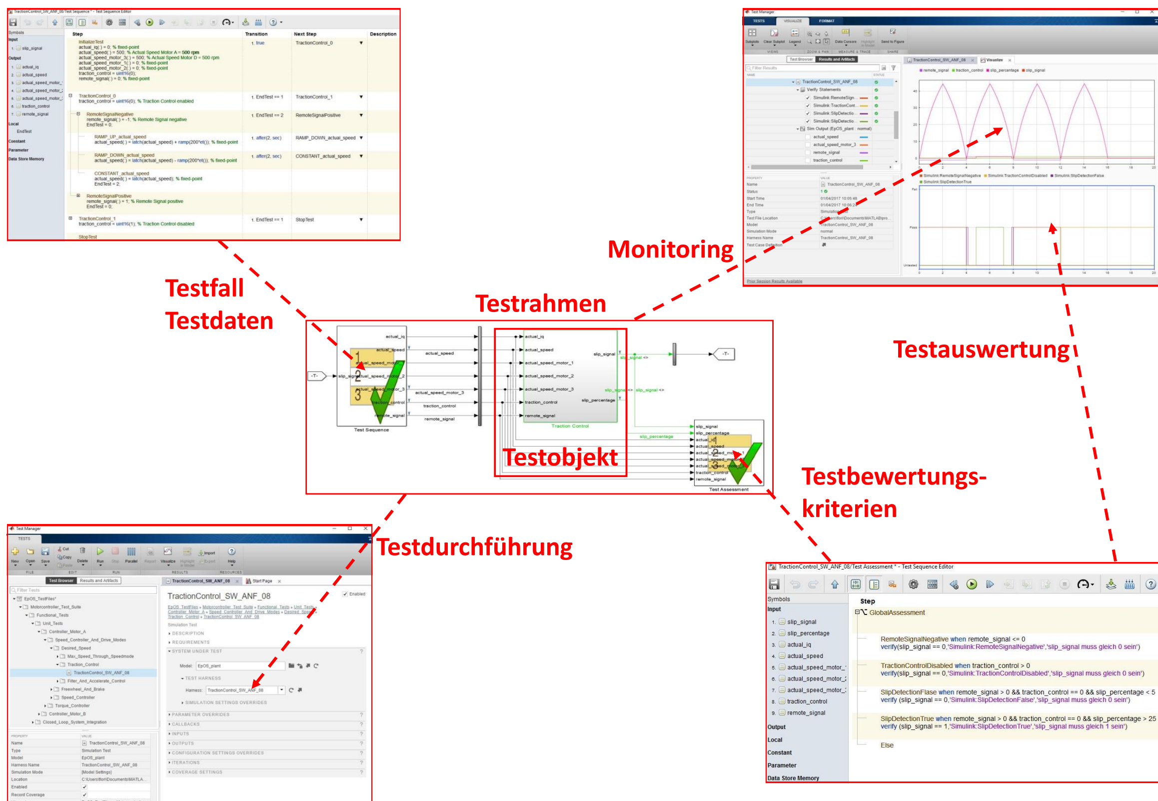


Abbildung 6.1: Durchgeführter Testprozess mit Simulink

Die jeweiligen Schritte im Testprozess mit Simulink zeigen große Parallelen mit dem in Abschnitt 2.4.1 vorgestellten modellbasierten Testprozess. Daher sind die folgenden Abschnitte dieses Kapitels unterteilt in:

- Testfallbeschreibung
- Testdatenerstellung
- Testrahmenerstellung
- Testdurchführung
- Testauswertung und -dokumentation

Mithilfe dieser Unterteilung werden die gesamten Testaktivitäten mit Simulink (siehe Abbildung 6.1) in einzelne Arbeitsschritte mit bestimmten Tools herunter gebrochen und anschaulich erläutert.

6.2 Testfallbeschreibung

In Abbildung 6.2 ist die Testfallbeschreibung innerhalb des modellbasierten Testprozesses eingeordnet.

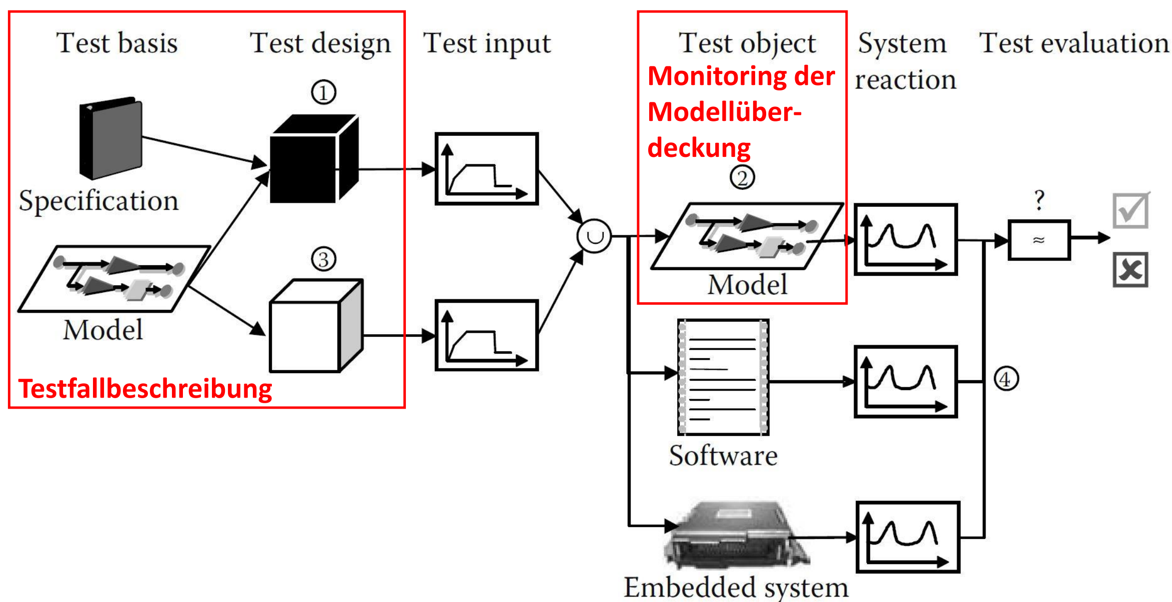


Abbildung 6.2: Einordnung der Testfallbeschreibung innerhalb des Testprozesses

Die Testfälle sind im ersten Schritt der Teststrategie für den modellbasierten Test (vgl. Tabelle 2.1 bzw. Abbildung 6.2) als Blackbox-Test zu beschreiben. Um einen Blackbox-Test zu erstellen, muss als Testbasis ein Anforderungsdokument und das Implementierungsmodell selbst herangezogen werden. Auch die Erstellung von Whitebox-Tests zur Erhöhung der Modellüberdeckung (vgl. Schritt 3 in Tabelle 2.1 bzw. Abbildung 6.2) erfordert das Implementierungsmodell als Testbasis.

Daher ist es als Grundlage für die Beschreibung eines Blackbox-Tests sinnvoll, Anforderungsdokumente mit dem umgesetzten Implementierungsmodell EpOS in Zusammenhang zu bringen. Für die Erstellung von Whitebox-Tests mit hoher Modellüberdeckung, ist es außerdem erforderlich, die erreichte Modellüberdeckung zu überwachen.

Aus diesen Gründen erfolgt vor der Beschreibung von Blackbox- und Whitebox-Tests die Umsetzung von Anforderungsnachverfolgung und Modellüberdeckungsanalyse mit Simulink.

6.2.1 Anforderungsnachverfolgung

Striktes funktionales Testen verknüpft jeden Testfall mit einer Modellanforderung. Mit dem Tool *Requirements Traceability* der *Simulink Verification and Validation*-Toolbox werden die einzelnen Modellanforderungen mit den jeweiligen Strukturen oder Blöcken des Motorcontroller-Funktionsmodells und den entsprechenden Testfällen verlinkt. Dadurch wird nachverfolgt, in welchem Modellteil eine bestimmte Anforderung erfüllt und durch welchen Testfall die Funktionalität überprüft wird (siehe Anhang A.1.9).

Durch die Erstellung von bidirektionalen Links ist es möglich, von einer bestimmten Anforderung zu dem entsprechenden Modellteil bzw. Testfall zu gelangen und umgekehrt. Dies ist in Abbildung 6.3 am Beispiel des Subsystems *Freewheel Signal and Brake Signal* dargestellt.

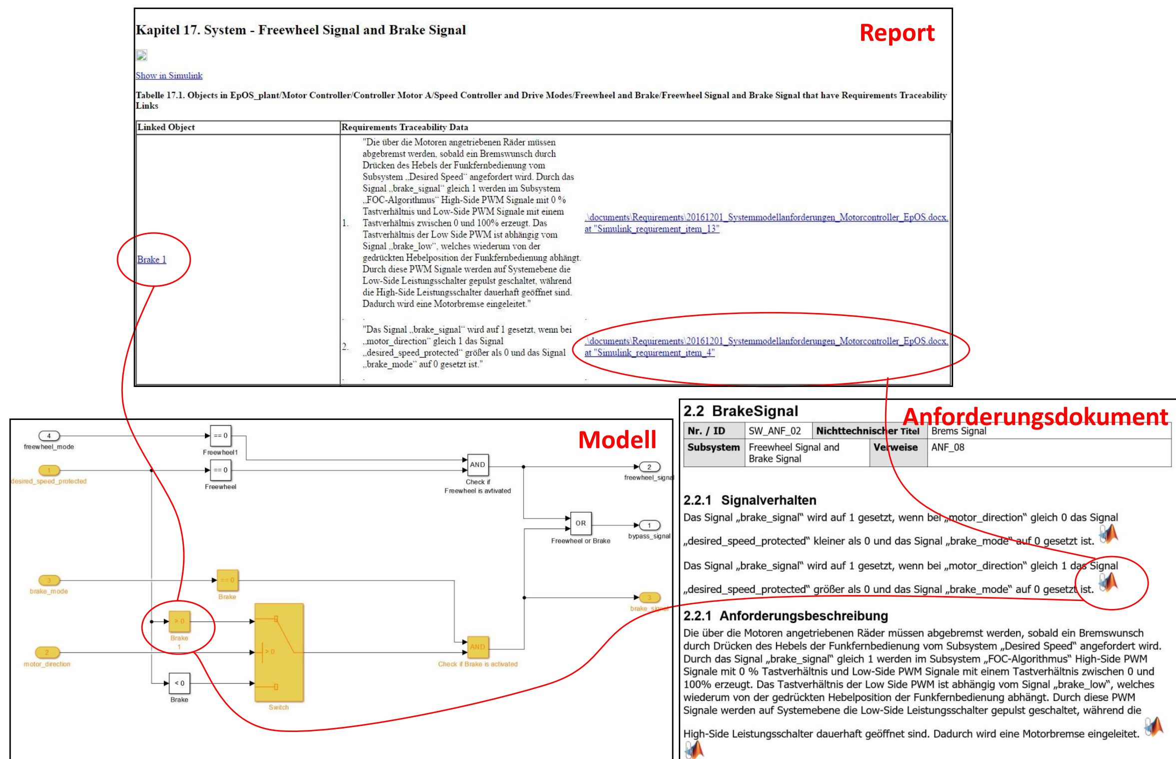


Abbildung 6.3: Anforderungsnachverfolgung *Freewheel Signal and Brake Signal*

Mithilfe des *Simulink Report Generators* wird ein Report erstellt, in dem die Modellanforderungen und das Motorcontroller-Funktionsmodell zusammengeführt werden. Es werden alle verlinkten Anforderungen dokumentiert, mit der Möglichkeit über die

hinterlegten Links sowohl in das Anforderungsdokument, als auch in das Modell zu gelangen (siehe Anhang A.1.10). Der sogenannte *Requirements Traceability Report* eignet sich für die Nutzung innerhalb von Reviews.

Die Konsistenz zwischen Modellanforderungen und Motorcontroller-Funktionsmodell wird durch das Tool *Requirements Consistency Checking* im *Model Advisor* überprüft. Änderungen im Anforderungsdokument oder im Modell führen zu Fehlermeldungen, so dass nachvollzogen werden kann, ob und wo eine Änderung vorliegt. Die gesetzten Links können nach einer beabsichtigten Änderung aktualisiert werden, um die Konsistenz wieder herzustellen. Auch für diese Funktionalität wird ein Report erzeugt (siehe Anhang A.1.11).

6.2.2 Modellüberdeckungsanalyse

Mit dem Tool *Coverage* der *Simulink Verification and Validation*-Toolbox wird die Modellüberdeckung während der Modellsimulation und dem modellbasiertem Test gemessen und analysiert. Dabei werden verschiedene Überdeckungskriterien wie Execution Coverage (EC), Condidtion Coverage (CC), Decision Coverage (DC) und Modified Condition/Decision Coverage (MCDC) analysiert.

Die Modellüberdeckungsanalyse ist der zweite Schritt der Teststrategie für den modellbasierten Test (vgl. Tabelle 2.1 bzw. Abbildung 6.2) und erfolgt in Simulink im Zuge der Testauswertung (siehe Abschnitt 6.6).

In Abbildung 6.4 ist die Modellüberdeckungsanalyse eines Whitebox-Tests des Subsystems *Freewheel Signal and Brake Signal* dargestellt.

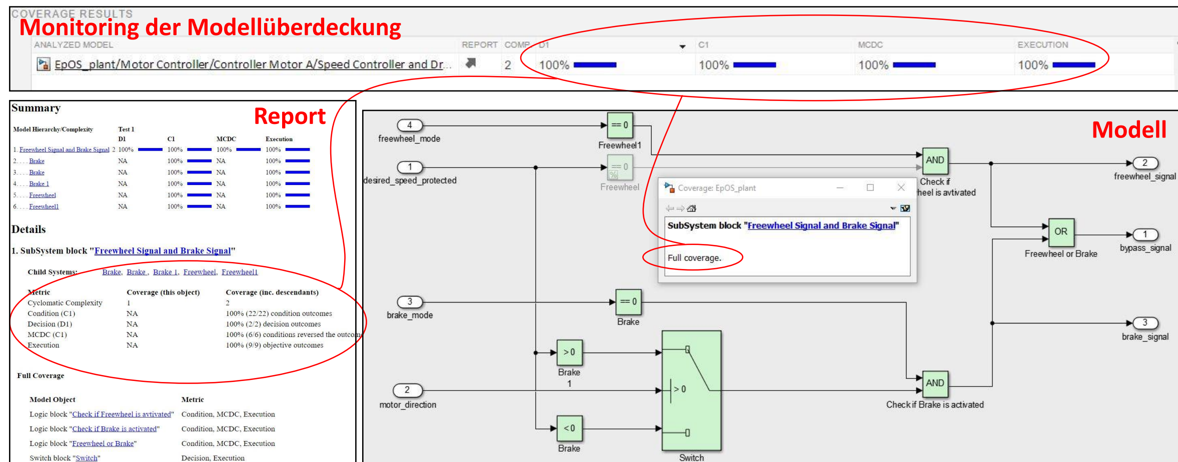


Abbildung 6.4: Modellüberdeckungsanalyse *Freewheel Signal and Brake Signal*

Der Testablauf dieses Whitebox-Tests wurde für das Erreichen einer vollständigen Modellüberdeckung erstellt (100 % *Model Coverage*). Nach der Testdurchführung werden die Ergebnisse der Modellüberdeckungsmessung innerhalb des Tools *Test Manager* der *Simulink Test*-Toolbox angezeigt (siehe Abbildung 6.4 oben). In diesem Ergebnisfenster wird die erreichte Modellüberdeckung der verschiedenen Überdeckungskriterien dargestellt. Über den hinterlegten Link wird eine durch Einfärbung hervorgehobene Darstellung des getesteten Modellteils geöffnet. Gleichzeitig öffnet sich das *Coverage Display Window*, welches Informationen über die aufgezeichneten Überdeckungskriterien der einzelnen Blöcke bereitstellt.

In diesem Fall wird das Subsystem mit einer vollständigen Modellüberdeckung getestet. Dies ist auch an der grünen Einfärbung aller Modellelemente zu erkennen.

Abbildung 6.5 zeigt die Modellüberdeckungsanalyse eines Blackbox-Tests des Subsystems *Max Speed through Speedmode*, bei dem keine vollständige Modellüberdeckung erreicht wird. Es handelt sich dabei um die Berechnung des in Abbildung 5.3 dargestellten Geschwindigkeitsmodus.

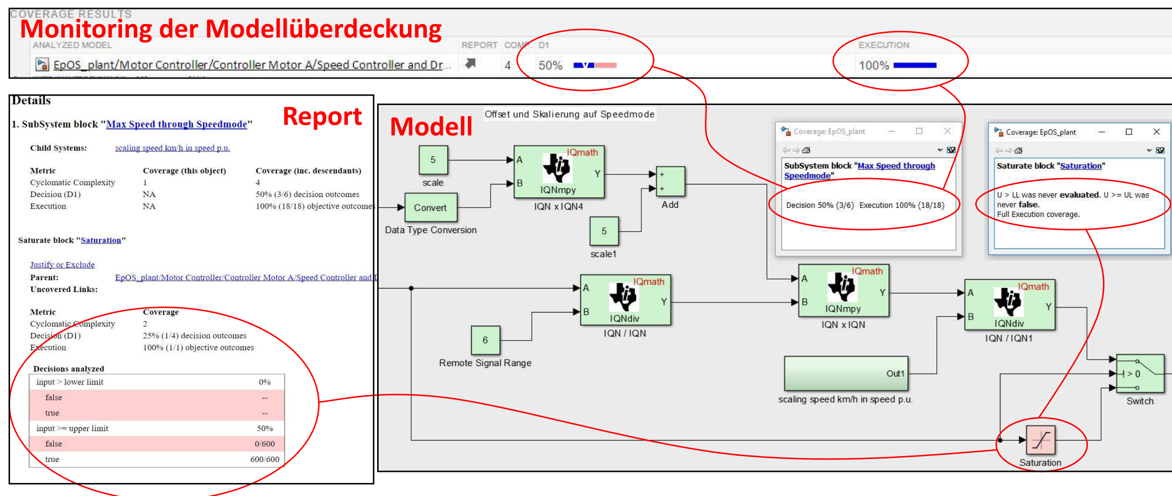


Abbildung 6.5: Modellüberdeckungsanalyse *Max Speed through Speedmode*

Im Testverlauf dieses Blackbox-Tests ist kein negatives Funkfernbedienungssignal n_{remote} vorgesehen. Dadurch erreicht die Condition Converage eine Überdeckung von 50 %. Durch die rote Einfärbung wird der Modellteil gekennzeichnet, welcher mit unvollständiger Modellüberdeckung getestet wird. Das *Coverage Display Window* gibt darüber hinaus Informationen über die fehlenden Signale zum Erreichen einer vollständigen Condition Coverage (siehe Abbildung 6.5 rechts).

Mithilfe des *Simulink Report Generator* wird ein Report erstellt, der die Ergebnisse der Modellüberdeckungsanalyse übersichtlich darstellt, um diese für Analysen sowie Reviews nutzbar zu machen (siehe Anhang A.1.15).

6.2.3 Blackbox-Test

Jede Anforderung an das Implementierungsmodell EpOS (siehe Anhang A.1.9) wird mit Testfällen verknüpft, die in Blackbox-, Whitebox- und Back-to-Back-Tests unterteilt sind.

Gemäß dem ersten Schritt der Teststrategie (vgl. Tabelle 2.1 bzw. Abbildung 6.2) sind folgende Testfälle als Blackbox-Test ausgeführt:

- AccelerateControl_SW_ANF_04_AV5
- AccelerateControl_SW_ANF_04_AV3
- SpeedController_SW_ANF_06_motor_direction_0
- SpeedController_SW_ANF_06_motor_direction_1
- TorqueController_SW_ANF_07
- TractionControl_SW_ANF_08
- Model_Simulation

Im Folgenden wird die Beschreibung des Testfalls *TractionControl_SW_ANF_08* für die Modellkomponente *Traction Control* erläutert. Der Testfall wird mit Hilfe einer Ursache-Wirkungs-Analyse beschrieben, da das Ausgangsverhalten dieses Subsystems von Eingabewertekombinationen abhängt.

Tabelle 6.1 zeigt die Schnittstellensignale der Modellkomponente *Traction Control*, die für die Beschreibung des Testfalls benötigt werden.

Signalname	**Typ**	**Beschreibung**	**Wertebereich**	**Datentyp**
actual_speed	in	Drehzahl	[0...4900]	sfix32_En17
actual_speed_3	in	Referenzdrehzahl	[0...4900]	sfix32_En17
traction_control	in	Traktionskontrolle on/off	[0...1]	uint16
remote_signal	in	Wunschgeschwindigkeit	[-6...6]	sfix32_En17
slip_percentage	out	Schlupfwert	[0...1]	sfix32_En17
slip_signal	out	Schlupfsignal	[0...1]	boolean

Tabelle 6.1: Schnittstellensignale der Modellkomponente *Traction Control*

In Abbildung 6.6 sind die Testdatenverläufe des beschriebenen Testfalls *TractionControl_SW_ANF_08* über den zeitlichen Verlauf dargestellt. Da es der Anspruch ist, die zu evaluierenden Tools in unverfälschter Form darzustellen, sind die Testdatenverläufe, so wie sie vom *Test Manager*-Tool bereitgestellt werden, ohne Achsenbeschriftung ausgeführt.

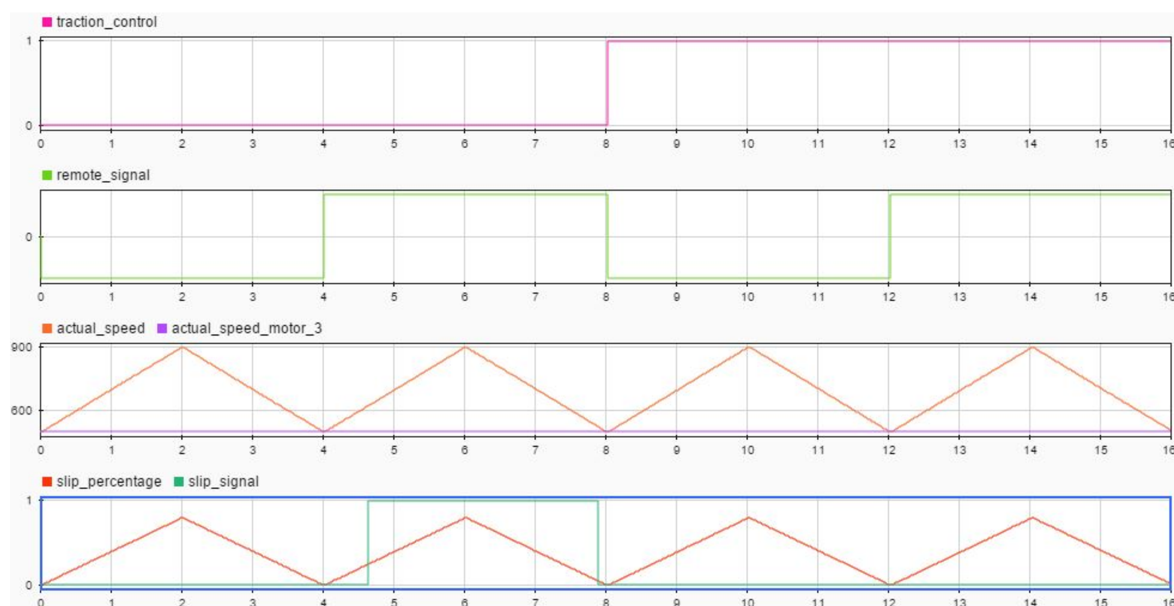


Abbildung 6.6: Testdatenverläufe Blackbox-Test *Traction Control*

Um mit möglichst wenigen Testdaten alle Eingangswertkombinationen zu überprüfen, die ein unterschiedliches Ausgangsverhalten hervorrufen, werden die Wertebereiche der Eingangssignale in Äquivalenzklassen unterteilt.

Für jedes Eingangssignal wird nur ein Wert der Äquivalenzklasse im Testverlauf überprüft. Dabei wird bei einer konstanten Referenzdrehzahl jeweils die Drehzahl *actual_speed* rampenförmig hoch und herunter gesetzt, wodurch ein prozentualer Schlupf zwischen 0 % und 80 % resultiert.

Obwohl die Systemreaktion des Subsystems *Traction Control* erst in der Testdurchführung zu einem späteren Zeitpunkt im Testprozess in Abschnitt 6.5 ermittelt wird, ist diese der Vollständigkeit halber bereits in Form des Ausgangssignals *slip_signal* dargestellt (siehe Abbildung 6.6 unten). Die Testbewertung inklusive Testauswertung erfolgt im weiteren Verlauf dieses Kapitels.

6.2.4 Whitebox-Test

Da es sich bei der Modellkomponente *Freewheel Signal and Brake Signal* um ein sicherheitskritisches Subsystem handelt, wird gemäß drittem Schritt der Teststrategie empfohlen, neben ausreichender Anforderungs- und Wertebereichsüberdeckung durch einen Blackbox-Test, zusätzlich eine ausreichende Modellstrukturüberdeckung durch einen Whitebox-Test zu überprüfen (vgl. Tabelle 2.1 bzw. Abbildung 6.2).

Zur Beschreibung eines Testfalls für eine vollständige Modellstrukturüberdeckung dient die in Abbildung 6.7 dargestellte Modellstruktur der Modellkomponente *Freewheel Signal and Brake Signal* als Testbasis.

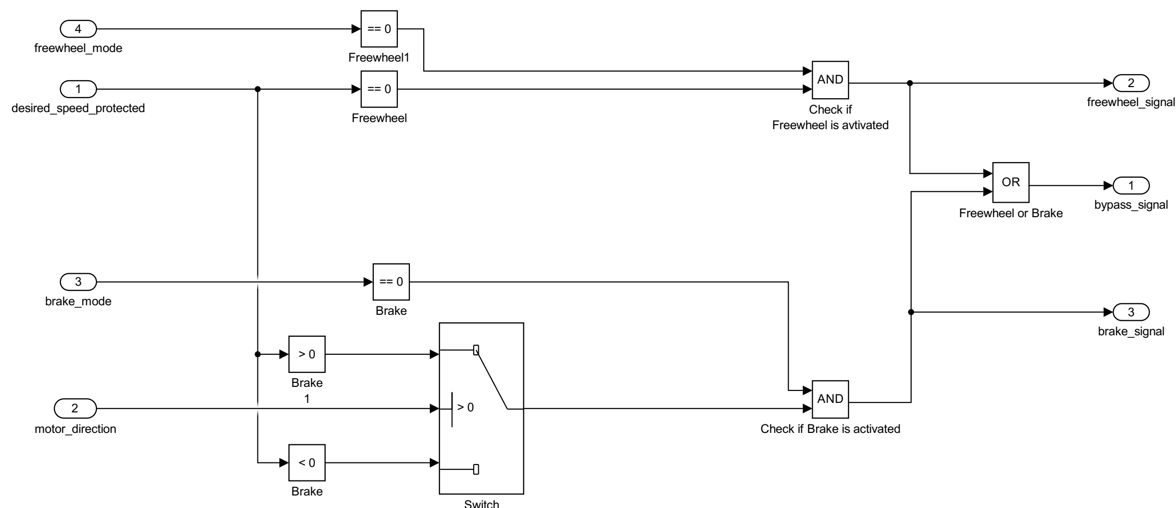


Abbildung 6.7: Modellstruktur *Freewheel Signal and Brake Signal*

Folgende Testfälle sind als Whitebox-Test ausgeführt:

- FreewheelSignal_SW_ANF_01
- BrakeSignal_SW_ANF_02

Beide Testfälle werden so beschrieben, dass eine vollständige Execution Coverage, Condidtion Coverage, Decision Coverage und Modified Condition/Decision Coverage erreicht wird. Dementsprechend sind alle Knoten, Transitionen, sowie alle Bedingungen der Subsystemstruktur durch den Testfall zu prüfen.

Tabelle 6.2 zeigt die Schnittstellensignale der Modellkomponente *Freewheel Signal and Brake Signal*, welche für die Beschreibung der Testfälle benötigt werden.

Signalname	Typ	Beschreibung	Wertebereich	Datentyp
desired_speed	in	Geschwindigkeitsanforderung	[-5...5]	sfix32_En17
motor_direction	in	Drehrichtung ccw/cw	[0...1]	uint16
brake_mode	in	Bremsmodus on/off	[0...1]	uint16
freewheel_mode	in	Freilaufmodus on/off	[0...1]	uint16
brake_signal	out	Bremssignal	[0...1]	boolean
freewheel_signal	out	Freilaufsignal	[0...1]	boolean

Tabelle 6.2: Schnittstellensignale der Modellkomponente *Freewheel Signal and Brake Signal*

In Abbildung 6.8 sind die Testdatenverläufe des beschriebenen Testfalls *FreewheelSignal_SW_ANF_01* dargestellt.

Abbildung 6.8: Testdatenverläufe Whitebox-Test *Freewheel Signal*

Für jede Kombination der Eingangssignale *freewheel_mode* und *motor_direction* wird das Signal *desired_speed* rampenförmig hoch und herunter gesetzt sowie auf Null belassen, um alle möglichen Signalkombinationen von Eingangswerten einer Äquivalenzklasse zu überprüfen. Der Vollständigkeit halber ist zusätzlich die Reaktion des Ausgangssignals *freewheel_signal* dargestellt, das in Kapitel 7 ausgewertet wird.

Analog dazu zeigt Abbildung 6.9 die Testdatenverläufe des beschriebenen Testfalls *BrakeSignal_SW_ANF_02* mit der Reaktion des Ausgangssignals *brake_signal*.

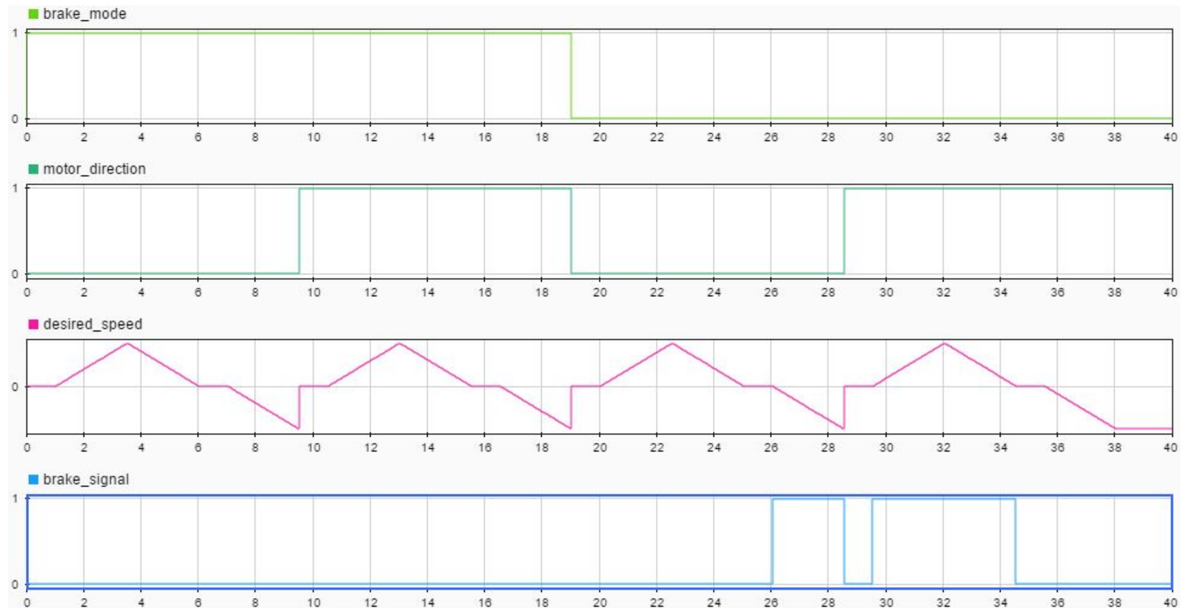


Abbildung 6.9: Testdatenverläufe Whitebox-Test *Brake Signal*

6.2.5 Back-to-Back-Test

Um Modellkomponenten des Motorcontroller-Funktionsmodells zu vergleichen, die in der Funktionalität geändert oder angepasst wurden, werden Back-to-Back-Tests durchgeführt. Entsprechend sind folgende Testfälle als Back-to-Back-Test ausgeführt:

- BrakeValue_SW_ANF_03_Baseline
- AccelerateControl_SW_ANF_04_Equivalence
- Speedmode_SW_ANF_05_Equivalence

Im Folgenden wird die Beschreibung des Back-to-Back-Tests *Speedmode_SW_ANF_05_Equivalence* der Modellkomponente *Max Speed through Speedmode* erläutert.

Um einen qualitativen Vergleich zwischen neuer und alter Modellkomponente zu ziehen, werden beide Subsysteme mit dem gleichen Testverlauf überprüft und deren Systemreaktion gegeneinander aufgetragen.

Tabelle 6.3 zeigt die Schnittstellensignale der Modellkomponente *Max Speed through Speedmode*, welche für die Beschreibung des Testfalls benötigt werden.

Signalname	Typ	Beschreibung	Wertebereich	Datentyp
speed_mode	in	Maximale Geschwindigkeit	[0...5]	uint16
remote_signal	in	Wunschgeschwindigkeit	[-6...6]	sfix32_En17
desired_speed	out	Geschwindigkeitsanforderung	[-5...5]	sfix32_En17

Tabelle 6.3: Schnittstellensignale der Modellkomponente *Max Speed through Speedmode*

Der Testfall wird mithilfe einer Ursache-Wirkungs-Analyse beschrieben. Jede Eingangssignalkombination einer Äquivalenzklasse von *remote_signal* und *speed_mode* wird überprüft, um die resultierenden Ausgangssignale *desired_speed* aufzuzeichnen.

In Abbildung 6.10 sind die Testdatenverläufe des beschriebenen Testfalls *Speedmode_SW_ANF_05_Equivalence* dargestellt.

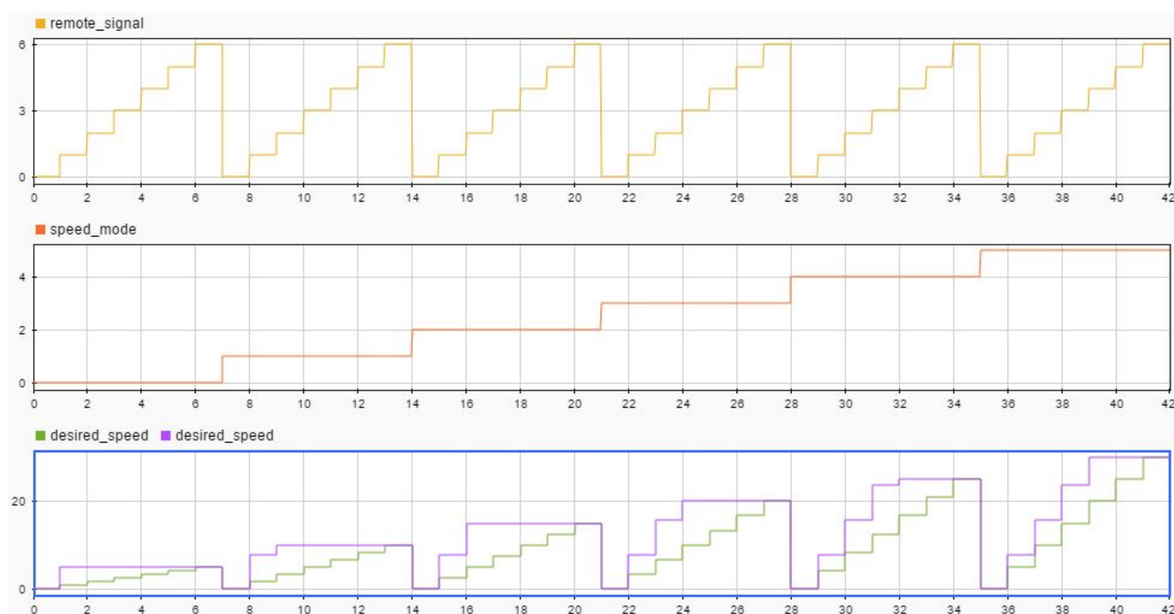


Abbildung 6.10: Testdatenverläufe Back-to-Back-Test *Max Speed through Speedmode*

An der Systemreaktion ist zu erkennen, dass das Ausgangssignal *desired_speed* der neuen Funktion (vgl. Abbildung 6.10 grün) im Vergleich zur alten Funktion (vgl. Abbildung 6.10 violett) abhängig vom Eingangssignal *speed_mode* skaliert wird, wie in Abschnitt 5.3.3 beschrieben. Dadurch kann der gesamte Wertbereich des Funkfernbedienungssignals unabhängig vom eingestellten Geschwindigkeitsmodus ausgenutzt und somit die Geschwindigkeitsanforderung genauer dosiert werden.

6.3 Testdatenerstellung

Die in den vorherigen Abschnitten beschriebenen Testverläufe müssen mithilfe bestimmter Anregungsblöcke als Testdaten zur Verfügung gestellt werden, um die spätere Testsimulationen mit der *Simulink Test*-Toolbox durchführen zu können. Hierfür werden für die Testfälle des Implementierungsmodells EpOS folgende Möglichkeiten genutzt:

- *Test Sequence*
- *Signal Builder*
- freie *Inports*

Die Testdaten für die Modellkomponente *Traction Control* werden im sogenannten *Test Sequence*-Block der *Simulink Test*-Toolbox modelliert, wie in Abbildung 6.11 dargestellt.

Die bereits in Abbildung 6.6 dargestellten Testdatenverläufe werden in der *Stateflow*-Beschreibungssprache nachgebildet. Durch die Formulierung von Übergangsbedingungen wird zwischen Stufen und Unterstufen gewechselt, um den spezifischen Testverlauf zu modellieren. Die Abtastrate der Testdaten wird wie die spätere Zykluszeit auf dem Motorcontrollermodul mit 0,01 Sekunden definiert.

TractionControl_SW_ANF_08/Test Sequence * - Test Sequence Editor

Symbols: Input: 1. slip_signal; Output: 1. actual_iq, 2. actual_speed, 3. actual_speed_motor_, 4. actual_speed_motor_, 5. actual_speed_motor_, 6. traction_control, 7. remote_signal; Local: EndTest; Constant; Parameter; Data Store Memory

Testdatenerstellung

Step	Transition	Next Step	Description
InitializeTest actual_iq(:) = 0; % fixed-point actual_speed(:) = 500; % Actual Speed Motor A = 500 rpm actual_speed_motor_3(:) = 500; % Actual Speed Motor D = 500 rpm actual_speed_motor_1(:) = 0; % fixed-point actual_speed_motor_2(:) = 0; % fixed-point traction_control = uint16(0); remote_signal(:) = 0; % fixed-point	1. true	TractionControl_0	
TractionControl_0 traction_control = uint16(0); % Traction Control enabled	1. EndTest == 1	TractionControl_1	
RemoteSignalNegative remote_signal(:) = -1; % Remote Signal negative EndTest = 0;	1. EndTest == 2	RemoteSignalPositive	
RAMP_UP_actual_speed actual_speed(:) = latch(actual_speed) + ramp(200*et()); % fixed-point	1. after(2, sec)	RAMP_DOWN_actual_speed	
RAMP_DOWN_actual_speed actual_speed(:) = latch(actual_speed) - ramp(200*et()); % fixed-point	1. after(2, sec)	CONSTANT_actual_speed	
CONSTANT_actual_speed actual_speed(:) = latch(actual_speed); % fixed-point EndTest = 2;			
RemoteSignalPositive remote_signal(:) = 1; % Remote Signal positive EndTest = 0;			
TractionControl_1 traction_control = uint16(1); % Traction Control disabled	1. EndTest == 1	StopTest	
StopTest			

Abbildung 6.11: Testdatenerstellung über *Test Sequence*

Die Testdaten für die Modellkomponente *Max Speed through Speedmode* werden über einen *Signal Builder*-Block zur Verfügung gestellt. Abbildung 6.12 zeigt die Modellierung des Eingangssignalverlaufs *remote_signal*.

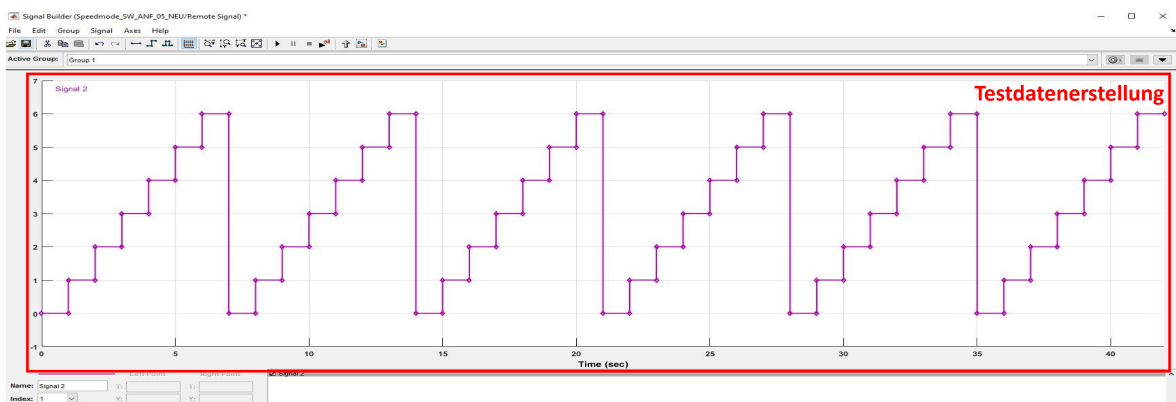


Abbildung 6.12: Testdatenerstellung über *Signal Builder*

6.4 Testrahmenerstellung

Die Erstellung eines Testrahmens (*Test Harness*) ist eine notwendige Voraussetzung für die spätere Testdurchführung der erstellten Testfälle. Mit dem Tool *Test Harness* der *Simulink Test*-Toolbox wird ein Testrahmen eines beliebigen Subsystems oder des gesamten Implementierungsmodells EpOS erstellt.

In Abbildung 6.13 ist die Testrahmenerstellung mit Simulink am Beispiel des Subsystems *Traction Control* dargestellt.

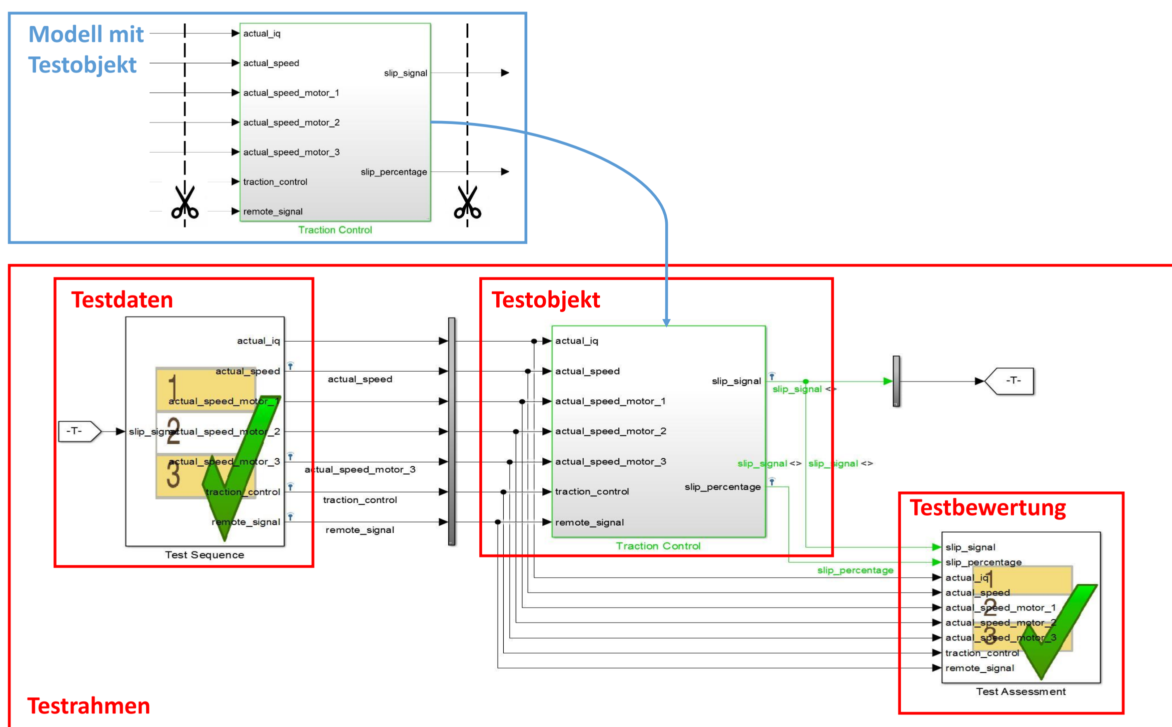


Abbildung 6.13: Testrahmenerstellung der Modellkomponente *Traction Control*

Das Tool *Test Harness* isoliert die zu testenden Modellkomponente *Traction Control* und ergänzt sie um Anregungsblöcke zum Bereitstellen der Testdaten sowie Ausgabeblöcke zur Aufzeichnung und Bewertung der Testergebnisse innerhalb einer vom Hauptmodell separaten Simulationsumgebung. Der Testrahmen wird mit der Modellkomponente *Traction Control* verknüpft und innerhalb des Hauptmodells verwaltet.

Die Anregungsblöcke können in verschiedenen Formen erzeugt werden. Neben freien *Inports* können *Signal Builder*-, *From Workspace*-, *From File*- oder *Test Sequence*-Blöcke

zur Bereitstellung der Testdaten genutzt werden. Im Fall des Testobjekts *Traction Control* werden die Testdaten des entsprechenden Testfalls mittels *Test Sequence*-Block vorgegeben (vgl. Abschnitt 6.3).

Als Ausgangsblöcke können freie *Outports*-, *Scopes*-, *To Workspace*- oder *To File*- gewählt werden.

Zusätzlich besteht die Möglichkeit, einen *Test Assessment*-Block zu erzeugen, mit dem es möglich ist, Testbewertungskriterien für die spätere Testdurchführung bzw. Testauswertung, wie in Abbildung 6.14 dargestellt, zu definieren.

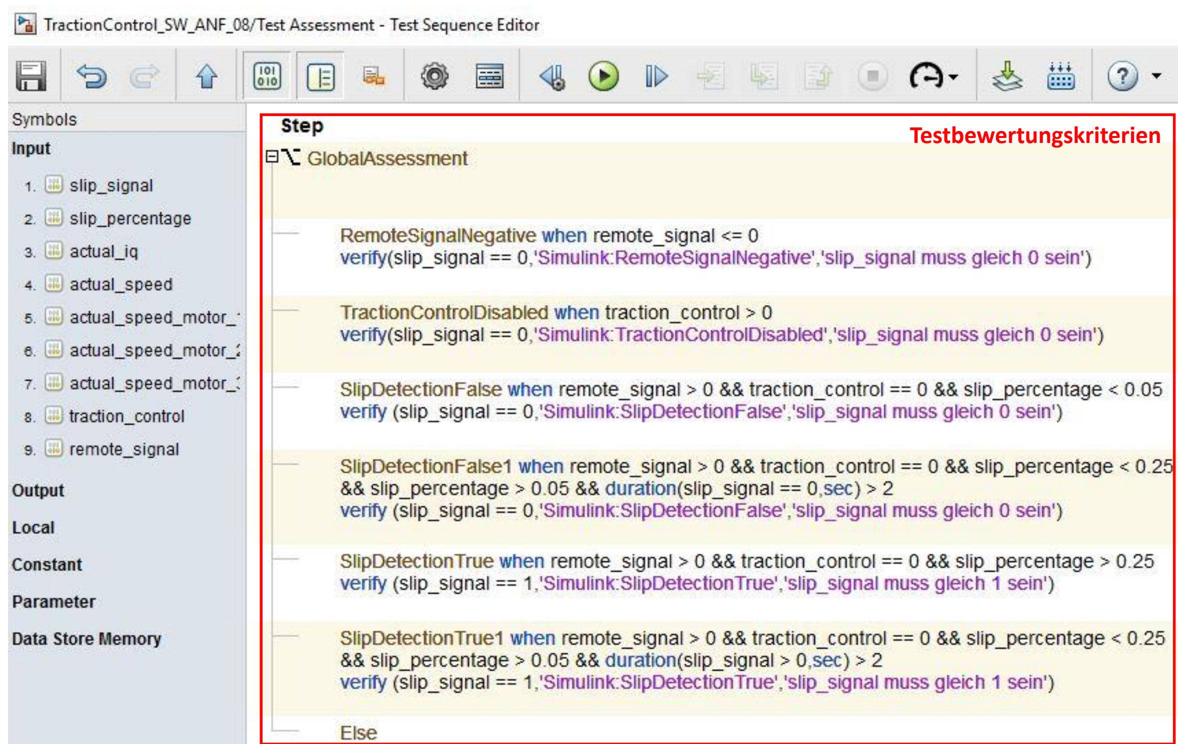


Abbildung 6.14: Testbewertung der Modellkomponente *Traction Control*

Durch die Definition sogenannter *Verify Statements* im *Test Assessment*-Block kann eine automatisierte Testauswertung im Zuge der späteren Testdurchführung erfolgen. Ein *Verify Statement* besteht aus einem zu überprüfenden logischen Ausdruck, einer Bezeichnung zur Nachverfolgung des Bewertungskriteriums im Zuge der Testauswertung, sowie einer Fehlermeldung.

Im *Test Assessment*-Block des Testrahmens zum Testobjekt *Traction Control* (siehe Abbildung 6.14) wird überprüft, ob sich das Ausgangssignal *slip* abhängig von der Eingangssignalkombination den Anforderungen entsprechend verhält. Der Testfall wird als bestanden bewertet, wenn das *slip*-Signal *true* ist, sobald bei aktivierter Traktionskontrolle und positiver Geschwindigkeitsanforderung ein Schlupf größer 25 % vorliegt und erst wieder *false* wird, wenn der Schlupf einen Wert kleiner 5 % erreicht (vgl. Abschnitt 5.3.4).

Um die Eingangs- und Ausgangssignale der Testrahmen-Simulationsumgebung in der späteren Testauswertung nutzen zu können, müssen diese im *Base Workspace* aufgezeichnet werden.

6.5 Testdurchführung

Die Testdurchführung erfolgt mit dem Tool *Test Manager* der *Simulink Test*-Toolbox, deren Bedienoberfläche in Abbildung 6.15 dargestellt ist.

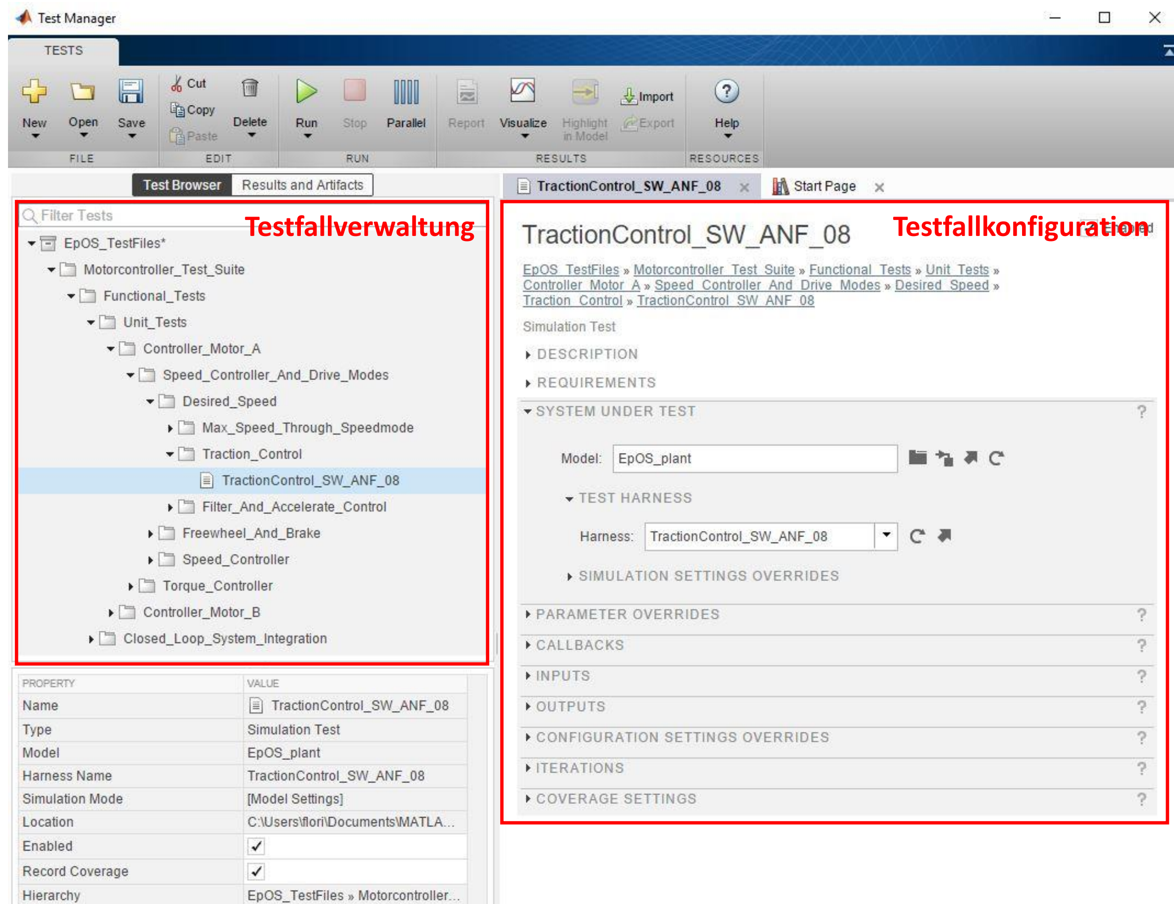


Abbildung 6.15: Testdurchführung mit dem *Test Manager*

Im *Test Manager* werden die verschiedenen Testrahmen zu Testfällen organisiert und automatisiert durchgeführt. Neben der Testfallverwaltung, der Testfallkonfiguration und der Testdurchführung wird der *Test Manager* für die Testauswertung und Testdokumentation genutzt, die in Abschnitt 6.6 beschrieben werden.

Es wird ein *Test File* erstellt (siehe Anhang A.1.12), das außerhalb des Implementierungsmodells EpOS als Datendatei gespeichert wird. Dieses *EpOS_ TestFile* enthält mehrere *Test Suites* für Komponententests und Integrationstests. Eine *Test Suite* beinhaltet je nach zu testender Modellkomponente einen oder mehrere Testfälle (*Test Case*).

Für die Erstellung eines Testfalls wird aus folgenden Bereichen gewählt:

- *Simulation Test*
- *Baseline Test*
- *Equivalence Test*
- *Test for Subsystem*

Je nach gewählter Art erfolgt eine spezifische Testfallkonfiguration (siehe Abbildung 6.15 rechts). Die in Abschnitt 6.2.3 und 6.2.4 aufgeführten Blackbox- bzw. Whitebox-Tests werden als *Simulation Test* oder *Test for Subsystem* angelegt.

Bei einem *Simulation Test* werden die in Abschnitt 6.4 entwickelten Testrahmen in den Testfall geladen. Dadurch verfügt der Testfall bereits über Testdaten und Testbewertungskriterien wie bspw. beim Testrahmen des Subsystems *Traction Control* (siehe Abbildung 6.13).

Ein *Test for Subsystem* erzeugt erst bei dessen Erstellung ein Testrahmen der jeweiligen Modellkomponente. Der erzeugte Testrahmen verfügt über freie *Inports* als Anregungselemente. Während der Testfallerstellung läuft eine closed-loop Simulation des Motorcontroller-Funktionsmodells mit angeschlossenem Streckenmodell, wodurch bestimmte Signalverläufe aufgezeichnet werden. Die aufgezeichneten Signalverläufe werden auf die freien Inports des erzeugten Testrahmens gemappt. Dadurch ist es möglich, Modellkomponenten, deren Verhalten vom Feedback der Regelstrecke abhängt, automatisiert mit realistischen Eingangsdaten zu versehen. Testbewertungskriterien können nachträglich über die Verwendung des *Test Assessment*-Blocks hinzugefügt werden. Diese Art von Testfall wird bspw. beim Subsystem *Speed Controller* verwendet.

Die in Abschnitt 6.2.5 beschriebenen Back-to-Back-Tests werden als *Baseline Test* oder *Equivalence Test* angelegt. Bei einem *Equivalence Test* wird ein Testfall erstellt, bei dem zwei verschiedene Testrahmen mit gleichen Stimuli ausgeführt werden. Dadurch werden die Ausgangssignale zweier Lösungsfunktionen zur Umsetzung einer Anforderung miteinander verglichen wie beim Subsystem *Max Speed through Speedmode*.

Der einzige Unterschied eines *Baseline Tests* ist der Vergleich von den Ausgangssignalen eines Testrahmens zu einer vorher definierten Baseline. Dadurch kann ein gefordertes Verhalten einer Modellkomponente im Vorfeld durch einen Signalverlauf definiert und mit dem Ausgangsverhalten einer Lösungsfunktion verglichen werden wie im Subsystem *BrakeValue*.

Alternativ kann die Testdurchführung einzelner Testfälle auch direkt innerhalb der Simulationsumgebung des erstellten Testrahmens erfolgen. Diese Art der Testdurchführung ist dazu geeignet, bei kleinen Änderungen der Modellkomponente oder bei der Erstellung oder Änderung der Testdatenbereitstellung oder der Testbewertungskriterien im *Test Sequence*-Block bzw. im *Test Assessment*-Block auf schnelle Weise Fehlerzustände oder Fehlerwirkungen aufzudecken. Die Visualisierung der Systemreaktion sowie die Auswertung der *Verify Statements* läuft in diesem Fall über das *Simulation Data Inspector*-Tool.

6.6 Testauswertung und -dokumentation

Die Testauswertung erfolgt genau wie die Testdurchführung mit dem Tool *Test Manager* der *Simulink Test*-Toolbox. Die in der Testdurchführung ausgeführten Testfälle der *EpOS_TestFile* werden unter dem Reiter *Results and Artifacts* gelistet, wie in Abbildung 6.16 dargestellt.

Abbildung 6.16: Gesamtübersicht der Testauswertung im *Test Manager*

Über eine Statusanzeige wird die Anzahl der durchgeführten Testfälle angezeigt. Dabei sind die als bestanden bewerteten Testfälle grün und die als durchgefallen bewerteten Testfälle rot gekennzeichnet (siehe Abbildung 6.16 links). Außerdem wird zu jedem Testfall die gemessene Modellüberdeckung dargestellt, die für die in Abschnitt 6.2.2 beschriebene Modellüberdeckungsanalyse genutzt wird (siehe Abbildung 6.16 rechts).

Die Auswahl eines einzelnen Testfalls führt zum jeweiligen Testergebnis mit Testauswertung innerhalb des *Test Manager*-Tools. In Abbildung 6.17 ist die detaillierte Testauswertung des Testfalls *TractionControl_SW_ANF_08* zum Testobjekt *Traction Control* dargestellt.

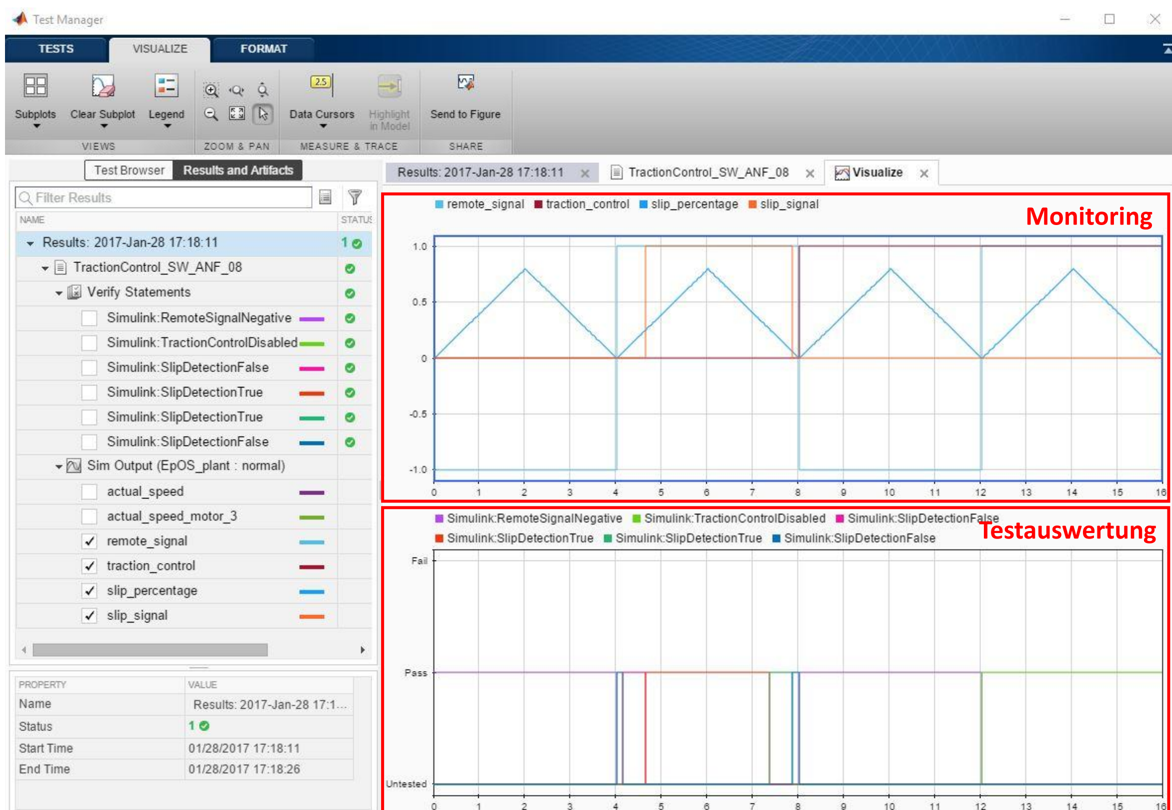


Abbildung 6.17: Testauswertung bestandener Testfall im *Test Manager*

Die während der Simulation des Testrahmens aufgezeichneten Testdaten und das Ausgangsverhalten des Testobjekts können separat oder im Verbund dargestellt werden (siehe Abbildung 6.17 oben). Falls für den Testfall ein Testorakel vorhanden ist, erfolgt außerdem eine automatische Testauswertung (siehe Abbildung 6.17 unten).

Für den Testfall *TractionControl_SW_ANF_08* und alle anderen als *Simulation Tests* angelegten Testfälle dienen die im *Test Assessment*-Block des Testrahmens definierten Bewertungskriterien (*Verify Statements*) als Testorakel. Die während der Testdurchführung überprüften Bewertungskriterien werden in *Pass*, *Fail* und *Untested* klassifiziert.

Das Testorakel der als *Baseline Test* oder *Equivalence Test* angelegten Testfälle ist die definierte Baseline bzw. die Systemreaktion des zu vergleichenden Testobjekts. Hierbei ist das Bewertungskriterium eine äquivalente Systemreaktion bzw. das Einhalten von im Vorfeld definierten Toleranzen, falls eine Abweichung gemessen wird.

In der Testauswertung des Testfalls *TractionControl_SW_ANF_08* ist zu erkennen, dass die Bewertungskriterien den kompletten Testablauf abdecken (siehe Abbildung 6.17 unten). Über die gesamte Simulationszeit ist mindestens ein *Verify Statement* als bestanden eingestuft. Keines der Bewertungskriterien wird während des Testdurchlaufs als durchgefallen oder ungetestet bewertet, so dass der gesamte Testfall als bestanden bewertet und mit dem Status grün gekennzeichnet wird.

Die Änderung einer einzigen Abfrage im Algorithmus des Testobjekts *Traction Control* bewirkt, dass ein *Verify Statement* während des Testdurchlaufs ungetestet bleibt und ein zweites *Verify Statement* nicht erfüllt werden kann, wie in Abbildung 6.18 dargestellt.

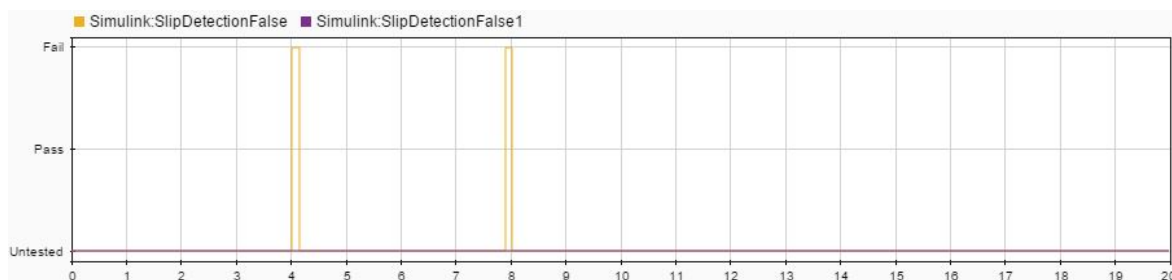


Abbildung 6.18: Testauswertung nicht bestandener Testfall im *Test Manager*

Dadurch wird der gesamte Testfall als nicht bestanden bewertet und mit dem Status rot gekennzeichnet.

Alle Testergebnisse können exportiert und außerhalb des Implementierungsmodells gespeichert werden (siehe Anhang A.1.13). Für Analysen und Reviews können gespeicherte Testergebnisse in den *Test Manager* importiert werden.

Mithilfe des *Simulink Report Generators* wird außerdem die Testdokumentation *EpOS_ Test_ Results_ Report* erstellt, in der die Auswertung aller durchgeführten Testfälle dokumentiert ist (siehe Anhang A.1.14).

6.7 Anforderungen und Beschränkung der Toolkette

Seitens der Simulink-Toolkette gibt es einige Anforderungen und Limitierungen, die es zu beachten gilt, um eine erfolgreiche Umsetzung einer durchgängigen modellbasierten Entwicklung inklusive modellbasiertem Test in ein Entwicklungsprojekt wie diesem zu integrieren.

6.7.1 Beschränkungen für die Modellstruktur

Als größtes Problem über den gesamten modellbasierten Entwicklungs- und Testprozess zeigt sich die Verwendung getriggerter Modellkomponenten und Subsysteme (*Function-Call Trigger*), da diese aktuell von vielen Tools nicht unterstützt werden.

Bedingt durch den Algorithmus des Implementierungsmodells EpOS wird als Triggerquelle ein *Hardware Interrupt*-Block aus der Library des *TI C2000 Support for Embedded Coder* benötigt (siehe Modellstruktur in Abbildung 5.1). Die Modellkomponente *FOC - Torque Controller* sowie die hardwarespezifischen Library-Blöcke der ADC-, eCAP- und ePWM-Module müssen von einem ADC-Interrupt getriggert werden, um die Funktionalität des FOC-Algorithmus sicherzustellen.

Durch die *Function-Call* getriggerten Signale gibt es starke Beschränkungen im Aufbau einer Modellstruktur (vgl. Abschnitt 5.1), die eine Möglichkeit bietet, sowohl hardwarespezifischen C-Code zu generieren, als auch Simulationen mit angeschlossenem Streckenmodell durchzuführen. Folgende Beschränkungen für die entwickelte Modellstruktur sind hervorzuheben:

- Ohne einen Workaround durch zusätzliche *Function-Call Generatoren* ist es nicht möglich, das vom *Hardware Interrupt*-Block getriggerte Signal mittels *Function-Call Split*-Block für mehrere Subsysteme aufzuteilen. Dies ist jedoch notwendig, um das Motorcontroller-Funktionsmodell ohne hardwarespezifische Library-Blöcke zu modellieren, um ein simulierbares Ergebnis erreichen zu können.

- Ein *Function-Call* kann kein *Atomic Subsystem* triggern, da sich alle Subsysteme, die von einem *Function-Call* getriggert werden, auf der gleichen Ebene der Modellhierarchie befinden müssen.

- Ein *Function-Call* kann kein *Variant Subsystem* triggern, welches zusätzlich über Eingänge verfügt, die von einer anderen Quelle getriggert werden. Dadurch ist es nicht möglich, das *Variant Management*-Tool zum Steuern der Modell-Varianten *Simulation* und *Hardware* zu verwenden.

- Ein *Function-Call* kann kein *Referenced Subsystem* triggern, in dem mehrere Instanzen referenzierter Modelle genutzt werden. Dadurch ist es nicht möglich, referenzierte Subsysteme zum Steuern der Modell-Varianten *Simulation* und *Hardware* zu verwenden.

- Subsysteme, die von einem *Function-Call* getriggert werden, dürfen über keine *Bus-Signale* verfügen.

Die Firma MathWorks ist sich dieser Problematik bewusst, kann jedoch mit dem aktuellen Stand der Toolkette keine andere Funktionalität zur Verfügung stellen. Heutiger Stand der Technik in Industrieprojekten ist es, das Funktionsmodell mit allen *Function-Call*-Triggern innerhalb einer hierarchischen Ebene zu modellieren. Dadurch kann mit referenzierten Subsystemen für Simulations- und Hardware-Varianten gearbeitet werden. Die Referenz der Modell-Variante *Simulation* wird dabei gänzlich ohne *Function-Call*-Trigger modelliert. [26]

In modellbasierten Softwareprojekten in der Industrie wird aktuell überwiegend mit einer Integration von modellbasiert entwickeltem Code und handgeschriebenem Code gearbeitet wird. Gänzlich modellbasiert entwickelte Systeme sind aufgrund der aktuell noch vorhandenen Beschränkungen und Limitierungen selten zu finden. [26]

6.7.2 Limitierungen und Fehler im modellbasierten Testprozess

Neben den aufgetretenen Schwierigkeiten während der Entwicklung einer geeigneten Modellstruktur werden in der Projektlaufzeit auch einige Limitierungen und vereinzelte Fehler im modellbasierten Testprozess mit den Toolboxen *Simulink Verification and Validation* und *Simulink Test* sichtbar. Die teilweise fehlerhafte Software sowie die aufgefallenen Einschränkungen in der Benutzerfreundlichkeit der Toolboxen sind

an die Firma MathWorks übermittelt worden und im aktuellsten Release 2016b bereits teilweise behoben bzw. verbessert.

Folgende Limitierungen schränken den modellbasierten Test ein und erfordern entsprechende Workarounds:

- Die Erstellung eines *Test for Subsystem* im *Test Manager*-Tool zur automatischen Testdatenerstellung aus dem Feedback des angeschlossenen Streckenmodells wird nicht für *Function-Call* getriggerte Subsysteme unterstützt. Die Modellkomponente *FOC - Torque Controller*, die aufgrund der Eingangssignale Phasenströme i_a und i_b sowie Polradwinkel ϑ zur Erstellung sinnvoller Testfälle das Feedback des PMSM-Modells benötigt, wird folglich nicht durch die automatische Testfallerstellung unterstützt. Ein Workaround hierfür ist die Modellierung des Streckenmodells innerhalb des Testrahmens der Modellkomponente, wie in Abbildung 6.19 dargestellt.

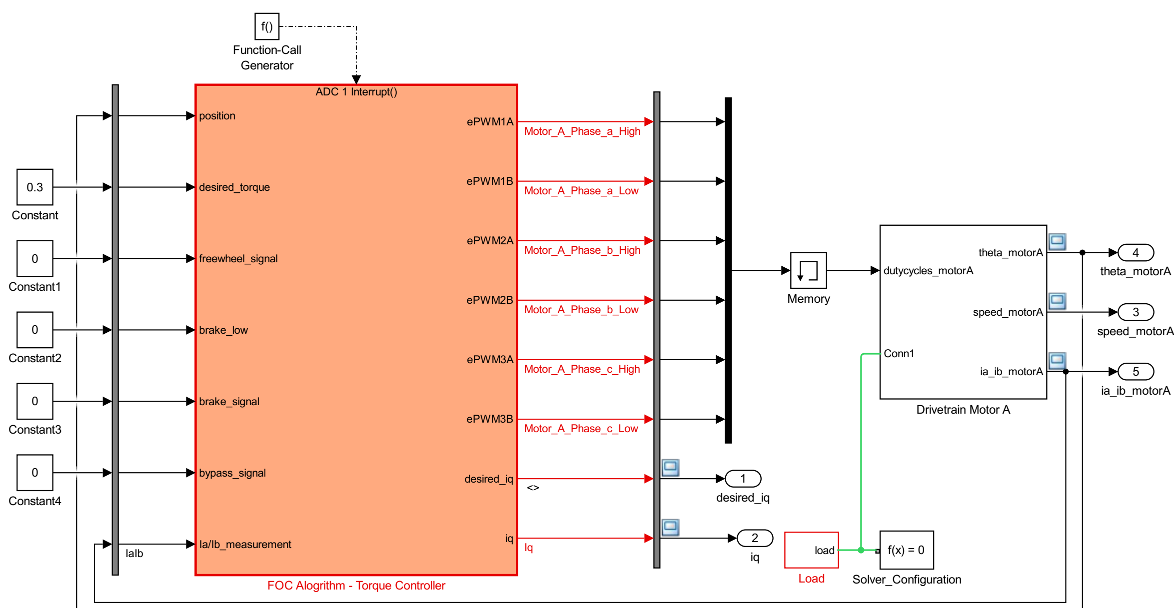


Abbildung 6.19: Testrahmen *FOC - Torque Controller* mit Streckenmodell

Dadurch können die Eingangssignale i_a, i_b und ϑ für den Testfall aus dem Feedback des PMSM-Modells bereitgestellt werden.

- Die Erstellung von *Verify Statements* zur automatischen Testbewertung im *Test Manager*-Tool (vgl. Abschnitt 6.6) ist nur für Ausgangssignale des Testrahmens

möglich. Interne Signale des Subsystems können nicht als Bewertungskriterium herangezogen werden. Aus diesem Grund müssen alle zur Testbewertung benötigten Signale aus dem Subsystem der jeweiligen Modellkomponente geführt werden (vgl. Signal *slip_percentage* in Abbildung 6.13). Laut MathWorks ist es aber bereits im aktuellen Release 2016b möglich *Verify Statements* tiefer liegender Signale über den Reiter *Custom Objectives* zu definieren [26].

- Bei der Erstellung eines *Test for Subsystem* im *Test Manager* zur automatischen Testdatenerstellung gibt es mehrere Möglichkeiten, externe Daten auf die freien *Inports* des Testrahmens zu mappen. In der Voreinstellung werden die aus dem Feedback der Simulation aufgenommenen Signale nach *Port Order* gemappt. Um jedoch einzelne aus dem Feedback aufgezeichnete Signale nachträglich zu manipulieren und dem Testrahmen bspw. als Konstanten bereitzustellen, besteht die Möglichkeit, die Testdaten bezüglich des *Port Name* oder des *Signal Name* zu mappen. Hierbei ist es jedoch erforderlich, die Signalnamen im Testrahmen entsprechend der kryptischen Variablenbezeichnungen in der automatisch angelegten *Data_Input_File* zu benennen. Aus Sicht des Benutzers wäre es jedoch sinnvoller, die Variablennamen in der *Data_Input_File* ändern zu können. Laut MathWorks ist es im aktuellen Release 2016b bereits möglich, von Beginn an über den Reiter *Custom Objectives* nur bestimmte Eingangssignale aus dem Feedback der Simulation aufzuzeichnen und die anderen Testdaten manuell zu setzen [26]. Dadurch wird diese Problematik umgangen.

- Bei der Erstellung eines *Test for Subsystem* im *Test Manager* zur automatischen Testdatenerstellung wird im erzeugten Testrahmen kein *Test Assessment*-Block zur Definition von Testbewertungskriterien für die automatische Testbewertung angelegt (vgl. Abschnitt 6.4). Dieser muss nachträglich innerhalb der Simulationsumgebung des Testrahmens hinzugefügt werden.

- Die Abtastraten für *Test Sequence*-Block und *Test Assesment*-Block des *Test Harness*-Tools werden einmalig beim Erstellen des Testrahmens über das Hauptmodell generiert. Dies liegt daran, dass *Stateflow*-Komponenten, anders als *Simulink*-Komponenten, bei der Erstellung direkt in C-Code kompiliert werden und sich dann als statische Zustandsautomaten verhalten [26]. Bei einer Änderung der Abtastrate im Hauptmodell ist daher ein Workaround erforderlich, indem ein neuer Build des Testrahmens aus dem Hauptmodell erfolgt.

Diese Limitierungen schränken die Benutzerfreundlichkeit zwar ein, können aber durch Workarounds umgangen werden. Es fallen jedoch auch einige Fehler in der Software auf, die eine Benutzbarkeit unmöglich machen:

- Beim Verlinken von Modellanforderungen und jeweiligen Testfällen mit dem Tool *Requirements Traceability* der *Simulink Verification and Validation*-Toolbox besteht ein Fehler in der bidirektionen Verlinkung. Es ist zwar möglich, über den gesetzten Link von Testfall zum Anforderungsdokument zu gelangen, nicht jedoch umgekehrt. Entsprechend resultiert bei Betätigung des gesetzten Links im Anforderungsdokument eine Fehlermeldung.

- Bei der Erstellung eines *Test for Subsystem* im *Test Manager* zur automatischen Testdatenerstellung kann keine Bezeichnung und kein Speicherort für die zu erstellende *Data_Input_File* angelegt werden, obwohl der Anwender dazu aufgefordert wird. Ansonsten wird das Mappen der Feedbacksignale auf die freien *Inports* nicht ausgeführt und es resultiert eine Fehlermeldung. Das entsprechende Feld muss frei bleiben, sodass die erzeugte *Data_Input_File* mit einer voreingestellten Bezeichnung an einem voreingestellten Speicherort angelegt wird. Nur auf diese Weise kann ein *Test for Subsystem* erfolgreich erstellt werden.

Aufgrund der umgesetzten Workarounds kann das Projekt trotz der aufgeführten Limitierungen und Fehler mit einem durchgängigen modellbasierten Entwicklungsprozess inklusive modellbasiertem Test erfolgreich durchgeführt werden. Dies wird im folgenden Kapitel gezeigt.

7 Verifikation

Die Verifikation des Traktionswechselrichters erfolgt, der Teststrategie des modellbasierten Tests entsprechend (vgl. Abbildung 2.7), auf mehreren Entwicklungsstufen.

Zuerst wird das Implementierungsmodell EpOS mit den in dieser Arbeit vorgestellten und entwickelten Testaktivitäten anhand des Verifikationsplans der Modellanforderungen verifiziert (MIL-Test).

Anschließend wird aus dem verifizierten Motorcontroller-Funktionsmodell hardwarespezifischer Code für die Zielplattform des C2000 F28069 Mikrocontroller generiert, der auf der Hardware des Motorcontroller EpOS in Betrieb genommen wird. Der Systemintegrationstest (HIL-Test) erfolgt anhand des gleichen Verifikationsplans. Die Testergebnisse des HIL-Tests werden mit den Ergebnissen des MIL-Tests verglichen (Back-to-Back-Test Ansatz). Dadurch kann ermittelt werden, welche Anforderungen bereits weitestgehend oder vollständig auf Modellebene verifiziert werden können.

Abschließend erfolgt die Verifikation des Gesamtsystems E-Mountainboard anhand des Verifikationsplans des Lastenhefts (siehe Anhang A.1.2).

7.1 Verifikation Motorcontroller-Funktionsmodell

Die Verifikation des Motorcontroller-Funktionsmodells erfolgt in verschiedenen Teststufen. Die einzelnen Subsysteme werden zunächst gekapselt getestet. Danach wird das Systemverhalten des Gesamtmodells mit den entwickelten Integrationstests verifiziert.

7.1.1 Komponententest

Im Komponententest werden folgende Subsysteme des Motorcontroller-Funktionsmodells mit den in Kapitel 6 entwickelten modellbasierten Tests anhand des Verifikationsplans der Modellanforderungen (siehe Anhang A.1.3) verifiziert:

- Max Speed through Speedmode
- Traction Control
- Accelerate Control
- Freewheel Signal and Brake Signal
- Torque Controller
- Speed Controller

Die Verifikation der Subsysteme stellt sicher, dass alle formulierten Modellanforderungen in einzelnen Komponenten implementiert und alle durch den modellbasierten Test aufgedeckten Fehlerwirkungen behoben worden sind.

In der Testauswertung des jeweiligen Testfalls im *Test Manager* müssen alle Bewertungskriterien als bestanden eingestuft sein und den gesamten Testablauf abdecken (vgl. Abschnitt 6.6), damit die Komponente als verifiziert eingestuft wird.

Im Folgenden werden die Ergebnisse und Bewertungskriterien der einzelnen Testfälle jeweils grafisch dargestellt und erläutert.

Max Speed through Speedmode: Der Komponententest *Speedmode_ SW_ ANF_ 05 _ Equivalence* der Modellkomponente *Max Speed through Speedmode* ist als Back-to-Back Test ausgeführt. Die Testdatenverläufe sowie die Systemreaktion wurden in Abschnitt 6.2.5 dargestellt. Abbildung 7.1 zeigt die Systemreaktion der neuen und alten Funktion in grün bzw. violett sowie deren Abweichung zueinander.

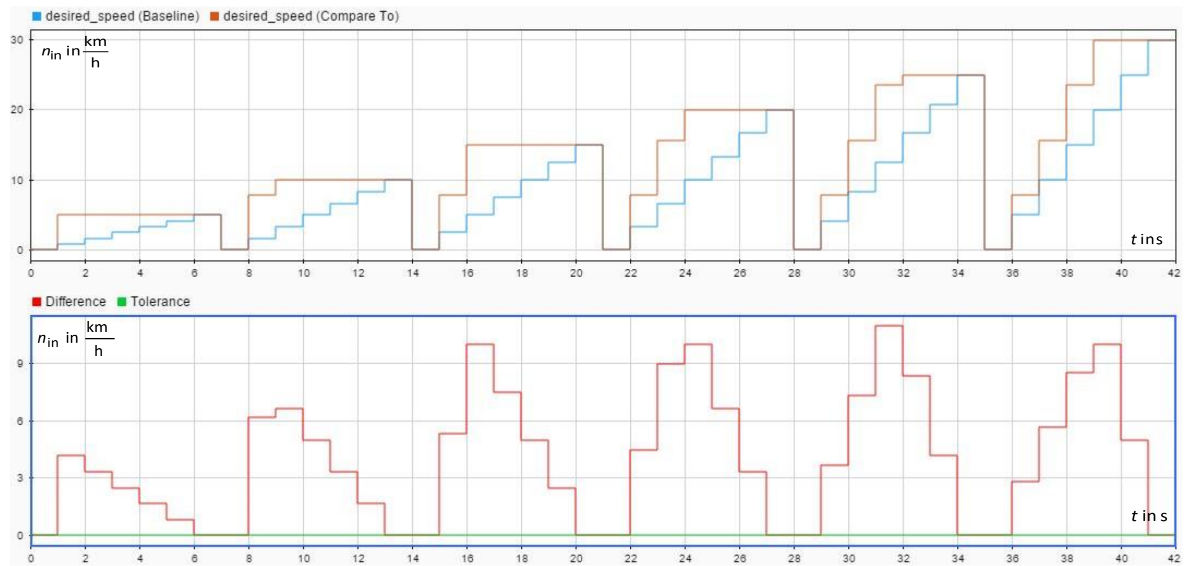


Abbildung 7.1: MIL-Verifikation Geschwindigkeitsmodus

Durch die neue Modellkomponente kann die Geschwindigkeitsanforderung *desired_ speed* innerhalb des gewählten Geschwindigkeitsmodus jeweils in sechs Geschwindigkeitsstufen dosiert werden. Dadurch wird eine bessere Bedienbarkeit gewährleistet.

Traction Control: Der Komponententest *TractionControl_SW_ANF_08* der Modellkomponente *Traction Control* ist als Blackbox-Test ausgeführt und wurde in Abschnitt 6.2.3 dargestellt. Die Systemreaktion des Komponententests sowie deren Auswertung ist in Abbildung 7.2 dargestellt.

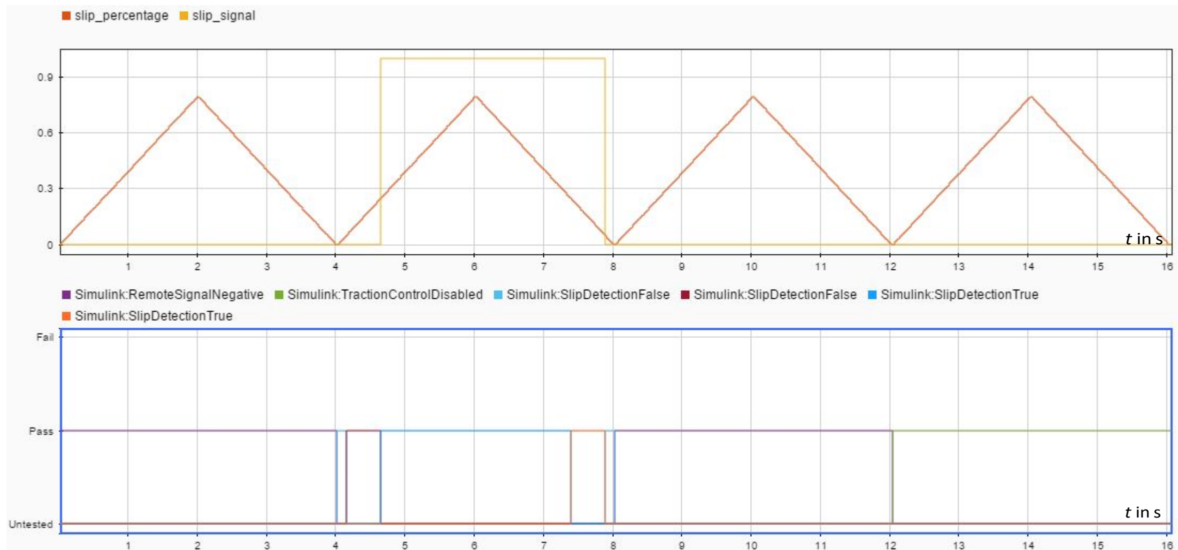


Abbildung 7.2: MIL-Verifikation Traktionskontrolle

Unter Hinzunahme von Abbildung 6.6 ist zu erkennen, dass das Ausgangssignal *slip_signal* nur *true* ist, wenn die Traktionskontrolle aktiviert, das Funkfernbedienungssignal positiv und ein definierter Schlupf von 25 % überschritten ist.

Die automatisierte Testauswertung zeigt, dass die Bewertungskriterien den gesamten Testablauf abdecken und zu jedem Zeitpunkt des Tests mindestens ein *Verify Statement* als bestanden eingestuft ist.

Accelerate Control: Der Komponententest *AccelerateControl_SW_ANF_04_Equivalence* der Modellkomponente *Accelerate Control* ist als Back-to-Back-Test ausgeführt. Die Systemreaktionen für zwei verschiedene Beschleunigungswerte sind in Abbildung 7.3 dargestellt.

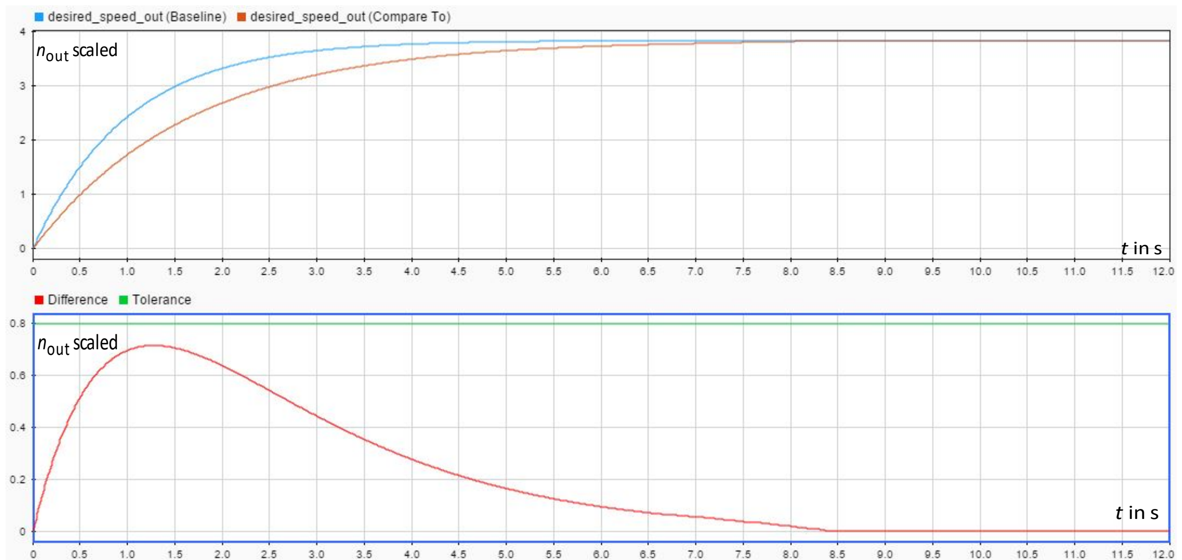


Abbildung 7.3: MIL-Verifikation von zwei Beschleunigungsrampen

Zu erkennen sind zwei unterschiedlich starke Beschleunigungsrampen und deren Abweichung zueinander. Die steilere Beschleunigungsrampe erfüllt die geforderte Beschleunigungszeit von unter 6 Sekunden.

Freewheel Signal and Brake Signal: Die Komponententests *FreewheelSignal_SW_ANF_0* und *BrakeSignal_SW_ANF_02* der Modellkomponente *Freewheel Signal and Brake Signal* sind als Whitebox-Test ausgeführt und wurden bereits in Abschnitt 6.2.4 dargestellt. Abbildung 7.4 zeigt die Systemreaktion des Freilauf-Signals sowie dessen Auswertung.

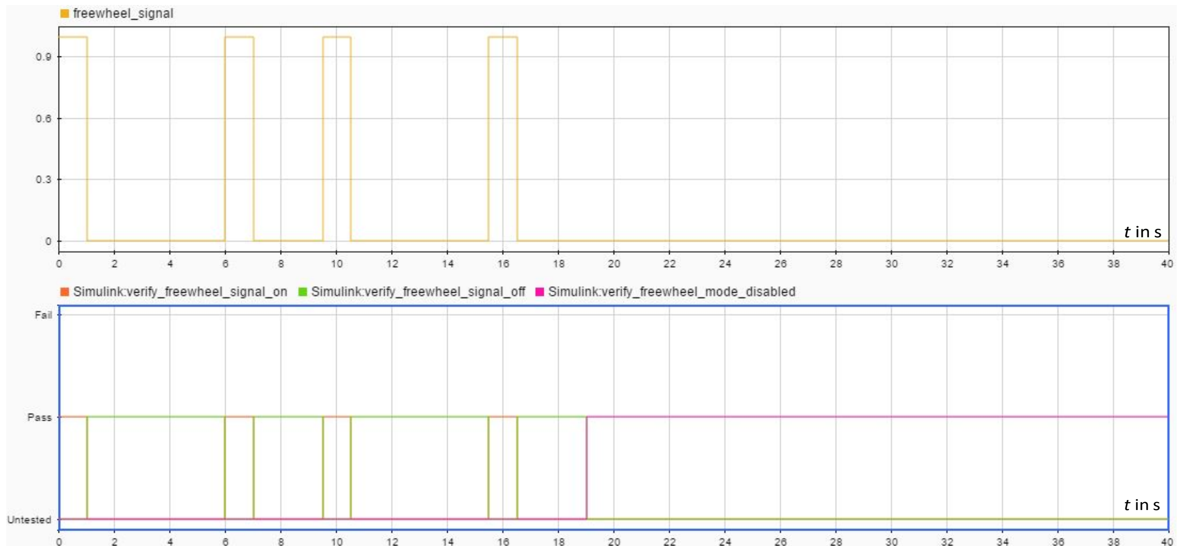


Abbildung 7.4: MIL-Verifikation Freilauf-Signal

Unter Hinzunahme von Abbildung 6.8 ist zu erkennen, dass das Ausgangssignal *freewheel_signal* nur *true* ist, wenn keine Geschwindigkeitsanforderung vorliegt. Die automatisierte Testauswertung zeigt, dass die Bewertungskriterien den gesamten Testablauf abdecken und zu jedem Zeitpunkt des Tests mindestens ein *Verify Statement* als bestanden eingestuft ist.

Analog dazu ist in Abbildung 7.5 die Systemreaktion des Brems-Signals und dessen Auswertung dargestellt.

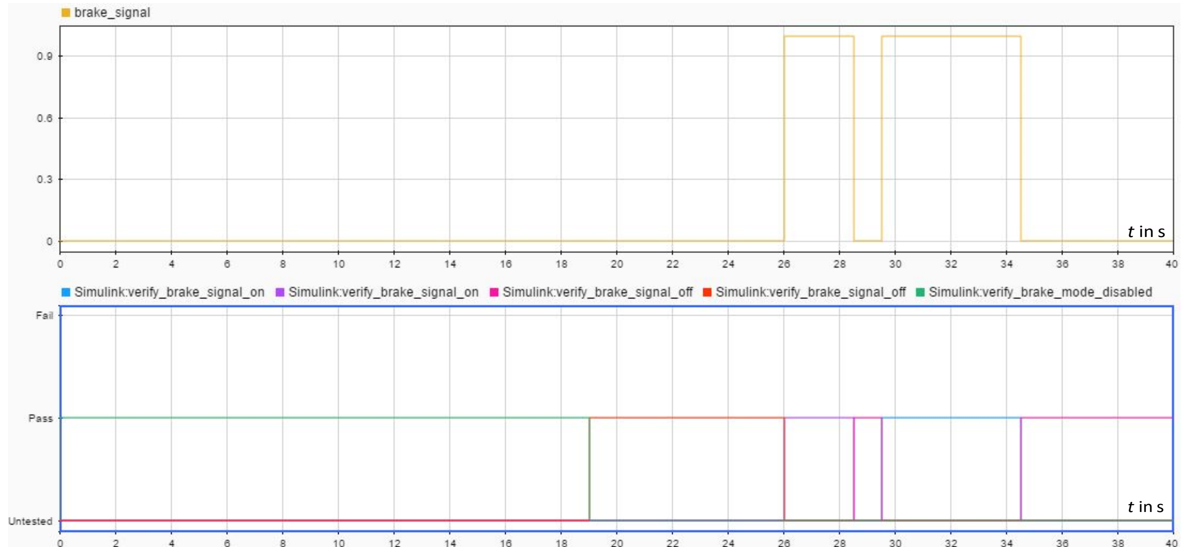


Abbildung 7.5: MIL-Verifikation Brems-Signal

Unter Hinzunahme von Abbildung 6.9 ist zu erkennen, dass das Ausgangssignal *brake_signal* nur *true* ist, wenn eine, abhängig von der Motordrehrichtung, negative Geschwindigkeitsanforderung vorliegt. Die automatisierte Testauswertung zeigt, dass die Bewertungskriterien den gesamten Testablauf abdecken und zu jedem Zeitpunkt des Tests mindestens ein *Verify Statement* als bestanden eingestuft ist.

Torque Controller: Der Komponententest *TorqueController_SW_ANF_07* der Modellkomponente *FOC - Torque Controller* ist als Blackbox-Test ausgeführt. In Abbildung 7.6 sind die Eingangssignale des FOC-Algorithmus über ein Zeitintervall von 0,7 Sekunden dargestellt.

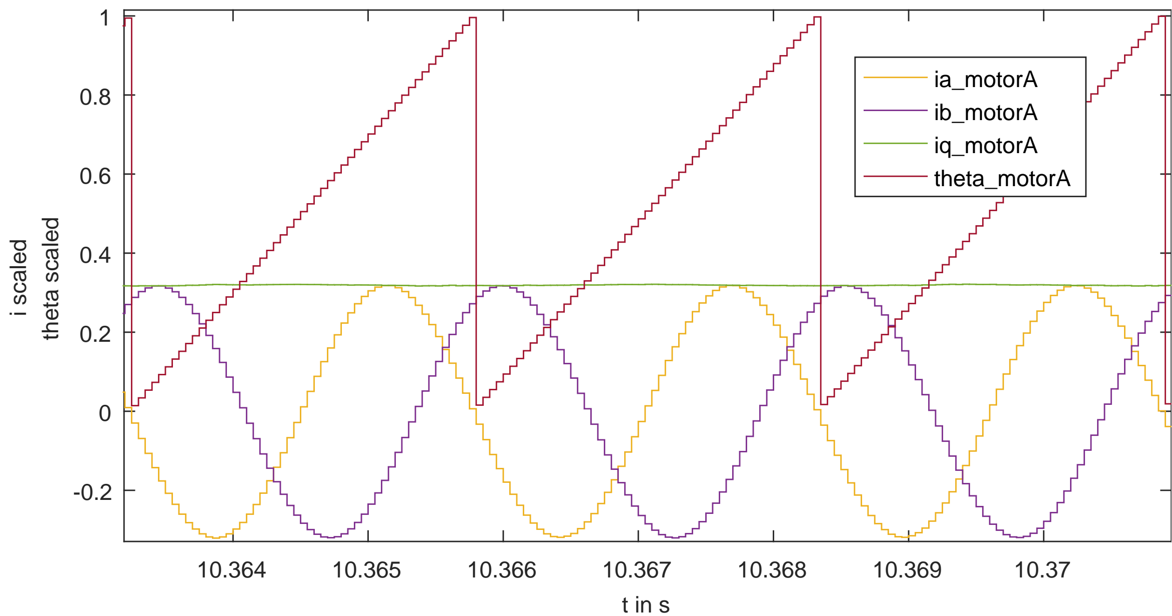


Abbildung 7.6: MIL-Verifikation FOC-Algorithmus

Über die in Abschnitt 6.5 erklärte *Test for Subsystem*-Funktionalität des *Test Manager*-Tools bzw. den erforderlichen Workaround für getriggerte Systeme (vgl. Abbildung 6.19) werden die Testdaten der Phasenströme i_{a} und i_{b} sowie des Polradwinkels ϑ bei angeschlossenem Streckenmodell mittels einer closed-loop Simulation aus dem Feedback des PMSM-Modells erzeugt. Die drehmomentbildende Stromkomponente $i_{\mathrm{s,q,w}}$, welche die Führungsgröße des Stromreglers ist, wird als konstant vorgegeben.

Aufgrund der sich einstellenden Systemreaktion in Form eines Beschleunigungsvorgangs des PMSM-Modells wird das korrekte Verhalten des FOC Algorithmus sichergestellt.

Speed Controller: Der Komponententest *SpeedController_SW_ANF_06* der Modellkomponente *Speed Controller* ist als Blackbox-Test ausgeführt. In Abbildung 7.7 sind die Eingangs- und Ausgangssignale des Drehzahlreglers dargestellt.

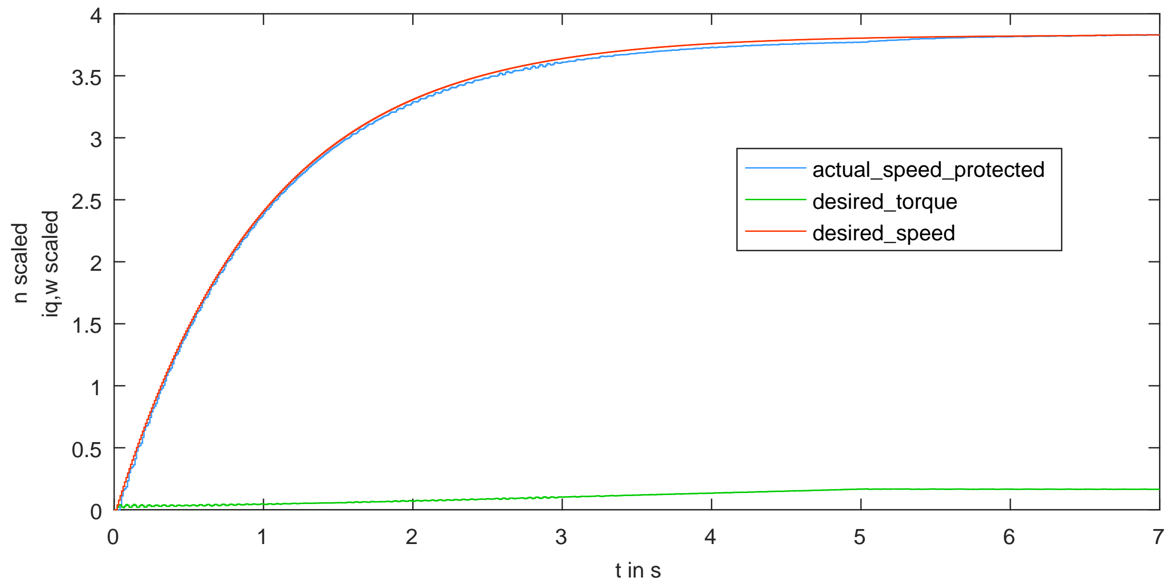


Abbildung 7.7: MIL-Verifikation Drehzahlregler

Die Testdaten werden über die *Test for Subsystem*-Funktionalität mittels closed-loop Simulation aus dem Feedback des angeschlossenen Streckenmodells sowie der Systemreaktion der Modellkomponente *Accelerate Control* generiert, um realistische Eingangsdaten zu erzeugen.

Die Regelgröße *actual_speed* wird mit Hilfe der Stellgröße des Drehzahlreglers *desired_torque* auf die Führungsgröße *desired_speed* geregelt.

7.1.2 Integrationstest

In der nächsten Teststufe werden alle im Komponententest einzeln verifizierten Modellkomponenten integriert, um das Verhalten des gesamten Motorcontroller-Funktionsmodells zu verifizieren.

Dabei werden Systemsimulationen mit angeschlossenem Streckenmodell über den gesamten Last- und Drehzahlbereich durchgeführt. Neben den herkömmlichen Simulationen innerhalb der Modelloberfläche besteht die Möglichkeit mit dem Tool *Test Harness*

der *Simulink Test*-Toolbox ein sogenanntes *Test Harness from Model* zu generieren. Dadurch können Systemsimulationen bzw. modellbasierte Tests auf oberster Modellebene analog den Komponententests mittels *Test Manager*-Tool erstellt, verwaltet, durchgeführt und ausgewertet werden.

In Abbildung 7.8 ist beispielhaft ein Testverlauf des Integrationstests über den gesamten Drehzahlbereich dargestellt.

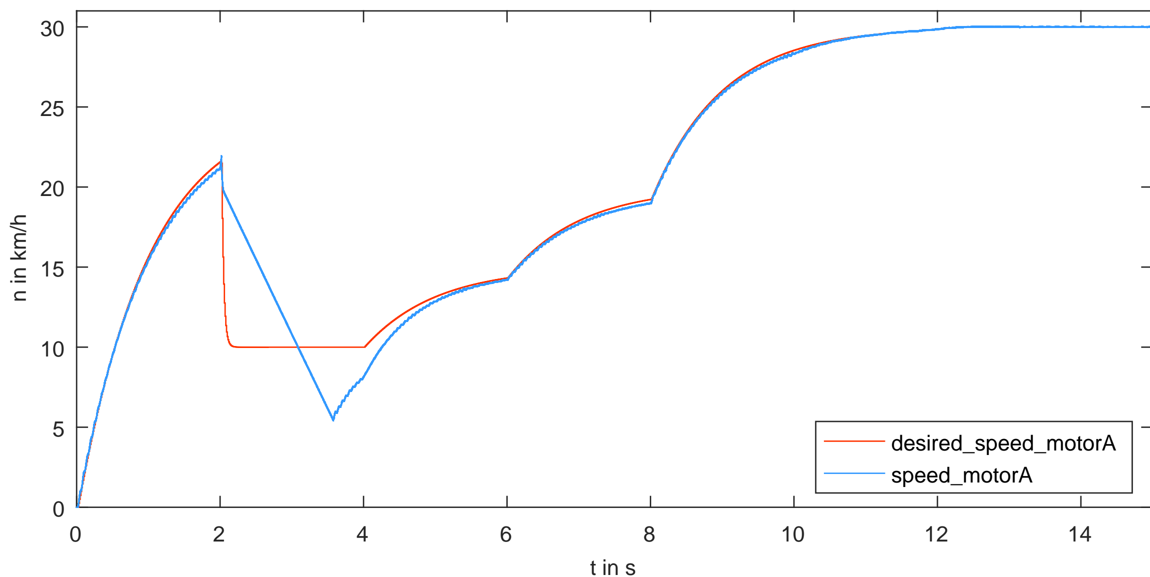


Abbildung 7.8: MIL-Verifikation Integrationstest Motorcontroller

Durch das Simulationsergebnis ist das Systemverhalten des Gesamtmodells verifiziert. Die kaskadierte Geschwindigkeitsregelung des Motorcontroller-Funktionsmodells (vgl. 5.3.1) stellt das Ausregeln der Regelgröße *speed_motorA* entsprechend der Führungsgröße *desired_speed_motorA* sicher, sofern eine Belastung im definierten Lastbereich vorliegt. Das resultierende Systemverhalten der sich einstellenden Geschwindigkeit wird in Abschnitt 7.2.1 erläutert.

Der dargestellte Testverlauf wird als Referenz für den folgenden Systemintegrationstest gewählt.

7.2 Systemverifikation Motorcontroller EpOS

Das innerhalb der Simulink-Umgebung durch MIL-Tests verifizierte Implementierungsmodell EpOS wird zur automatischen Codegenerierung durch den *Emedded Coder* genutzt. Die funktionale Äquivalenz der eingebetteten Software des Motorcontrollermoduls wird entsprechend der Teststrategie für den modellbasierten Test (vgl. Tabelle 2.1 Schritt 4) durch Back-to-Back-Tests auf der Hardware verifiziert (HIL-Tests).

7.2.1 Systemintegrationstest

Die modellbasiert entwickelte Software wird mit der parallel entwickelten Hardware zu einem Traktionswechselrichter eingebettet (vgl. [5]). Das Konzept sowie das fertige Motorcontrollermodul EpOS sind in Abbildung 7.9 dargestellt.

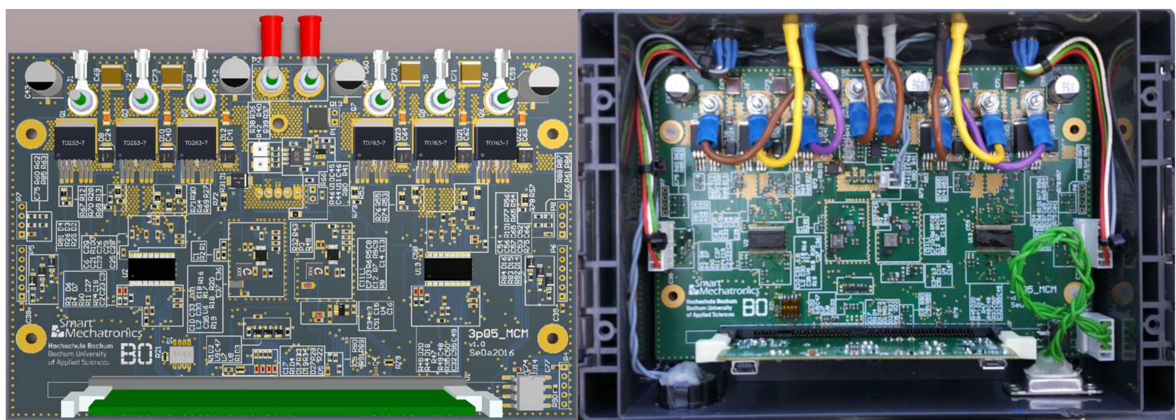


Abbildung 7.9: Konzept und Fotografie des Motorcontrollermoduls EpOS

Für aussagekräftige Back-to-Back-Tests wird ein PCAN-USB-Adapter sowie die zugehörige Software PCAN-View genutzt. Über diesen CAN-Monitor werden die Testdaten für den Systemintegrationstest auf den CAN-Bus gesendet und die funktionale Äquivalenz des Systemverhaltens von Implementierungsmodell und Embedded System wird durch das Aufzeichnen des CAN-Datenverkehrs verifiziert.

Accelerate Control: Abbildung 7.10 zeigt die Systemreaktion der Modellkomponente *Accelerate Control* auf die gleichen Testdaten, die für die Verifikation des Motorcontroller-Funktionsmodells genutzt wurden.

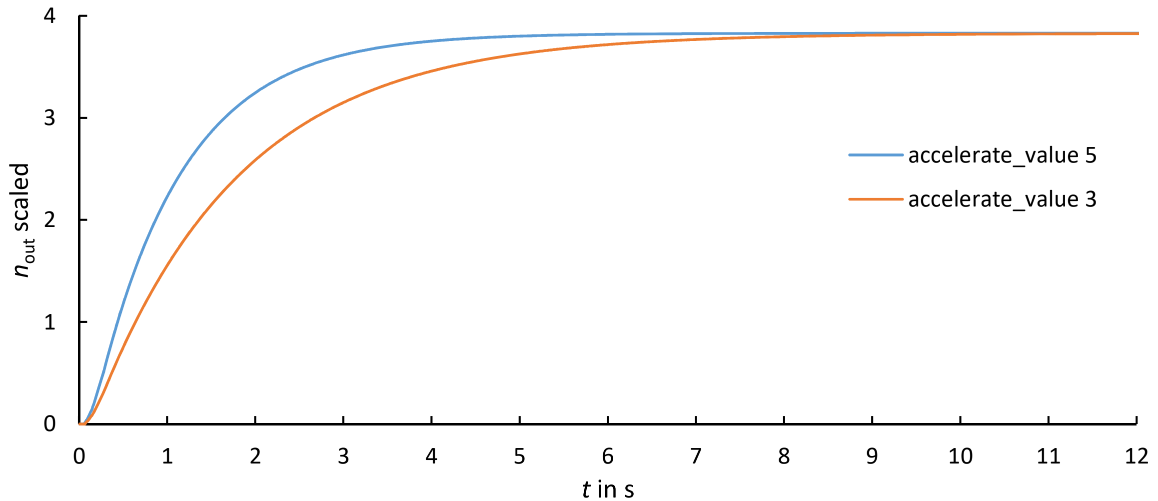


Abbildung 7.10: HIL-Verifikation von zwei Beschleunigungsrampen

Es zeigt sich eine funktionale Äquivalenz zu der in Abbildung 7.3 dargestellten Systemreaktion des Motorcontroller-Funktionsmodells.

Speed Controller: In Abbildung 7.11 ist das Testergebnis der Modellkomponente *Speed Controller* dargestellt.

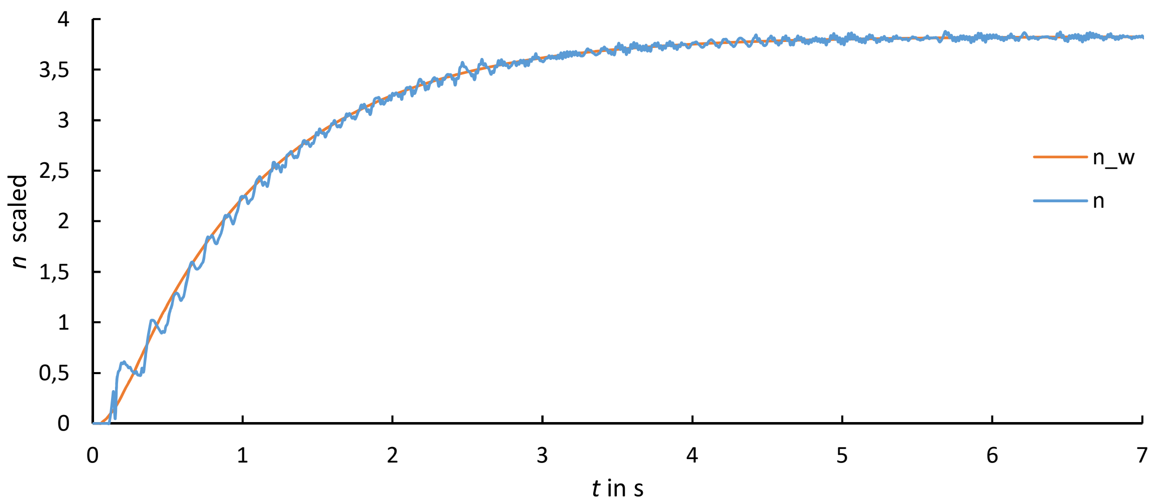


Abbildung 7.11: HIL-Verifikation Drehzahlregler

Es ist eine funktionale Äquivalenz zwischen MIL- und HIL-Test gegeben (vgl. Abbildung 7.7). Die Schwankungen der Geschwindigkeit n können mit der fehlenden Last beim HIL-Test begründet werden. Die Motordrehzahl wird im Leerlaufbetrieb gemessen, da ein geeignete Motorprüfstand nicht zur Verfügung steht.

In Abbildung 7.12 ist die gemessene Abweichung der Geschwindigkeiten n_{w} und n im Leerlaufbetrieb dargestellt.

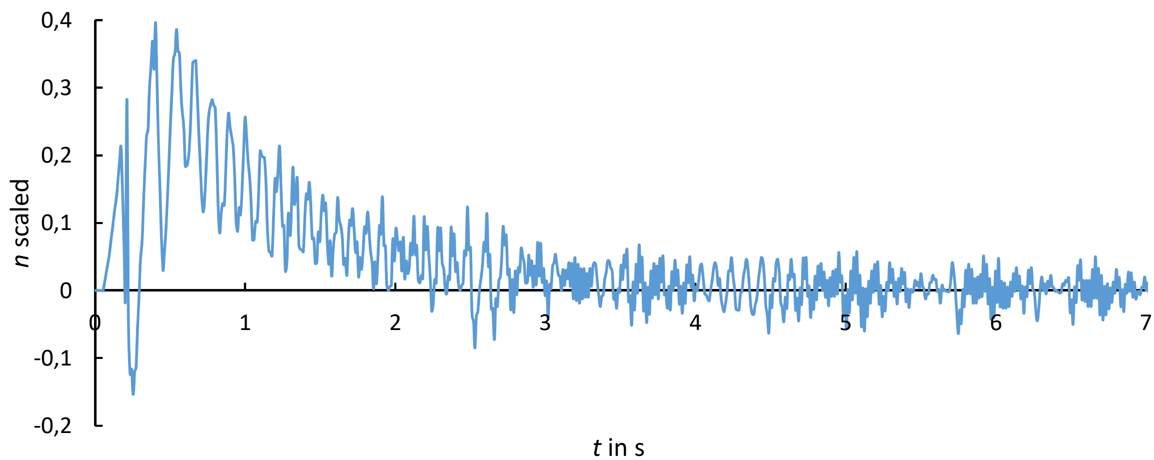


Abbildung 7.12: HIL-Verifikation Drehzahlreglerabweichung

Im Leerlaufbetrieb sind aufgrund der geringeren Trägheit schnellere Drehzahlsprünge des Motors möglich, die eine Schwankung der Geschwindigkeit n bewirken. Diese schnellen Drehzahlsprünge werden von dem für den Belastungsfall parametrierten PI-Regler nicht genau ausgeregelt.

Motorcontroller: Äquivalent zum Integrationstest in Abschnitt 7.1.2 wird der Testverlauf für den Systemintegrationstest des Embedded Systems durchgeführt. Abbildung 7.13 zeigt das Testergebnis der kaskadierten Geschwindigkeitsregelung. Abbildung 7.14 stellt die Reglerabweichung zwischen n_w und n dar.

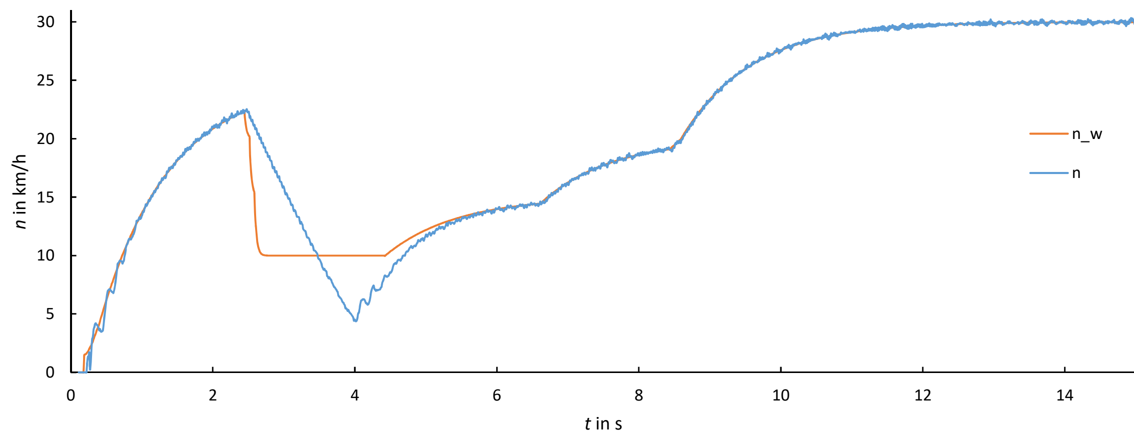


Abbildung 7.13: HIL-Verifikation Systemintegrationstest Motorcontroller

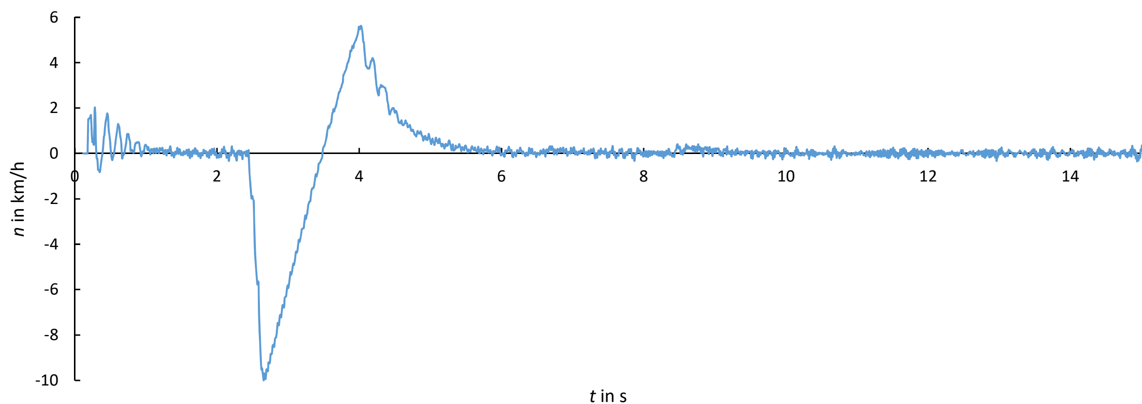


Abbildung 7.14: HIL-Verifikation Reglerabweichung Motorcontroller

Die Systemreaktion des Systemintegrationstest weist funktionale Äquivalenz zum Integrationstest auf (vgl. Abbildung 7.8). Auffällig ist das Systemverhalten beider Testfälle beim Übergang zu einer geringeren Geschwindigkeitsanforderung n_w nach ca. 2 Sekunden. Sowohl in der Modellsimulation, als auch beim Test auf dem Motorcontrollermodul wird die Geschwindigkeit des PMSM-Modells bzw. des angeschlossenen Motors nicht runtergeregelt. Dadurch stellt sich eine negative Regeldifferenz ein

(siehe Abbildung 7.14). Diese wird jedoch nicht ausgeregelt, da den Anforderungen entsprechend kein aktiver Bremsvorgang durch Feldstellung eingeleitet wird, um ein natürliches Fahrverhalten des E-Mountainboards sicherzustellen. Die drehmomentbildende Stromkomponente $i_{\mathsf{s,q,w}}$ wird daher zu Null gesetzt, wodurch ein unbelastetes Ausdrehen des PMSM-Modells bzw. des angeschlossenen Motors resultiert.

Sobald sich nach ca. 3 Sekunden eine positive Regeldifferenz einstellt, wird $i_{\mathsf{s,q,w}}$ wieder entsprechend der Drehmomentanforderung des Drehzahlreglers gestellt, sodass die angeforderte Geschwindigkeit n_{w} erreicht wird.

Die eingebettete Software des Motorcontrollermoduls EpOS ist somit verifiziert und erfüllt alle im Systementwurf gestellten Anforderungen. Nach erfolgreicher Verifikation der parallel getesteten Hardware werden zwei Traktionswechselrichter in das E-Mountainboard integriert.

Das Gesamtsystem E-Mountainboard wird im Feldversuch erprobt. Im Zuge dessen erfolgt die domänenübergreifende Verifikation aller Systemanforderungen des Lastenhefts (siehe Anhang A.1.16).

7.2.2 Abnahmetest und Validierung

Nach der Systemverifikation und Feldversuchen wird das fertige Produkt E-Mountainboard EpOS, welches in Abbildung 7.15 dargestellt ist, vom Projektteam an den Auftraggeber übergeben.

Abbildung 7.15: Konzept und Fotografie des E-Mountainboards EpOS

Durch einen Abnahmetest des Auftraggebers in Form einer Testfahrt unter realistischen Einsatzbedingungen wird das Produkt abgenommen und für valide erklärt. Das Projekt ist damit erfolgreich abgeschlossen.

8 Beurteilung der Toolkette

Die gesamte Toolkette des modellbasierten Entwicklungsprozesses wurde am Limit der aktuellen Möglichkeiten betrieben. Die Entwicklung einer Modellstruktur, die sowohl für die Generierung von hardwarespezifischem C-Code, als auch für die Durchführung von Modellsimulationen mit angeschlossenem Streckenmodell sowie modellbasierten Tests genutzt werden kann, stellte sich mit der aktuell verfügbaren Toolkette als große Herausforderung dar. Eine solche Modellstruktur war jedoch Voraussetzung für das vorliegende Projekt, da nur auf diese Weise der gleiche Modellstand getestet und verifiziert werden kann, welcher in der Anwendung eingebettet wird.

Dabei stellte sich die Verwendung von *Function-Call* getriggerten Subsystemen als größte Problematik dar, da diese von vielen Tools nicht unterstützt werden. Ein Implementierungsmodell für die Generierung von hardwarespezifischem C-Code kommt jedoch in den meisten Anwendungen nicht ohne getriggerte Subsysteme aus.

Der gesamte modellbasierte Entwicklungsprozess ist aktuell für die Simulation und Verifikation von physikalischen Modellen oder Implementierungsmodellen ohne interne Hardwareanbindung ausgelegt (vgl. Abbildung 2.6). Für Entwickler und Tester von eingebetteter Software ist es jedoch von Vorteil, den Modellstand zu simulieren, zu analysieren und zu verifizieren, der für die automatische Codegenerierung genutzt wird, um den generierte Code nicht zusätzlich einem aufwändigen Verifikationsprozess unterziehen zu müssen. Es wird daher als suboptimal erachtet, ausschließlich ein frühes Artefakt des Modells zu verifizieren, das lediglich die eigentlichen Regelungs- und Steuerungsalgorithmen beinhaltet, jedoch nicht an den Zielprozessor angepasst ist. Denn nur auf diese Weise kann eine durchgängige modellbasierte Entwicklung gewährleistet werden.

Der modellbasierte Testprozess mit den Toolboxen *Simulink Verification and Validation* und *Simulink Test* überzeugte gänzlich, sobald die den Anforderungen entsprechende Modellstruktur des Implementierungsmodells vorhanden war.

Die eindeutige Rückverfolgbarkeit durch die Verlinkung von Anforderungsdokument und Implementierungsmodell hilft beim Überblick von bereits implementierten und noch fehlenden Modellfunktionen und gibt dem Entwickler Sicherheit für Reviews.

Die Erstellung von Testrahmen der zu testenden Modellkomponenten in einer separaten Simulationsumgebung, die automatisch mit dem Hauptmodell verwaltet und synchronisiert werden, erleichtern die Testfallentwicklung, da die Testumgebung bereits automatisch vorhanden ist. Allgemeine Testfallberschreibungen können durch eine *Test Sequence* oder einen *Signal Builder* in für Simulink lesbare Testdaten übersetzt werden, ohne dass ein Zwischenschritt notwendig ist. Die Definition von Testbewertungskriterien aus Anforderungsdokumenten mithilfe eines *Test Assessments* zeigt sich als sehr hilfreich hinsichtlich der Analyse und Auswertung von Testergebnissen.

Mit dem *Test Manager*-Tool gelingt es, den gesamten Testprozess übersichtlich und strukturiert zusammenzuführen und zu verwalten. Die Möglichkeit zur Erstellung verschiedener *Test Suiten* für Komponenten- und Integrationstests, die Durchführbarkeit einzelner oder der gesamten Testfälle über nur einen Befehl sowie die grafische Darstellung der Systemreaktionen inkl. Modellüberdeckungsmessung und automatischer Auswertung der Ergebnisse vereinfachen die Testaktivitäten Testdurchführung und Testauswertung enorm. Aufgrund der Exportierungsmöglichkeit können die Testdaten außerdem für spätere Analysen oder Reviews innerhalb des *Test Manager*-Tools genutzt werden, ohne dass alle Testergebnisse manuell gespeichert und aufgearbeitet werden müssen. Die wenigen Limitierungen und Fehler im *Test Manager* werden in neuen Releases behoben bzw. sind im aktuellen Release 2016b bereits durch zusätzliche Funktionalitäten ergänzt worden und werden daher nicht als gravierend eingestuft.

Die evaluierten Tools zum modellbasierten Test vervollständigen die bis dahin am Institut für Systemtechnik eingesetzte Toolkette zur modellbasierten Entwicklung und werden für eine erfolgreiche Umsetzung eines durchgängigen modellbasierten Entwicklungsprozesses von der Anforderungserhebung bis zur Verifikation und Validierung des fertigen Produktes als notwendig erachtet. Daher wird empfohlen, die evaluierte Toolkette in zukünftigen Projekten des Instituts für Systemtechnik sowie der Smart Mechatronics GmbH einzusetzen.

9 Fazit

Die Umsetzung einer modellbasierten Entwicklung und Verifikation konnte in diesem Projekt erfolgreich in einer Praxisanwendung am Beispiel einer modularen Antriebsplattform gezeigt werden. Die modellbasiert entwickelte und verifizierte Software konnte mit der parallel entwickelten Hardware zu einem Motorcontrollermodul eingebettet und erfolgreich als Traktionswechselrichter in einem elektrischen Mountainboard integriert und erprobt werden. Dadurch wurde ein Antriebssystem mit vier separat angesteuerten PMSM und integrierter Traktionskontrolle umgesetzt.

Durch die Möglichkeit einer zwischen Hardware und Simulation umschaltbaren Modellstruktur, konnte der notwendige „Knopf“ konzeptioniert und realisiert werden, um aus dem Implementierungsmodell ein für die Simulation und den modellbasierten Test relevantes Artefakt zu erzeugen.
Aufgrund des modellbasierten Tests konnten in diesem Projekt viele Fehlerwirkungen bereits auf Modellebene aufgedeckt werden, die ansonsten erst während der HIL-Tests auf dem Embedded System aufgefallen wären und ein zeit- und kostenintensiveres Redesign mit sich gezogen hätten. Alle Softwareanforderungen konnten bereits im MIL-Test verifiziert werden. Die Ergebnisse der Modellverifikation zeigen im Vergleich zu der darauf folgenden Systemverifikation sehr überzeugende Ergebnisse. Im HIL-Test und im Feldversuch musste lediglich die funktionale Äquivalenz der eingebetteten Software durch Back-to-Back-Tests bestätigt, sowie die Feinparametrierung der Regler vorgenommen werden, ohne dass neue Fehlerwirkungen aufgetreten sind.

Der modulare Architekturaufbau des Implementierungsmodells ermöglicht es, die modellbasiert entwickelte Software der Regelungs- und Steuerungsfunktionen des Motorcontrollers separat von anderen Softwarefunktionen des E-Mountainboards zu organisieren. Dadurch besteht die Software des Motorcontrollermoduls ausschließlich aus modellbasiert entwickeltem Code des Implementierungsmodells EpOS und kann in andere elektromobile Nutzanwendungen integriert werden.

Abbildungsverzeichnis

2.1 Klassifizierung von Software-Testverfahren 5
2.2 Testrahmen Blackbox-Verfahren 7
2.3 Testrahmen Whitebox-Verfahren 10
2.4 Testprozess nach ISTQB 12
2.5 Traditionelle und modellbasierte Softwareentwicklung 15
2.6 Testartefakte beim modellbasierten Test [14] 16
2.7 Teststrategie für den modellbasierten Test [14] 19

3.1 Einordnung von CONSENS® im V-Modell [15] 20
3.2 Durchgeführter Worfkflow nach CONSENS® [15] 21
3.3 Umfeldmodell E-Mountainboard EpOS 22

4.1 Wirkstruktur E-Mountainboard EpOS 25
4.2 Wirkstruktur Elektronik 26
4.3 Wirkstruktur Motorcontrollermodul 28

5.1 Minimalbeispiel der Implementierungsmodellstruktur 33
5.2 Gesamtstruktur des Motorcontroller-Funktionsmodells 36
5.3 Funktion des Subsystems Geschwindigkeitsmodus 39

6.1 Durchgeführter Testprozess mit Simulink 46
6.2 Einordnung der Testfallbeschreibung innerhalb des Testprozesses 47
6.3 Anforderungsnachverfolgung *Freewheel Signal and Brake Signal* 48
6.4 Modellüberdeckungsanalyse *Freewheel Signal and Brake Signal* 50
6.5 Modellüberdeckungsanalyse *Max Speed through Speedmode* 51
6.6 Testdatenverläufe Blackbox-Test *Traction Control* 53
6.7 Modellstruktur *Freewheel Signal and Brake Signal* 54
6.8 Testdatenverläufe Whitebox-Test *Freewheel Signal* 55
6.9 Testdatenverläufe Whitebox-Test *Brake Signal* 56

6.10 Testdatenverläufe Back-to-Back-Test *Max Speed through Speedmode* . . 57
6.11 Testdatenerstellung über *Test Sequence* 59
6.12 Testdatenerstellung über *Signal Builder* 59
6.13 Testrahmenerstellung der Modellkomponente *Traction Control* 60
6.14 Testbewertung der Modellkomponente *Traction Control* 61
6.15 Testdurchführung mit dem *Test Manager* 63
6.16 Gesamtübersicht der Testauswertung im *Test Manager* 65
6.17 Testauswertung bestandener Testfall im *Test Manager* 66
6.18 Testauswertung nicht bestandener Testfall im *Test Manager* 67
6.19 Testrahmen *FOC - Torque Controller* mit Streckenmodell 70

7.1 MIL-Verifikation Geschwindigkeitsmodus 75
7.2 MIL-Verifikation Traktionskontrolle 76
7.3 MIL-Verifikation von zwei Beschleunigungsrampen 77
7.4 MIL-Verifikation Freilauf-Signal 78
7.5 MIL-Verifikation Brems-Signal 79
7.6 MIL-Verifikation FOC-Algorithmus 80
7.7 MIL-Verifikation Drehzahlregler 81
7.8 MIL-Verifikation Integrationstest Motorcontroller 82
7.9 Konzept und Fotografie des Motorcontrollermoduls EpOS 83
7.10 HIL-Verifikation von zwei Beschleunigungsrampen 84
7.11 HIL-Verifikation Drehzahlregler 84
7.12 HIL-Verifikation Drehzahlreglerabweichung 85
7.13 HIL-Verifikation Systemintegrationstest Motorcontroller 86
7.14 HIL-Verifikation Reglerabweichung Motorcontroller 86
7.15 Konzept und Fotografie des E-Mountainboards EpOS 88

Tabellenverzeichnis

2.1 Eingesetzte Teststrategie für den modellbasierten Test nach [12] 18

4.1 Vergleich möglicher TI C2000 Derivate 29
4.2 Verwendete Software-Toolkette 31

6.1 Schnittstellensignale der Modellkomponente *Traction Control* 52
6.2 Schnittstellensignale der Modellkomponente *Freewheel Signal and Brake Signal* 55
6.3 Schnittstellensignale der Modellkomponente *Max Speed through Speedmode* 57

Literatur

[1] Kevin Leiffels. »Entwicklung eines E-Skateboards«. Masterthesis. Hochschule Bochum - Bochum University of Applied Sciences, 2015.

[2] Raphael-David Volmering. »Entwicklung der Leistungselektronik eines E-Skateboards«. Masterthesis. Hochschule Bochum - Bochum University of Applied Sciences, 2015.

[3] Mirko Conrad; Ines Fey. »Testing Automotive Control Software«. In: *Automotive Embedded Systems Handbook*. Boca Raton, USA: CRC Press, 2009. Kap. 11.

[4] SolarCar-Projekt Hochschule Bochum. 2016. URL: `www.bosolarcar.de/`.

[5] Sean William Dalton. »Entwicklung der eingebetteten Hardware einer modularen Antriebsplattform«. Bachelorarbeit. Hochschule Bochum - Bochum University of Applied Sciences, 2017.

[6] Andreas Spillner; Tilo Linz. *Basiswissen Softwaretest*. Bd. 5. Auflage. Heidelberg: dpunkt.verlag, 2012.

[7] Frank Witte. *Testmanagement und Softwaretest*. Wiesbaden: Springer Vieweg, 2016.

[8] Thomas Roßner; Christian Brandes; Helmut Götz; Mario Winter. *Basiswissen Modellbasierter Test*. Heidelberg: dpunkt.verlag, 2010.

[9] Peter Liggesmeyer. *Software-Qualität - Testen, Analysieren, und Verifizieren von Software*. Bd. 2. Auflage. Heidelberg: Spektrum Akademischer Verlag, 2009.

[10] Dirk W. Hoffmann. *Software-Qualität*. Bd. 2. Auflage. Berlin Heidelberg: Springer Vieweg, 2013.

[11] Holger Schlingloff. »Modellbasiertes Testen fordert einen differenzierten Blick«. In: *Computerzeitung* 44 (Okt. 2007).

[12] Mirko Conrad. *Modell-basierter Test eingebetteter Software im Automobil - Auswahl und Beschreibung von Testszenarien*. Bd. 1. Auflage. Wiesbaden: Deutscher Universitäts-Verlag, 2004.

[13] Holger Schlingloff; Mirko Conrad; Heiko Dörr; Carsten Sühl. *Modellbasierte Steuergerätesoftwareentwicklung für den Automobilbereich*. Techn. Ber. Humbold-Universität zu Berlin, 2004.

[14] Mirko Conrad. *Automotive Software Enginerring - Modell-basiertes Testen*. Vorlesungsskript WS 2005/2006 Humbold-Universität zu Berlin.

[15] Smart Mechatronics GmbH. *CONSENS Workshop HS Bochum*. 2016.

[16] Andrea Herrmann; Eric Knauss; Rüdiger Weißbach. *Requirements Engineering und Projektmanagement*. Berlin Heidelberg: Springer Vieweg, 2013.

[17] Marcus Grande. *100 Minuten für Anforderungsmanagement*. Bd. 2. Auflage. Wiesbaden: Springer Vieweg, 2014.

[18] Texas Instruments. *TMS320F2806x Piccolo Microcontrollers*. SPRS698F, 2016.

[19] Texas Instruments. *TMS320F28335 Digital Signal Controllers*. SPRS439M, 2012.

[20] Texas Instruments. *TMS320F2807x Piccolo Microcontrollers*. SPRS902C, 2014.

[21] Ramu Krishnan. *Permanent Magnet Synchronous and Brushless DC Motor Drives*. Boca Raton, USA: CRC Press, 2010.

[22] Florian Wagner. »Entwicklung und Implementierung einer feldorientierten Regelung eines bürstenlosen Motors für ein Stabantriebssystem unter Verwendung eines modellbasierten Entwicklungsprozesses mit MATLAB/Simulink«. Bachelorthesis. Hochschule Bochum - Bochum University of Applied Sciences, 2015.

[23] Hermann Winner; Stephan Hakuli; Felix Lotz; Christina Singer. *Handbuch Fahrerassistenzsysteme*. Bd. 3. Auflage. Wiesbaden: Springer Vieweg, 2015.

[24] The MathWorks. *Simulink Verification and Validation User's Guide*. Version 3.11, 2016.

[25] The MathWorks. *Simulink Test User's Guide*. Version 2.0, 2016.

[26] Adam Whitmill. Schriftliche und mündliche Kommunikation mit der Firma The MathWorks zwischen Juni 2016 und Februar 2017.

A Anhang

A.1 Simulink Project auf Daten-CD

EpOS_Motorcontroller\...

1 requirements\20160708_Anwendungsszenarien.pdf

2 requirements\20160831_Lastenheft_E-Skateboard_V2.0.pdf

3 requirements\20161201_Systemmodellanforderungen_Motorcontroller_EpOS.pdf

4 requirements\20161118_Signalbeschreibung_Motorcontroller_EpOS.xlsx

5 documents\Leitfaden_zur_Codegenerierung.pdf

6 models\EpOS_back.slx bzw. models\EpOS_front.slx

7 code\EpOS_back_ert_rtw bzw. code\EpOS_front_ert_rtw

8 documents\CAN_Messages.slx und documents\CANmapEpOS.slx

9 requirements\20161201_Systemmodellanforderungen_Motorcontroller_EpOS.docx

10 reports\20170214_EpOS_back_requirements.html

11 reports\20170214_Requirements_Consistency_Checking.pdf

12 tests\EpOS_TestFile.mldatx

13 tests\test_results\20170214_EpOS_TestResults.mldatx

14 reports\20170214_EpOS_Test_Results_Report.pdf

15 reports\20170214_EpOS_back_Speedmode_Coverage_Report.pdf

16 tests\test_results\20170214_Verifikation_Lastenheft.pdf

A.2 Sonstiger Inhalt auf Daten-CD

1 Masterarbeit

2 Projektauftrag

3 Meilensteinplan

4 Datenblätter

5 Digitale Quellen

6 Abbildungen

7 Besprechungsprotokolle