

Kristian Trenkel

Konzept und Umsetzung einer modularen, portierbaren Middleware für den automatisierten Test eingebetteter Systeme

Kristian Trenkel

Konzept und Umsetzung einer modularen, portierbaren
Middleware für den automatisierten Test
eingebetteter Systeme



TECHNISCHE UNIVERSITÄT
CHEMNITZ

Universitätsverlag Chemnitz

2016

Impressum

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Angaben sind im Internet über <http://dnb.d-nb.de> abrufbar.

Technische Universität Chemnitz/Universitätsbibliothek
Universitätsverlag Chemnitz
09107 Chemnitz
<http://www.tu-chemnitz.de/ub/univerlag>

Herstellung und Auslieferung
Verlagshaus Monsenstein und Vannerdat OHG
Am Hawerkamp 31
48155 Münster
<http://www.mv-verlag.de>

ISBN 978-3-944640-70-9

<http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-192611>

Konzept und Umsetzung einer modularen, portierbaren Middleware für den automatisierten Test eingebetteter Systeme

von der Fakultät für Elektrotechnik und Informationstechnik
der Technischen Universität Chemnitz

genehmigte
Dissertation
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften

Dr.-Ing.

vorgelegt

von Dipl.-Ing. (FH) KristianTrenkel

geboren am 20. März 1983 in Gera

eingereicht am 19.05.2015

Gutachter:

Prof. Dr.-Ing. Ulrich Heinkel

Prof. Dr.-Ing. Jürgen Teich

Tag der Verleihung: 30.11.2015

Bibliographische Beschreibung

Titel: Konzept und Umsetzung einer modularen, portierbaren Middleware für den automatisierten Test eingebetteter Systeme

Kristian Trenkel – 2015 – 207 Seiten

57 Abbildungen, 26 Tabellen, 20 Listings, 128 Literaturquellen

Technische Universität Chemnitz, Fakultät für Elektrotechnik und Informationstechnik, Dissertation

Referat

Die Dissertationsschrift diskutiert die Machbarkeit einer modularen, portierbaren Middleware für die automatisierte Ausführung und Dokumentation von Software-Tests mit einer durchgehenden Nachverfolgbarkeit von der Anforderungsspezifikation bis hin zur Dokumentation der Testergebnisse. Es werden die Eigenschaften und Probleme bestehender Testautomatisierungslösungen analysiert und dargelegt. Unter Berücksichtigung dieser Probleme werden neuartige Lösungsansätze entwickelt. Die Neuheiten dieser Arbeit sind der modulare Aufbau der Middleware mit einer unproblematischen Portierbarkeit auf neue Testsysteme in Verbindung mit dem neu erarbeiteten Speicherformat für die Testergebnisse. Es wird die Möglichkeit aufgezeigt, Testfälle sowohl mit graphischer als auch textueller Eingabe zu bearbeiten. Neben den typischen Einsatzbereichen, wie zum Beispiel Hardware In The Loop-Tests (HIL), werden auch weitere Felder, vom Modul-Test bis zum Bandende-Test, abgedeckt. Das Speicherformat der Testergebnisse ermöglicht die Ablage aller wichtigen Informationen zu den Tests, ist flexibel erweiterbar und erlaubt die Generierung von Testreports in unterschiedlichen Zielformaten. Ein weiterer zentraler Punkt ist der automatisierte Austausch von Informationen und Testergebnissen mit verschiedenen Requirementsmanagement-Systemen sowie eine nahtlose Integration in vorhandene Versionsmanagement-Systeme. Basierend auf den theoretischen Ausarbeitungen wurde eine modulare, portierbare Middleware in Form des modularen aufgebauten Testautomatisierungs-Frameworks (modTF) umgesetzt. Anhand der dabei gesammelten Erfahrungen und der Ergebnisse der praktischen Erprobung werden die Vorteile des Frameworks gezeigt.

Schlagwörter

modulare, portierbare Middleware für Testautomatisierung, Automatisierter Softwaretest, TTCN-3, durchgehende Nachverfolgbarkeit

Inhaltsverzeichnis

Abkürzungsverzeichnis	ix
Danksagung	xiii
1. Einleitung	1
1.1. Motivation	1
1.2. Ziel der Arbeit	2
1.3. Struktur der Arbeit	3
2. Grundlagen	5
2.1. Aufbau von eingebetteten Systemen	5
2.1.1. Übersicht	5
2.1.2. Rechenkern	6
2.1.3. Aktorik	7
2.1.4. Sensorik	7
2.1.5. Bussysteme	8
2.1.6. Spezielle Trends im Automotive-Bereich	9
2.2. Test- und Entwicklungsprozess	10
2.2.1. Test	10
2.2.2. Der Produktlebenszyklus	10
2.2.3. Sequentielle Entwicklungsprozesse	12
2.2.4. Iterativ-Inkrementelle Entwicklungsprozesse	15
2.2.5. Entwicklung im Automotive-Bereich	19
2.2.6. Der fundamentale Testprozess	21
2.2.7. Test im Automotive-Bereich	21
2.2.8. Spezielle Trends im Automotive-Bereich	23
2.3. Software-Test	24
2.3.1. Arten des Software-Tests	24
2.3.2. Testautomatisierung im Software-Test	28
2.4. Fertigungsendtest	31
3. Stand der Technik	33
3.1. Testbeschreibungsmethoden	33
3.1.1. Programmatischer Ansatz	33
3.1.2. Modellbasierter Ansatz	38
3.1.3. Grenzen und Probleme verfügbarer Testbeschreibungsmethoden	42

3.2.	Testautomatisierungssysteme	43
3.2.1.	Standards für Testausführungssysteme	43
3.2.2.	Kommerzielle Produkte	45
3.2.3.	Firmeninterne Entwicklungen	50
3.2.4.	Freie Entwicklungen	52
3.2.5.	Grenzen und Probleme verfügbarer Testautomatisierungs-Tools	53
3.3.	Testdokumentation	55
3.3.1.	Standards	55
3.3.2.	Inhalte	56
3.3.3.	Formate	56
3.3.4.	Praktische Umsetzungen	57
3.4.	Anbindung der Testautomatisierung an den Entwicklungsprozess	58
3.4.1.	Anbindung an das Requirementsmanagement	59
3.4.2.	Anbindung an das Changemanagement	59
3.4.3.	Anbindung an das Versionsmanagement	59
4.	Konzept der Middleware	61
4.1.	Allgemeines Konzept	61
4.1.1.	Vorbetrachtung	62
4.1.2.	Anforderungen	65
4.1.3.	Modulkonzept	75
4.1.4.	Aufbau des Gesamtsystems	77
4.2.	Testbeschreibung	79
4.2.1.	Zielsetzung	79
4.2.2.	Anforderungen	79
4.2.3.	Lösungsvarianten	79
4.2.4.	Lösungskonzept	84
4.3.	Testausführung	85
4.3.1.	Zielsetzung	85
4.3.2.	Anforderungen	85
4.3.3.	Lösungsvarianten	85
4.3.4.	Lösungskonzept	87
4.4.	Testdokumentation	88
4.4.1.	Zielsetzung	88
4.4.2.	Anforderungen	88
4.4.3.	Lösungsvarianten	88
4.4.4.	Lösungskonzept	89
4.5.	Integration in den Entwicklungsprozess	90
4.5.1.	Zielsetzung	90
4.5.2.	Anforderungen	90
4.5.3.	Lösungskonzept	90
4.6.	Gesamtkonzept	92

5. Realisierung der Middleware	97
5.1. Systemdesign	97
5.2. Beispiel-Testszenario	98
5.3. Testbeschreibung	103
5.3.1. Vorbetrachtungen	103
5.3.2. Umsetzung des Lösungskonzeptes in die Softwarearchitektur . . .	103
5.3.3. Implementierung der Softwarearchitektur	104
5.3.4. Anwendung auf Beispiel-Testfälle	105
5.3.5. Umgesetzte Anforderungen	109
5.3.6. Anwendung des Moduls	112
5.4. Testausführung	113
5.4.1. Umsetzung des Lösungskonzeptes in die Softwarearchitektur . . .	113
5.4.2. Implementierung der Softwarearchitektur	117
5.4.3. Anwendung auf Beispiel-Testfälle	132
5.4.4. Umgesetzte Anforderungen	134
5.4.5. Anwendung des Moduls	137
5.5. Testdokumentation	139
5.5.1. Umsetzung des Lösungskonzeptes in die Softwarearchitektur . . .	139
5.5.2. Implementierung der Softwarearchitektur	139
5.5.3. Anwendung auf Beispiel-Testfälle	144
5.5.4. Umgesetzte Anforderungen	145
5.5.5. Anwendung des Moduls	148
5.6. Integration in den Entwicklungsprozess	149
5.6.1. Umsetzung des Lösungskonzeptes	149
5.6.2. Umgesetzte Anforderungen	151
6. Validierung und Evaluierung	153
6.1. Validierung des Interpreters	153
6.2. Hardware In The Loop-Anwendung	154
6.2.1. Gamma V-basierte Hardware In The Loop-Systeme	154
6.3. Anwendung der Einzelmodule	158
6.3.1. Verteilte Testerstellung und Testausführung	159
6.3.2. Anbindung der Testdokumentation an kommerzielle Testautomatisierungssysteme	159
7. Vergleich zu bestehenden Testautomatisierungssystemen	163
7.1. Vergleich des Integrationsaufwandes	163
7.2. Vergleich des Portierungsaufwandes bei dem Wechsel zwischen Testsystemen	164
7.2.1. Vergleich des Initialaufwandes	164
7.2.2. Vergleich des Implementierungsaufwandes der Testfälle	164

7.3. Vergleich des Implementierungsaufwandes beim Einsatz verschiedener Testbeschreibungsmethoden	165
7.3.1. Vergleich des Initialaufwandes	165
7.3.2. Vergleich des Implementierungsaufwandes der Testfälle	165
8. Zusammenfassung und Ausblick	167
8.1. Zusammenfassung	167
8.2. Ausblick	169
A. Eigene Veröffentlichungen	171
B. Literaturverzeichnis	173
C. Abbildungsverzeichnis	183
D. Tabellenverzeichnis	185
E. Listings	187
F. Lebenslauf	189

Abkürzungsverzeichnis

Abkürzung	Beschreibung
ADC	Analog-to-Digital-Converter
ArXML	AUTOSAR XML-Spezifikationen
ASAM	Association for Standardisation of Automation and Measuring Systems
AST	Abstract Syntax Tree
ATE	Automatic test equipment
ATML	Automatic Test Markup Language – IEEE 1671
Automotive SPICE	Ausprägung des Standards Software Process Improvement and Capability Determination – ISO/IEC 15504 für die den Automobilindustrie
AUTOSAR	AUTomotive Open System ARchitecture
AVB	Audio Video Bridging
BNF	Backus-Naur-Form
BSW	AUTOSAR Basis Software
CAN	Controller Area Network
CAN-FD	CAN with flexible Data Rate
CD	Coding / Decoding
CH	Component Handling
DATPG	Automated test program generator
DSL	Domain-specific language
DTIF	Standard for Digital Test Interchange Format – IEEE 1445
DUT	Device Under Test
ECU	Electronic Control Unit
ETSI	European Telecommunication Standards Institute
FET	Feldeffekttransistoren
Fibex	Field Bus Exchange Format
GFT	Graphical Format
HIL	Hardware In The Loop
I ² C	Inter-Integrated Circuit
IDE	integrierte Entwicklungsumgebung
KL15	Klemme15 – geschaltete Plusleitung vom Zündstartschalter nach DIN 72552

KL30	Klemme30 – Plusleitung direkt von der Batterie nach DIN 72552
KL31	Klemme31 – Minusleitung direkt von der Batterie nach DIN 72552
LIN	Local Interconnect Network
LVDS	Low Voltage Differential Signaling
MIL	Model In The Loop
modTF	Modularen Testautomatisierungs-Framework
MOST	Media Oriented Systems Transport
OEM	Original-Equipment-Manufacturer
OTX	Open Test Exchange – ISO 13209
PA	Platform Adapter
PDU	Protocol Data Unit
PIL	Prozessor In The Loop
PSF	Python Software Foundation
PSI5	Peripheral Sensor Interface 5
PWM	Pulsweitenmodulation
RAD	Rapid Application Development
ReqIF	Requirements Interchange Format - früher RIF
RTE	AUTOSAR Runtime Environment
SA	SUT Adapter
SENT	Single Edge Nibble Transmission
SIL	Software In The Loop
SoC	Systems on a Chip
SPI	Serial Peripheral Interface
SPICE	Software Process Improvement and Capability Determination – ISO/IEC 15504
STIL	Standard Test Interface Language – IEEE 1450
SUT	System Under Test
SW-C	AUTOSAR Software Component
TAP	The Test Anything Protocol
TCI	TTCN-3 Control Interface
TDD	Test-Driven Development
TE	TTCN-3 Executable
TET	Test Environment Toolkit
TL	Logging
TM	Test Management
TMC	Test Management and Control
TRI	TTCN-3 Runtime Interface
TTCN-3	Testing and Test Control Notation Version 3

UART	Universal Asynchronous Receiver/Transmitter
UML	Unified Modeling Language
W3C	World Wide Web Consortium
XCP	Universal Measurement and Calibration Protocol / eXtended Calibration Protocol
XML	Extensible Markup Language
XP	Extreme Programming

Danksagung

Die vorliegende Arbeit entstand im Rahmen meiner Tätigkeit an der Professur Schaltkreis- und Systementwurf der TU Chemnitz in Kooperation mit der iSyst Intelligente Systeme GmbH in Nürnberg. An dieser Stelle möchte ich mich bei den Helfern herzlich bedanken, ohne deren Unterstützung die Anfertigung dieser Arbeit in der Form nicht möglich gewesen wäre.

Einen wesentlichen Anteil an dieser Arbeit hat mein Betreuer Prof. Ulrich Heinkel. Mit seiner fachlichen und organisatorischen Unterstützung war die Möglichkeit der Promotion und der Forschung im Gebiet der Testautomatisierung möglich. Weiterhin möchte ich mich bei Dr. Erik Markert für die Unterstützung bei der Arbeit sowie bei fachlichen und organisatorischen Fragen bedanken. Darüber hinaus möchte ich Prof. Jürgen Teich für die Übernahme des zweiten Gutachtens danken.

Des Weiteren möchte ich der Geschäftsleitung der iSyst Intelligente Systeme GmbH – Christine Rauch, Daniel Heinrich und Prof. Hans Rauch – für die Unterstützung dieser Arbeit danken. Weiterhin danke ich Prof. Rauch, der mir bei Durchführung, Strukturierung und Ausarbeitung dieser Arbeit mit seinem Erfahrungsschatz zur Seite stand. Für die praktische Umsetzung dieser Arbeit möchte ich Jörg Tremmel für die Unterstützung mit seinem tiefen Wissen in Python, in grammatischen Fragen sowie für den Wissensaustausch und Diskussionen der Konzeptionierung wie der Umsetzung dieser Arbeit danken. Ebenfalls möchte ich Norman Franchi und Christian Heinz danken, welche durch ihren Wissensaustausch und Diskussionen im Rahmen des Förderprojektes ProTecT zur Entwicklung der Ideen dieser Arbeit beigetragen haben.

Darüber hinaus möchte ich meiner Familie und insbesondere meiner Frau für die Unterstützung und die gebrachten Opfer für diese Arbeit danken.

Die Realisierung des Frameworks wurde durch das Bundesministerium für Wirtschaft im Rahmen des ZIM-Projektes "ProTecT Embedded Systems" mit dem Förderkennzeichen KF2139704MS gefördert.

1. Einleitung

1.1. Motivation

Im Bereich der Automobilelektronik nimmt die Zahl der Funktionen der Steuergeräte mit jeder Fahrzeuggeneration zu. Neben der bloßen Anzahl der Funktionen steigt der Grad der Vernetzung mit immer mehr Bussystemen an. Kamen bisher Bussysteme mit niedrigen Bandbreiten, wie zum Beispiel K-Line, LIN oder CAN zum Einsatz, so werden heute immer häufiger FlexRay oder auch Ethernet [1] verwendet. Die steigende Funktionsvielfalt der Steuergeräte sowie die Anzahl und der Umfang der zwischen den Steuergeräten ausgetauschten Daten erfordern diese genannten Bussysteme mit höherer Bandbreite. Weiterhin führt die Einbindung von Steuergeräten mit Funkschnittstellen (Car2X-Kommunikation) zu einem zusätzlichen Anstieg der Komplexität und zu neuen Anforderungen an die Testsysteme und Testverfahren.

Die immer kürzeren Entwicklungszyklen sowie die immer größere Anzahl von Funktionen auch schon in frühen Entwicklungsmustern stellen vor allem die Testabteilungen vor immer neue Herausforderungen. Es muss in kurzer Zeit eine Vielzahl von Funktionen und Signalen teilweise auf unterschiedlichen Bussystemen geprüft werden. Dabei besteht für die Entwickler und Tester die Herausforderung, dass das Verhalten in der Basis-Software innerhalb des Steuergerätes mittels Zugriffsmöglichkeiten, wie bspw. Universal Measurement and Calibration Protocol / eXtended Calibration Protocol (XCP), geprüft werden muss. Zur Realisierung dieser Tests und zur Erreichung der geforderten Testabdeckung ist ein hoher Automatisierungsgrad, bei der Testerstellung wie auch bei der Testausführung, notwendig. Dabei muss die Wiederverwendbarkeit der Testfälle (Steuergeräte wie auch Bussystem übergreifend) gewährleistet sein. Neben diesen Whitebox-Tests können auch Blackbox-Tests zum Einsatz kommen, wie zum Beispiel für die Absicherung der Signalweiterleitung über ein Gateway, z. B. von FlexRay auf einen CAN-Bus.

Die aktuell im Markt erhältlichen Werkzeuge zur Erstellung und Ausführung von Tests unterstützen die Wiederverwendbarkeit von Testfällen nur unzureichend. Es fehlt die Unterstützung unterschiedlicher Testsysteme von verschiedenen Herstellern. Des Weiteren beschränkt sich die Einsatzmöglichkeit meist auf nur einzelne Stufen des Testprozesses, z.B. den Integrationstest. Für eine signifikante Steigerung der Effizienz ist die Wiederverwendbarkeit der Testfälle in Verbindung mit verschiedenen Testsystemen in unterschiedlichen Phasen des Testprozesses notwendig. Darüber hinaus muss eine fehlerfreie und verwertbare Weitergabe der Testfälle aus der Entwicklung in die Fertigung die Wiederverwendung ermöglichen. Die bisher vorherrschende Trennung zwischen

den entwicklungsbegleitenden Tests und den Fertigungstests, welche durch den Einsatz verschiedener Testsysteme gekennzeichnet sind, führt häufig zu einer aufwändigen Neuentwicklung von Testfällen mit neuem Fehlerpotential und zu einem Verlust an während der Entwicklung gesammelten Daten und Know-how [2].

1.2. Ziel der Arbeit

Im Rahmen dieser Arbeit soll die Machbarkeit einer modularen, portierbaren Middleware für die automatisierte Ausführung und Dokumentation von Software-Tests mit einer durchgehenden Nachverfolgbarkeit von den Anforderungen bis zu den Testergebnissen diskutiert werden. Der modulare Aufbau, die Ausrichtung auf eine einfache Portierbarkeit auf verschiedene Testsysteme (z. B. Hardware In The Loop-Tests, HIL) sowie die Erweiterbarkeit sollen den Einsatz in den verschiedenen Phasen des Testprozesses, während der Entwicklung und in der Fertigung, ermöglichen. Nur die geforderte Wiederverwendbarkeit bzw. Portierbarkeit von Testfällen führt zu der angestrebten breiten Einsatzmöglichkeit. Andererseits soll die Möglichkeit zum effektiven Einsatz von bereits erstellten Tests auf verschiedenen Testsystemen nicht nur in der Entwicklung sondern auch in der Fertigung realisiert werden. Durch eine größere Wiederverwendbarkeit von Testfällen sollen auch deren Qualität und durch die damit verbundene größere Anzahl von Testfällen auch die Testabdeckung signifikant gesteigert werden. Mit steigender Qualität der Testfälle ist auch eine höhere Reproduzierbarkeit der Tests zu realisieren.

Durch die Unabhängigkeit der Testfälle von den Testsystemen soll ein Einsatz bei verschiedenen Testabteilungen bei Zulieferern und OEMs ermöglicht werden. Vor allem die Verwendung des während der Entwicklung entstandenen Know-hows kann zu einer Verbesserung der Tests in der Produktion führen. Durch die Wiederverwendung von Testfällen ist eine Senkung des Aufwandes für die Fertigungstests möglich und eine höhere Testabdeckung, vor allem im Bereich der funktionalen Tests, realisierbar. Es ist dabei in dieser Arbeit ausdrücklich zu berücksichtigen, dass mit dem Ende der Entwicklung ein Produkt erst am Anfang seines Produktlebenszyklus steht und somit bis zum Ende seiner Produktlebensdauer zusätzliche weitere Testaufwände entstehen werden.

Ebenso ist die Einbindung der Middleware in das Requirementsmanagement und das Versionsmanagement notwendig. Die automatisierte Anbindung an Requirementsmanagement-Systeme ist für die sinnvolle Bestimmung der Abdeckung der Anforderungen durch den Test sowie für die Realisierung der notwendigen Nachverfolgbarkeit nötig. Dies ist für die Realisierung sicherheitskritischer Systeme zwingend erforderlich, wird jedoch durch verfügbare Tools kaum unterstützt. Die explizite Betrachtung der Anbindung des Tests bzw. der Testfälle an das Versionsmanagement-System ist notwendig, da dies ebenfalls für die Dokumentation der Nachverfolgbarkeit erforderlich ist. Dies wird bei vielen aktuell verfügbaren Systemen nicht oder nur unzureichend

betrachtet. Alles in allem sind von den Testsystemen und Testfällen die selben Anforderungen zu erfüllen, wie sie an die Entwicklung von Software für eingebettete Systeme gestellt werden.

Ziel dieser Arbeit ist die Prüfung der Machbarkeit und die beispielhafte Umsetzung einer modularen, portierbaren Middleware für die automatisierte Ausführung und die Dokumentation von Software-Tests. Ziel dieser Arbeit ist es nicht Möglichkeiten zur automatisierten Generierung von Tests zu erarbeiten, wie sie unter anderem in [3] diskutiert werden. Die hier entwickelte Middleware soll vielmehr eine flexible und portierbare Ausführungsumgebung für Software-Tests bieten. Dabei ist der Overhead für die Abstraktion vom Testsystem während der Testausführung zu minimieren. Die vorgeschlagene Implementierung der Middleware unterstützt eine manuelle wie auch eine automatisierte Erstellung von Testfällen. Gemäß der Aufgabenstellung liegen die Schwerpunkte dieser Arbeit bei der Ausführungsumgebung und der Dokumentation der Testfälle. Dabei liegt der Fokus auf dem funktionsorientierten Test von Software eingebetteter Systeme, beginnend beim Modultests in simulierter Umgebung bis hin zu Tests der Software auf dem realen, eingebetteten System.

1.3. Struktur der Arbeit

In den folgenden Abschnitten wird eine Übersicht über die Kapitel der Arbeit und ihren Inhalt gegeben.

In Kapitel 2 werden die notwendigen Grundlagen zu dieser Arbeit dargestellt. Zu Beginn wird der Aufbau typischer eingebettete Systeme und deren Komponenten, mit dem Fokus auf den Automotive-Bereich, dargestellt. Im Weiteren wird der Test- und Entwicklungsprozess mit den daraus resultierenden Anforderungen an die Testsysteme erläutert. Im Anschluss werden die Bereiche Software-Test und Fertigungsendtest gesondert besprochen.

In Kapitel 3 wird der Stand der Technik im Bezug auf die Inhalte dieser Arbeit dargestellt. Es werden dabei die verwendeten Testbeschreibungsmethoden sowie ihre Vor- und Nachteile erläutert. Im Anschluss werden verschiedene Testautomatisierungssysteme mit ihren Eigenschaften beschrieben. Weiterhin werden aktuelle Ansätze für die Testdokumentation sowie die Einbindung der Testautomatisierung in den Entwicklungsprozess dargestellt. In Kapitel 4 wird das allgemeine Konzept der modularen, portierbaren Middleware diskutiert. Dabei werden Lösungen für die verschiedenen Problemstellungen gegenübergestellt. Im Anschluss werden die Konzepte der einzelnen Module erläutert. Weiterführend wird das Konzept für die Integration in den Entwicklungsprozess dargestellt und darauf aufbauend das resultierende Gesamtkonzept präsentiert.

In Kapitel 5 wird die Realisierung der Middleware in Form des modularen Testautomatisierungs-Frameworks (modTF) beginnend bei der Systemarchitektur erläutert. Es werden dabei die realen Umsetzungen der einzelnen in Kapitel 4 konzeptionierten Module dargestellt und die erreichten technischen Möglichkeiten und Grenzen verdeutlicht.

In Kapitel 6 wird die Erprobung des modTF an verschiedenen Testszenarien gezeigt. Dabei werden Testfälle auf verschiedenen Hardware In The Loop-Systemen (HIL-Systemen) erläutert. Außerdem werden Möglichkeiten zur automatisierten Testfallgenerierung aufgezeigt. Außerdem kommen einzelne Module des modTF getrennt zum Einsatz, um die weitreichenden Möglichkeiten des modularen Ansatzes aufzuzeigen.

In Kapitel 7 werden die erreichbaren Ergebnisse des modTF mit den Möglichkeiten aktueller Testautomatisierungssysteme verglichen und Vor- und Nachteile dargestellt. Es werden dabei Gesichtspunkte wie Integrationsaufwand, Portierungsaufwand und Implementierungsaufwand betrachtet.

In Kapitel 8 werden die Ergebnisse dieser Arbeit zusammengefasst, und es wird ein Ausblick über die weitere Entwicklung gegeben.

2. Grundlagen

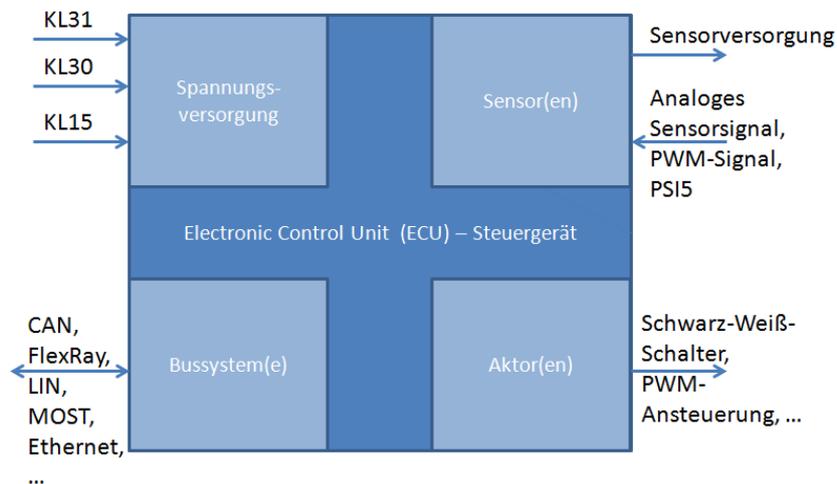
Es werden in diesem Kapitel die notwendigen Grundlagen für diese Arbeit dargestellt. Die Inhalte dienen der Herausarbeitung der betrachteten eingebetteten Systeme sowie der Rahmenbedingungen für den Test dieser und die mit dem Test verbundenen Anforderungen. Weiterhin wird eine Übersicht über das betrachtete Themenfeld gegeben.

2.1. Aufbau von eingebetteten Systemen

2.1.1. Übersicht

Eingebettete Systeme zeigen, bedingt durch ihren verbreiteten Einsatz für die Steuerung und Regelung, einen typischen Aufbau. Dieser Aufbau, wie er im industriellen- und automotive Umfeld typischerweise auftritt, ist beispielhaft in Abbildung 2.1 dargestellt. In der Abbildung werden dabei typische Bezeichnungen aus dem Automotive-Bereiche (z. B. Klemme30 (KL30), Klemme31 (KL31) und Klemme15 (KL15)) für die Spannungsversorgung verwendet [4].

Abbildung 2.1.: Schematische Darstellung eines Steuergerätes



Das eingebettete System, auch als Steuergerät bzw. Electronic Control Unit (ECU) bezeichnet, besteht im Kern aus einem Prozessor für die Verarbeitung der Daten sowie die Ausführung der Steuerungs- und Regelungsaufgaben. Zur Erfüllung der Aufgaben ist die Interaktion mit der Umwelt sowie mit anderen eingebetteten Systemen notwendig. Dafür

ist der Anschluss von Sensoren zur Messwerterfassung, von Aktoren zum Eingriff in Prozesse sowie von Bussystemen zur Kommunikation mit anderen eingebetteten Systemen grundlegend. Es ist eine Spannungsversorgung für das eingebettete System erforderlich.

2.1.2. Rechenkern

Als Rechenkern kommen meist Mikrocontroller in Form von Systems on a Chip (SoC) zum Einsatz. Diese bieten, neben dem eigentlichen Rechenkern, eine Auswahl der folgenden Komponenten:

- nicht flüchtigen Speicher (z. B. Flash)
- flüchtigen Speicher (z. B. SRAM)
- analoge und digitale Ein- und Ausgänge
- Anschlüsse für Bussysteme (z. B. SPI, I²C, CAN, FlexRay und Ethernet)
- spezielle Funktionseinheiten (z. B. Hardware Zufallszahlengeneratoren und Verschlüsselungseinheiten)

Es ist dabei festzustellen, dass nicht in allen Mikrocontrollern sämtliche genannten Komponenten vorhanden sind. Beispielsweise ist ein nicht flüchtiger Speicher auch extern, als getrennter IC, anschließbar.

Neben den technischen Eigenschaften spielen auch die Umweltbedingungen eine wichtige Rolle. Im Automotive-Bereich müssen die Mikrocontroller für Temperaturbereiche von -40 °C bis 85 °C bzw. teilweise bis 140 °C einsatzfähig sein. Dies hängt vom jeweiligen Verbauort des eingebetteten Systems ab [4].

Dabei kommen – je nach Sicherheitsanforderungen – unterschiedliche Mikrocontroller zum Einsatz. Vor allem im Bereich der sicherheitskritischen Systeme (z. B. Airbag und Bremssysteme) werden Mikrocontroller mit redundanten Ausführungseinheiten und redundanter Peripherie verwendet. Im Gegensatz dazu werden zum Beispiel im Infotainment-Bereich Mikrocontroller mit mehreren Kernen für die parallele Ausführung zur Steigerung der Rechenleistung eingesetzt.

Bedingt durch die große Bandbreite an Einsatzbereichen von eingebetteten Systemen alleine schon im Automotive-Bereich, gibt es eine annähernd unüberschaubare Anzahl von Mikrocontrollern bzw. SoCs. Im Bereich der sicherheitsrelevanten Systeme mit zeitkritischen Regelungsaufgaben kommen häufig Mikrocontroller aus der PowerPC-Familie von Freescale oder aus der Aurix/TriCore-Familie von Infineon zum Einsatz. Für Infotainment-Systeme werden aktuell ARM-basierte Mikrocontroller wie der Tegra 3 von Nvidia oder die iMX6-Familie von Freescale verwendet [5].

2.1.3. Aktorik

Für die Steuerung bzw. Regelung von Prozessen und Systemen ist es notwendig diese zu beeinflussen. Die Beeinflussung erfolgt über Aktoren. Aktoren können dabei Ventile, Pumpen, Antriebe oder Motoren sein. Es lassen sich, weitgehend unabhängig von den verwendeten Aktoren, drei Methoden der Ansteuerung unterscheiden.

Die erste und einfachste Methode ist ein Schalter mit den Zuständen Ein und Aus, auch als Schwarz/Weiß-Schalter bezeichnet. Dies wird meist über Feldeffekttransistoren (FET), in seltenen Fällen auch über Relais, umgesetzt. Es können damit die Aktoren nur ein- und ausgeschaltet werden. Eine Regelung ist damit nur sehr eingeschränkt möglich [4].

Die zweite und komplexere Methode ist die Ansteuerung über Pulsweitenmodulation (PWM). Dies wird typischerweise über Leistungsschalter in Form von FETs realisiert. Mittels dieser Methode ist eine weitreichende Regelung des Aktors (z. B. Drehzahl des Motors bzw. der Pumpe) möglich [4].

Die dritte Methode verwendet Analog-Ausgänge eines Mikrocontrollers, um die Ansteuerung des Aktors zu realisieren. Der Analogwert wird mittels einer Verstärkerschaltung (z. B. Operationsverstärker) direkt an den zu regelnden Aktor angelegt. Mittels des Analog-Ausgangs des Mikrocontrollers ist damit eine weitreichende Regelung des Aktors möglich.

2.1.4. Sensorik

Für die Steuerung bzw. Regelung von Prozessen und Systemen ist es notwendig, entsprechende Messwerte aufzunehmen. Dies erfolgt über Sensoren. Es kommen dabei verschiedenste Sensoren, angefangen bei einfachen Temperatur-Sensoren bis hin zu komplexen Sensorsystemen für Beschleunigungs- und Drehratenmessungen zum Einsatz. Es werden dabei auch sehr verschiedene Messmethoden eingesetzt. Aus Sicht des eingebetteten Systems ist die Schnittstelle des Sensors ein wichtiges Merkmal der Kategorisierung. Es können hier einfache Schnittstellen, wie Analogsignale oder PWM-Signale, sowie komplexere Schnittstellen, wie zum Beispiel Peripheral Sensor Interface 5 (PSI5) [6], Single Edge Nibble Transmission (SENT) [7] oder Serial Peripheral Interface (SPI) [8, S. 272] unterschieden werden.

Die einfachere Art der Schnittstellen kommt für die Aufnahme einzelner Werte, wie zum Beispiel einer Temperatur, zum Einsatz. Es werden dafür meist analoge Signale verwendet. Dabei versorgt das eingebettete System den Sensor mit Spannung. Der Sensor liefert eine Spannung bzw. einen Strom äquivalent zum Messwert. Über die ADC-Einheiten des Rechenkerns können diese Werte eingelesen und verarbeitet werden. Alternativ kann das Signal auch als PWM an das eingebettete System gemeldet werden. Die Pulsweite kodiert dabei den gemessenen Wert.

Im Rahmen der zunehmenden Komplexität eingebetteter Systeme, nimmt auch die Anzahl der zu messenden Werte zu. Aus diesem Grund kommen immer häufiger komplexe Sensoren, zum Beispiel für die Messung von Beschleunigungen und Drehraten, zum

Einsatz. Diese Sensoren stellen oftmals eigene Systeme mit Signalverarbeitung dar. Zum Anschluss dieser Art von Sensoren sind komplexere digitale Schnittstellen, wie PSI5 oder SENT, notwendig. Diese Schnittstellen bieten mit ihren Datenprotokollen, neben dem Zugang zu den Sensorwerten, auch Möglichkeiten, um Diagnose-Informationen der Sensoren zu erfassen. Dies ist im Bereich sicherheitsrelevanter Systeme notwendig.

2.1.5. Bussysteme

Für die Realisierung der komplexen Steuerungs- und Regelungsaufgaben in heutigen Fahrzeugen kommen verschiedene eingebettete Systeme, welche über unterschiedliche Bussysteme verbunden sind, zum Einsatz. Je nach den benötigten Eigenschaften, wie zum Beispiel der Bandbreite, der Anzahl der Busteilnehmer oder der Anforderung an die Übertragungssicherheit, kommen andere Bussysteme zum Einsatz.

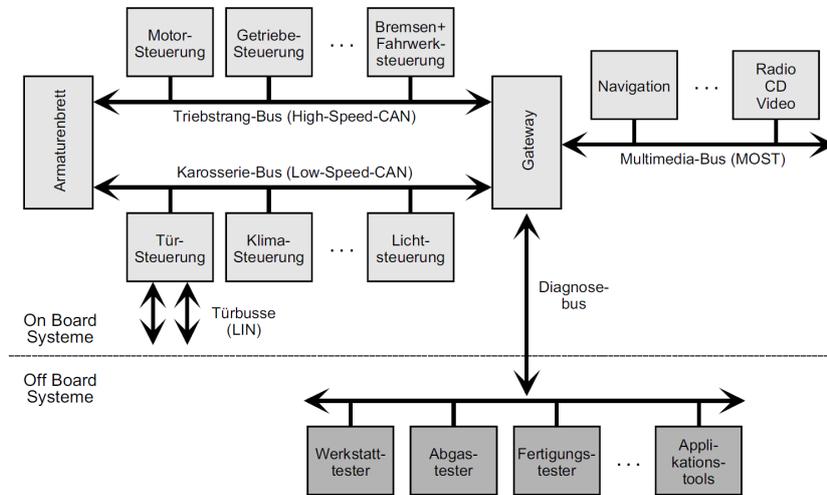
Für einfache Aufgaben mit geringer Bandbreite und wenigen Busteilnehmern kommen K-Line [9] oder Local Interconnect Network (LIN) [10] zum Einsatz. Beide Busse sind Eindraht-Busse und werden typischerweise über die UART-Schnittstelle von Mikrocontrollern in Verbindung mit einem K-Line- bzw. LIN-Transceiver realisiert. Die maximale Bitrate liegt für LIN bei 20 kbit/s und für K-Line bei 10,4 kbit/s. Beide Bussysteme werden immer häufiger durch leistungsfähigere Bussysteme, meist durch das Controller Area Network (CAN-Bus – ISO 11898) [11], verdrängt. Die größte Verbreitung in aktuellen Fahrzeugen besitzt der CAN-Bus. Er verwendet eine differenzielle Datenübertragung über 2 Leitungen und benötigt dazu einen speziellen CAN-Controller und CAN-Transceiver für die Realisierung. Er bietet mit bis zu 1 Mbit/s eine deutlich höhere Bandbreite als K-Line oder LIN. Des Weiteren können – je nach Transceiver – bis zu 120 Teilnehmer an einen CAN-Bus angeschlossen sein. Es kommen in heutigen Fahrzeugen meist mehrere CAN-Busse zum Einsatz, welche über entsprechende Gateways untereinander und mit anderen Bussystemen verbunden sind, wie dies beispielhaft in Abbildung 2.2 dargestellt ist [12].

In Bereichen mit höheren Anforderungen an die Bandbreite, wie zum Beispiel im Infotainment-Bereich oder bei der aktiven Fahrwerksregelung, wird der CAN-Bus durch FlexRay [13] oder Media Oriented Systems Transport (MOST) [14] ersetzt. FlexRay wurde dabei für den Bereich X-by-Wire entworfen und bietet eine besonders hohe Ausfallsicherheit. Es verwendet eine differenzielle Datenübertragung über 2 Leitungen und benötigt einen speziellen FlexRay-Controller und FlexRay-Transceiver für die Realisierung. Mit einer Übertragungsrate von 10 Mbit/s ist FlexRay entsprechend leistungsfähiger als CAN. Weiterhin besitzt FlexRay ein deterministisches Zeitverhalten.

MOST ist ein Bussystem, welches im Infotainment-Bereich zum Einsatz kommt. Es dient der Übertragung von Audio- und Video-, Sprach- und Datensignalen. MOST verwendet normalerweise Lichtwellenleiter zur Übertragung und benötigt ebenfalls spezielle MOST-Controller und MOST-Transceiver. Mit einer Bitrate von bis zum 150 Mbit/s bietet MOST auch gegenüber FlexRay eine deutlich erhöhte Bandbreite [15].

Neben den genannten Bussystemen kommen in Fahrzeugen noch spezielle Lösungen,

Abbildung 2.2.: Bussysteme mit Gateway im Fahrzeug [12, S. 1]



zum Beispiel für die Anbindung intelligenter Sensoren oder Kamerasysteme, zum Einsatz. Diese Lösungen sind meist als Punkt-zu-Punkt-Verbindungen, wie zum Beispiel Serial Peripheral Interface (SPI) und Inter-Integrated Circuit (I²C) für Sensoren oder Low Voltage Differential Signaling (LVDS) für Kamerasysteme, ausgelegt.

2.1.6. Spezielle Trends im Automotive-Bereich

In der Domäne der eingebetteten Systeme im Automotive-Bereich zeigen sich aktuell vor allem zwei Entwicklungstrends. Einerseits wird versucht, trotz immer weiter steigender Komplexität und Anzahl der Funktionen, die Zahl der eingebetteten Systeme zu reduzieren. Dabei werden immer mehr Funktionen auf ein Steuergerät integriert. Unterstützt durch die standardisierte Softwarestruktur, basierend auf AUTOSAR [16] und der damit verbundenen einfacheren Portierbarkeit der Funktionen, nimmt die Funktionsvielfalt des einzelnen eingebetteten Systems stark zu, bei extrem hohen und immer noch steigenden Anforderungen. Dies wiederum führt zu hohen Anforderungen an die Zuverlässigkeit und die Leistungsfähigkeit der einzelnen Systeme. Vor allem der Anspruch an die zur Verfügung stehende Rechenleistung nimmt dabei sehr stark zu. Dies führt zu einem immer schnelleren Wechsel hin zu neuen und leistungsfähigeren Mikrocontroller-Familien, wie dies an der Aurix-Familie von Infineon zusehen ist. Damit verbunden sind sehr hohe Anforderungen an die Weiterentwicklung bei der Softwareentwicklung und dem Test. Ebenso steigt mit der Komplexität auch die Notwendigkeit zur Integration weiterführender Diagnosefunktionalitäten, wie sie zum Beispiel in [17] dargestellt sind.

Als zweiter großer Trend ist die immer stärkere Vernetzung der einzelnen eingebetteten Systeme zu sehen. Die Notwendigkeit der Vernetzung steht ebenfalls im Zusammenhang mit der immer größer werdenden Anzahl von Funktionen. Bedingt durch die exponentiell

steigende Informationsmenge [13], müssen immer schnellere Bussysteme zum Einsatz kommen. Dabei ist vor allem die Verwendung von Ethernet – auf Basis von BroadR-Reach [18, 15] und AVB [17] – zu nennen, welches in den kommenden Jahren verstärkt in Fahrzeugen zum Einsatz kommen wird [19]. Mit der Erweiterung des CAN-Busses um CAN-FD und der damit verbundenen erhöhten Datenrate von bis zu 15 Mbit/s ist auch in diesem Bereich eine deutlich höhere Bandbreite zu erwarten. CAN-FD soll dabei in kommenden Fahrzeuggenerationen ein schnelleres Flashen des Steuergerätes und später auch eine schnellere Kommunikation der Steuergeräte untereinander ermöglichen [20, 21]. Bedingt durch die großen Datenmengen ist die Absicherung der Korrektheit der Datenkommunikation mittels Tests eine immer größere Herausforderung.

2.2. Test- und Entwicklungsprozess

Der Test ist integraler Bestandteil der Entwicklung eines eingebetteten Systems. Im folgenden Abschnitt werden verschiedene Modelle für den Entwicklungs- und Testprozess dargestellt. Diese Prozessmodelle bilden die Basis für alle Entwicklungs- und Testaktivitäten und stellen damit grundlegende Rahmenbedingungen für ein modulares Testautomatisierungssystem.

2.2.1. Test

Der Begriff Tests ist in verschiedenen Formen definiert. Die IEEE 610.12-1990 definiert Test wie folgt: "the process of operating a system or component under specified conditions, observing or recording the results and making an evaluation of some aspects of the system or component." [22]. Ziel ist es dabei das Fehlverhalten bzw. den Ausfall (engl. failure) eines Systems zu zeigen und damit Fehler (engl. faults) aufzudecken. Die Ursachen von Fehlern sind Irrtümer (engl. errors), welche Spezifikationsfehler oder Entwurfsfehler sein können. Ein Einsatz der Tests für den Bandend-Test und damit für die Ermittlung von Herstellungsfehlern bzw. Produktionsfehlern wird betrachtet [3, S. 5].

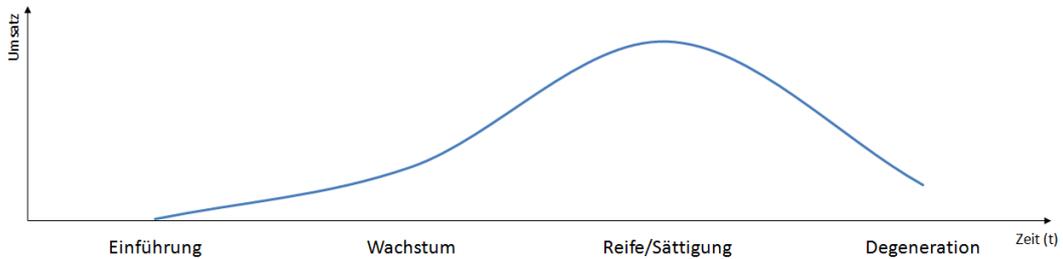
Im Rahmen dieser Arbeit wird der Begriff Test für den Test der Software (auch als Software-Test bezeichnet) eines eingebetteten Systems beginnend bei Modultests in simulierter Umgebung bis hin zu Tests der Software auf realen eingebetteten Systemen verwendet. Dabei wird von funktionsorientierten Testfällen ausgegangen. Die Erstellung der Testfälle kann dabei manuell oder automatisiert erfolgen.

2.2.2. Der Produktlebenszyklus

Unter dem Begriff Produktlebenszyklus sind zwei unterschiedliche Betrachtungen zu finden. Die erste Betrachtung spiegelt die betriebswirtschaftliche Sicht auf die Vermarktung eines Produktes wider. Es wird dabei die Umsatzentwicklung über die Zeit betrachtet, wie dies in Abbildung 2.3 dargestellt ist [23]. Diese Betrachtung soll hier nicht weiter

erläutert werden, da sie für den Test nur geringe Relevanz hat. Für die Einordnung des

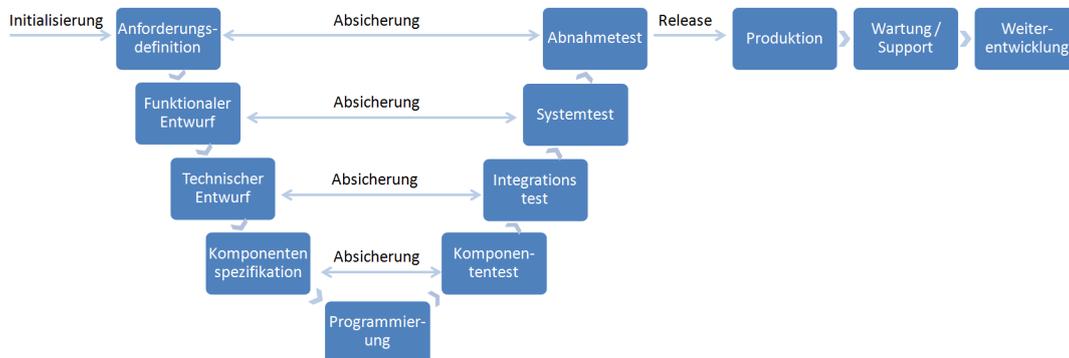
Abbildung 2.3.: Produktlebenszyklus nach Boston Consultin Group (BCG) [23]



Tests ist die zweite Betrachtung als Prozessmodell sinnvoller. In Abbildung 2.4 ist diese Art der Betrachtung, als einer Erweiterung des V-Modells nach Markus Rentschler, dargestellt [24]. Diese Darstellung gibt eine Übersicht über die gesamten Aufwände für Entwicklung und Produktion eines Produktes. Hierbei wird auch verdeutlicht, dass die Aufwände – auch die Testaufwände – nach Abschluss des Abnahmetests nicht unerheblich sind. Im Bezug auf den Test sind der Fertigungsendtest am Ende der Produktion sowie Wartung und Support zu berücksichtigen. Dies wird bei den sequentiellen wie auch bei den iterativ-inkrementellen Entwicklungsprozessen nicht betrachtet. Es ist daher sinnvoll, bei der Testplanung und Testumsetzung während der Entwicklung, auch die Produktion sowie Wartung und Support zu berücksichtigen, um Synergieeffekte nutzen zu können. Weiterhin kann aus dieser Betrachtung ebenfalls abgeleitet werden, dass der Test von Rückläufern aus dem Feld schon während des Entwicklungsprozesses berücksichtigt werden sollte.

Zusammenfassend betrachtet, ergeben sich aus den genannten Punkten Anforderungen bezüglich der Lebensdauer und Einsatzzeit von Testsystemen. Die Testsysteme – als Gesamtheit aus Software und Hardware – müssen über die gesamte Herstellungszeit eines Produktes zur Verfügung stehen, um vor allem Regressionstests zu ermöglichen.

Abbildung 2.4.: Produktlebenszyklus nach Markus Rentschler [24]



2.2.3. Sequentielle Entwicklungsprozesse

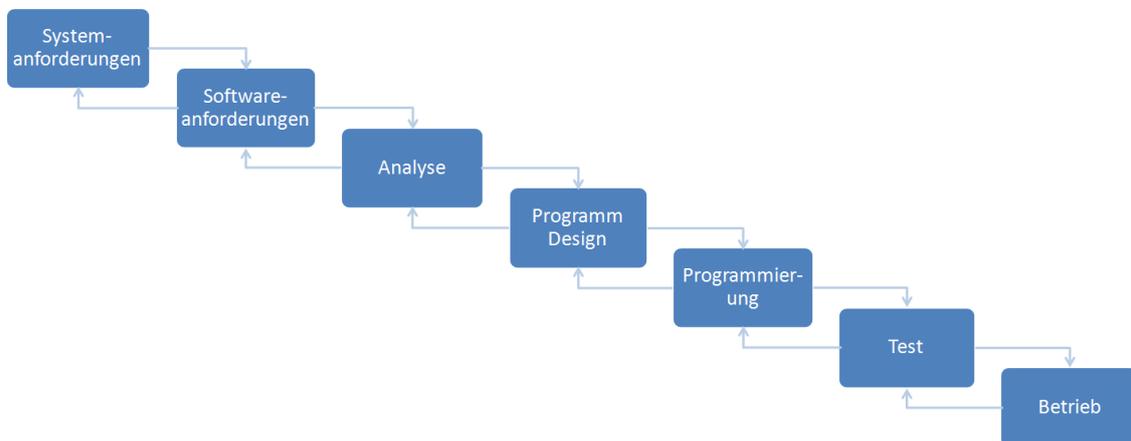
Bei den sequentiellen Modellen zu Entwicklungsprozessen wird dieser als sequenzielle Abfolge verschiedener Phasen betrachtet. Es gibt, je nach Modell, keine oder nur wenige Rückkopplungen zu vorhergehenden Phasen. Als Ergebnis des Durchlaufes des gesamten Prozesses steht das fertige Produkt. Es werden keine expliziten Prototypen oder verschiedene Versionsstände in den Modellen abgebildet.

2.2.3.1. Wasserfall-Modell

Das Wasserfall-Modell ist das erste grundlegende Entwicklungsmodell. Es geht von einer einfachen sequenziellen Abarbeitung der einzelnen Phasen aus, wie dies in Abbildung 2.5 dargestellt ist [25, S. 19]. Erst nach Abschluss einer Phase wird die nächste begonnen. Eine Rückkopplung ist nur zwischen benachbarten Phasen vorgesehen. In diesem Modell ist der Test eine Phase, welche einmalig am Ende der Entwicklung durchlaufen wird. Der Test wird dabei als Abnahmetest vor der Kundenauslieferung betrachtet. Eine entwicklungsbegleitende Absicherung ist im Wasserfall-Modell nicht vorgesehen.

Von Nachteil ist, dass weder Fehler noch Änderungen in den Anforderungen vorgesehen sind, da es keine Rückkopplungen zu diesen Phasen gibt. In der Praxis ist die nur schwer zu realisieren.

Abbildung 2.5.: Wasserfall-Modell [25, S. 19]



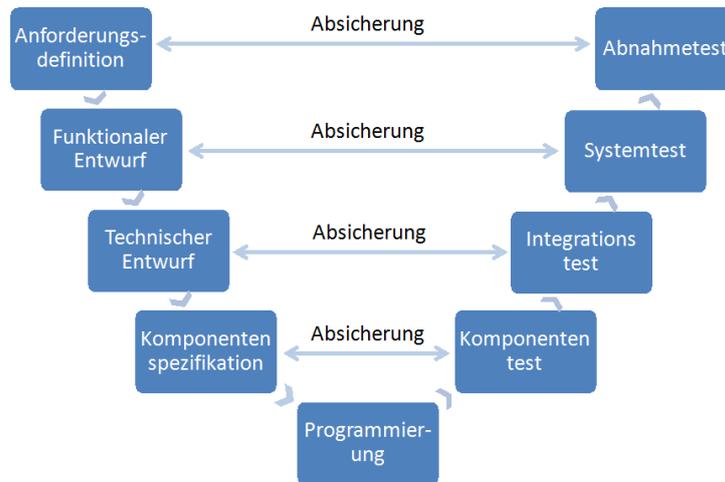
2.2.3.2. V-Modell

Das V-Modell ist als eine Weiterentwicklung des Wasserfall-Modelles zu betrachten. Im V-Modell werden die Entwicklungsphasen (absteigender Teil des V's) und die Testphasen (aufsteigender Teil des V's) getrennt. Jeder Entwicklungsphase ist eine Testphase zugeordnet, wie dies in Abbildung 2.6 dargestellt ist [26, S.33]. Im Gegensatz zum Wasserfall-Modell sind im V-Modell jederzeit Rückkopplungen in die anderen Phasen des Modelles

möglich. Damit ist es auch vorgesehen aus den Testphasen heraus Änderungen an den Anforderungen zu realisieren bzw. auf während der Entwicklung geänderte Anforderungen zu reagieren.

Durch die Gegenüberstellung der Entwicklungs- und Testphasen wird dem Test im V-Modell ein deutlich höheres Gewicht als im Wasserfall-Modell zugeordnet. Das V-Modell ist wohl das am weitesten verbreitete Entwicklungsmodell.

Abbildung 2.6.: V-Modell [26, S.33]

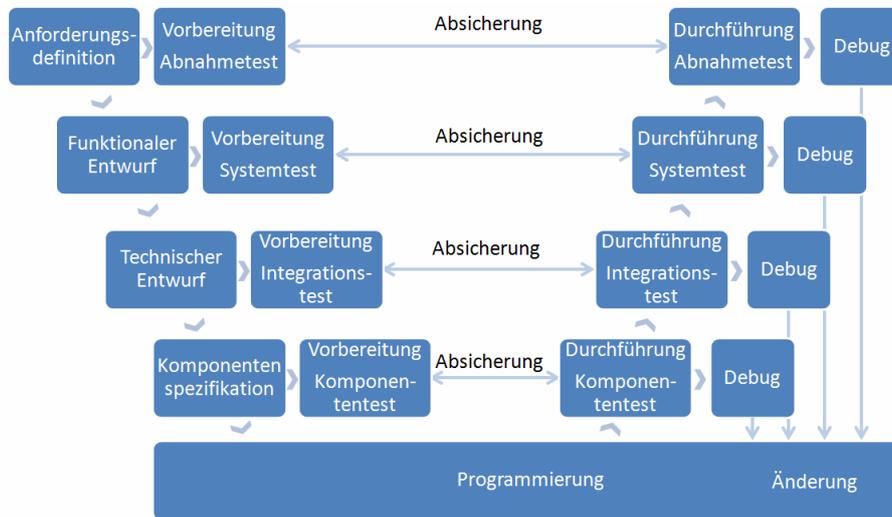


2.2.3.3. W-Modell (V²-Modell)

Das W-Modell – teilweise auch als V²-Modell bezeichnet – ist eine Weiterentwicklung des V-Modelles. Die Phasen der Entwicklung und des Tests sind dabei weitgehend unverändert geblieben, wie dies in Abbildung 2.7 dargestellt ist [26, S.33]. Das W-Modell behebt eine Darstellungsschwäche des V-Modelles, in dem die Spezifikation der Testfälle explizit den Entwurfsphasen zugeordnet wird. Dies war beim V-Modell nur implizit vorgesehen. Ebenfalls wird die Änderung auf Basis der Testergebnisse explizit dargestellt. Da die Darstellung nun je zwei absteigende und aufsteigende Teile besitzt, wird dieses Modell als W-Modell bezeichnet.

Im Hinblick auf den Test erfordert das W-Modell explizit den Start der Testaktivitäten parallel zum Start der Entwicklungsaktivitäten. Damit ist eine frühzeitige Berücksichtigung der Anforderungen für den Test in der Entwicklung gewährleistet. Weiterhin kann eine frühzeitige Bereitstellung der Testsysteme realisiert werden. Denn, bevor das erste Funktionsmodul getestet werden kann, müssen alle Komponenten des Testsystems getestet sein.

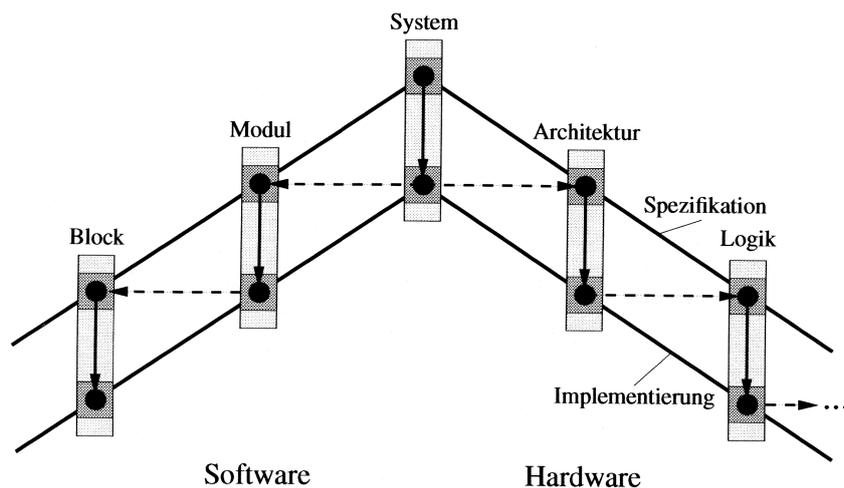
Abbildung 2.7.: W-Modell [26, S.35]



2.2.3.4. Doppel-Dach-Modell

Das Doppel-Dach-Modell ist ein Modell für den Entwurfs- und Verifikationsprozess. Gegenüber den bisher vorgestellten Modellen betrachtet dieses Modell Hardware und Software parallel. Es zielt dabei auf das Co-Design von Hardware und Software ab. Die Darstellung in Abbildung 2.8 [3, S.14] zeigt den Top-Down-Ansatz für die Entwurfsschritte. Dabei werden vergleichbare Stufen für Hardware und Software gegenübergestellt.

Abbildung 2.8.: Doppel-Dach-Modell [3, S.14]



Für die Verifikationsschritte wird der Pfad des Entwurfes in entgegengesetzter Richtung, beginnend auf Logik- bzw. Block-Ebene bis hin zu Systemebene, durchlaufen.

Die in der Praxis häufig vorkommende Trennung zwischen der Entwicklung von Software und Hardware erschwert den Einsatz dieses Modelles in der Praxis. Weiterführende Erläuterungen zu diesem Modell sind in [3, S.14ff] zu finden.

2.2.4. Iterativ-Inkrementelle Entwicklungsprozesse

Die iterativ-inkrementellen Modelle – welche oft auch als agile Entwicklungsmodelle bezeichnet werden – betrachten die Entwicklung als iterativen Prozess, in dem sich die Abfolge der gegebenen Phasen mehrfach wiederholt. Im Gegensatz zu dem sequenziellen Modellen ist damit eine Rückkopplung zu jeder Entwicklungsphase möglich und explizit vorgesehen. Als Ergebnis eines Durchlaufes des Modells entsteht eine Produktversion mit einem bestimmten Funktionsumfang. Mit jedem weiteren Durchlauf werden mehr Funktionen integriert, bis das angestrebte Produkt fertig ist. Die Modelle sehen damit explizit Prototypen und verschiedene Versionsstände des Produktes vor.

Aktuell werden dies für die Software-Entwicklung erarbeiteten Modelle bei der Erstellung der Software eingesetzt. Der Einsatz für die Entwicklung von Produkten aus Hardware, Software und Mechanik hat sich dabei als schwieriger erwiesen, da bei Hardware und Mechanik längere Entwicklungszyklen notwendig sind. Sie sind damit nicht in diesem Maße dynamisch, wie die Entwicklung von Software.

2.2.4.1. Spiralmodell

Das Spiralmodell unterteilt den Entwicklungsprozess in vier Phasen, welche je nach Quelle unterschiedlich bezeichnet sind, aber typischerweise mit Analyse, Entwurf, Programmierung und Test benannt werden. Diese Phasen werden wiederholt durchlaufen, wobei nach jedem Durchlauf – mit Abschluss des Tests – eine erweiterte, funktionsfähige Version des zu entwickelnden Produktes mit weiteren Funktionen bereit steht. Wie die Darstellung in Abbildung 2.9 verdeutlicht, steigen mit jedem Durchlauf der Phasen neben der Funktion auch die Kosten [27].

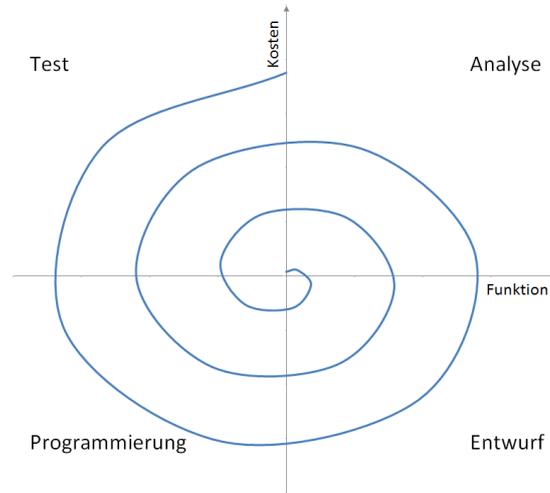
Der Vorteil dieses Entwicklungs- bzw. Vorgehensmodelles ist die Möglichkeit, dynamisch auf Änderungen der Anforderungen zu reagieren. Es sind frühzeitig funktionsfähige Prototypen verfügbar.

Als Anforderung an die Testsysteme ergibt sich die Notwendigkeit, dass die Testsysteme frühzeitig einsetzbar sind. Sie müssen schnell und flexibel an neue Anforderungen anpassbar sein.

2.2.4.2. Rapid Application Development

Das Modell des Rapid Application Development (RAD) bezieht sich explizit auf die Entwicklung von Software. Es werden dabei keine neuen Phasen der Entwicklung definiert. Vielmehr wird die Verkürzung der Entwicklungszeit, durch den vermehrten Einsatz von Werkzeugen im Entwicklungsprozess, angestrebt. Es ist dabei vorgesehen die Phasen

Abbildung 2.9.: Spiralmodell [27]

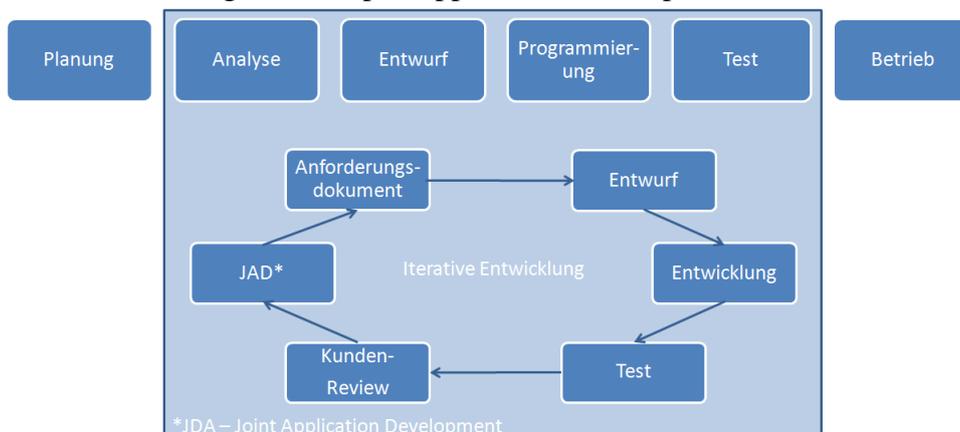


Analyse, Entwurf, Programmierung und Test mehrfach iterativ zu durchlaufen, wie dies in Abbildung 2.10 dargestellt ist [26, S.44].

Der Vorteil dieses Entwicklungs- bzw. Vorgehensmodelles ist ebenfalls die Möglichkeit dynamisch auf Änderungen der Anforderungen zu reagieren. Es sind frühzeitig funktionsfähige Prototypen verfügbar.

Die Entwicklung bzw. Anpassung der Testsysteme erfolgt während der Produktentwicklung. Es müssen frühzeitig Testsysteme zur Verfügung stehen und diese müssen flexible an neue Anforderungen anpassbar sein.

Abbildung 2.10.: Rapid Application Development [26, S.44]

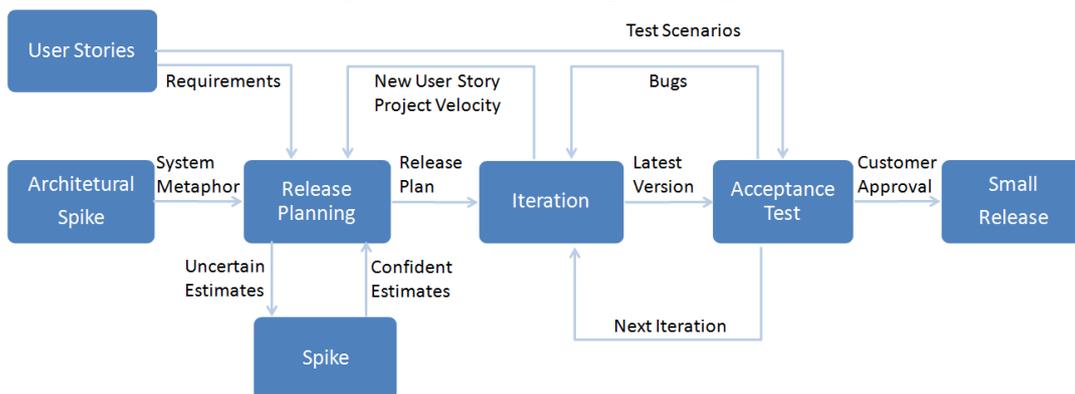


2.2.4.3. Extreme Programming

Das Modell des Extreme Programming (XP) ist, wie die anderen Iterativ-Inkrementelle Entwicklungsprozesse, ein Vertreter der *leichten Prozesse* [28], welche den Mehraufwand für Verwaltung und Dokumentation im Entwicklungsprozess minimieren sollen. Bei XP werden die Anforderungen auf Basis von *User Stories* erarbeitet. Die *User Stories* bilden dabei die Kundenwünsche ab und werden einerseits für die Anforderungen und andererseits für die Erstellung von Testscenarien für den Abnahmetest verwendet, wie dies in Abbildung 2.11 dargestellt ist [26, S.41]. Es werden typischerweise einzelne User Stories iterativ umgesetzt. Bedingt durch die frühzeitige Betrachtung wird dem Test im XP eine hervorgehobene Bedeutung zugeschrieben. Dabei kommen, neben dem Abnahmetest, vor allem Modultests zum Einsatz. Durch die inkrementelle Entwicklung und die damit verbundene häufige Durchführung der Tests, müssen diese in einem hohen Maße automatisiert werden. Es kommen dabei auch typischerweise Continuous Integration Systeme [29], für die kontinuierliche und automatisierte Übersetzung und Test der Software, zum Einsatz.

Durch den hohen Testfokus bietet dieses Entwicklungsmodell frühzeitig eine hohe Absicherung der entwickelnden Software. Da der Modultest und der Abnahmetest im Mittelpunkt der Betrachtung stehen, kann es bei umfangreichen Projekten zu Problemen in der Integration kommen. Integrationstests sind in diesem Modell nicht explizit vorgesehen. Aus den zuvor genannten Punkten ergibt sich die Anforderungen nach einem möglichst flexiblen Testsystem, welches die Realisierung von Modul- und Abnahmetests ermöglicht. Weiterhin ist ein hohes Maß an Automatisierung der Testabläufe notwendig. Ein manuelle Ausführung der Tests wäre zeitlich nicht möglich.

Abbildung 2.11.: Extreme Programming [26, S.41]



2.2.4.4. Test-Driven Development

Das Modell des Test-Driven Development (TDD) stellt den Test an den Anfang der Entwicklung. Im Gegensatz zu den anderen Entwicklungsmodellen werden erst die Tests auf

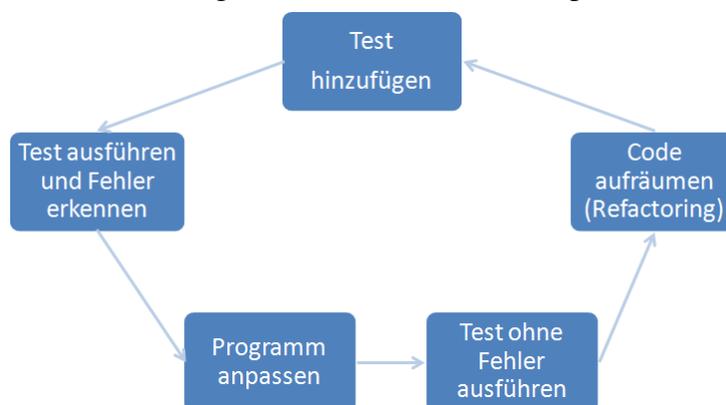
Basis der Anforderungen erstellt, wie dies in Abbildung 2.12 dargestellt ist. Nach der Erstellung der Tests wird die eigentliche Funktion implementiert und gegen die vorher erstellten Tests getestet. Es werden dabei iterativ einzelne Softwaremodule bzw. Funktionen umgesetzt. Dieses iterative Vorgehen wird solange wiederholt, bis alle Funktionen umgesetzt sind. Im Zentrum dieses Modelles steht der Modultest. Es wird dabei eine möglichst vollständige Automatisierung der Tests angestrebt.

Das Modell lässt sich auch auf Systemtests erweitern. Dabei werden die Systemtests auf Basis der Anforderungen vor der Erstellung des Systems spezifiziert und implementiert [30].

Der Vorteil dieses Modelles ist die frühzeitige hohe Testabdeckung. Wobei sich die Testabdeckung auf die Prüfung der getesteten Anforderungen gegenüber allen Anforderungen bezieht. Durch die Erstellung der Tests vor den eigentlichen Funktionen werden auch frühzeitig Unklarheiten oder Fehler in den Anforderungen erkannt und können noch vor der Implementierung der eigentlichen Funktion geklärt werden.

Aus den zuvor genannten Punkten ergeben sich die Anforderungen für ein möglichst flexibles Testsystem, welches die Realisierung von Modul- und Systemtests ermöglicht. Weiterhin ist ein hohes Maß an Automatisierung der Testabläufe notwendig, um die nötige Anzahl der Regressionstests durchführen zu können.

Abbildung 2.12.: Test-Driven Development



2.2.4.5. Scrum

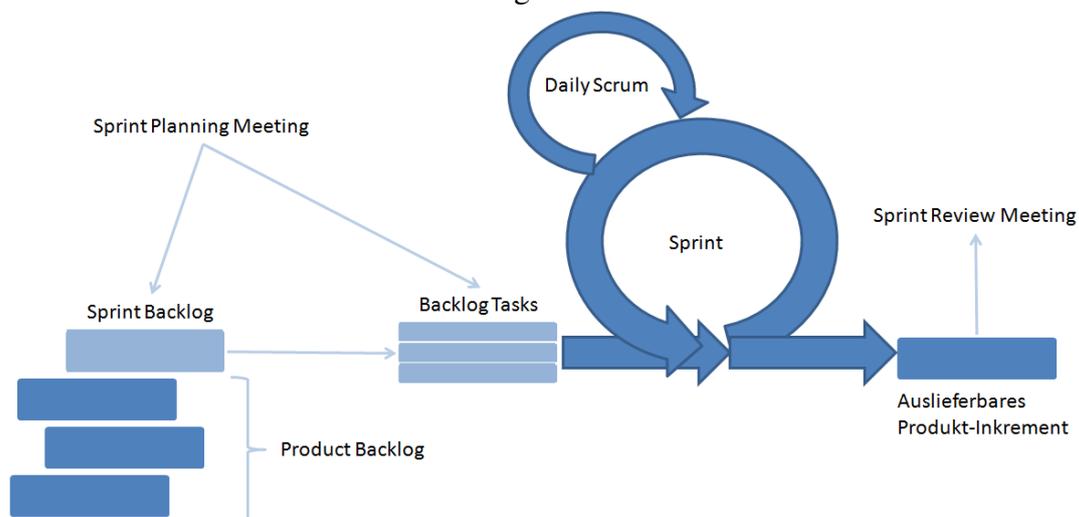
Das Modell des Scrum ist ein iteratives Modell, welches davon ausgeht, dass Entwicklungsprojekte zu komplex sind, um von Beginn an komplett planbar zu sein. Die Änderungen und Neupriorisierung von Anforderungen stellt einen zentralen Punkt bei Scrum dar. Als Basis für den Prozess dient das *Product Backlog*, welches die aktuellen Anforderungen und ihre Prioritäten enthält. Auf dieser Basis wird für jeden iterativen Durchlauf – auch Sprint genannt – ein *Sprint Backlog* mit den zu implementierenden Anforderungen erstellt. Während des Sprints, welcher typischerweise zwischen 2 und 4 Wochen dauert,

werden die Anforderungen implementiert und die zugehörigen Tests erstellt und durchgeführt. Am Ende des Sprints sollten die Anforderungen aus dem *Sprint Backlog* umgesetzt sein. Danach erfolgt die Planung des nächsten Sprints. Bei der Durchführung des Sprints werden die Arbeitspakete täglich in einem *Daily Scrum* besprochen und abgestimmt. Der Ablauf ist in Abbildung 2.13 dargestellt [31].

Der Vorteil vom Scrum ist, dass nach jedem Sprint eine lauffähige und getestete Version der Produktes zur Verfügung steht. Durch das hohe Maß an Kommunikation im Entwicklungsteam – durch das *Daily Scrum* – können Probleme gelöst sowie unklare Anforderungen erkannt und beseitigt werden.

Aus den zuvor genannten Punkten ergibt sich die Notwendigkeit eines möglichst flexiblen Testsystem, welches die Realisierung von Modul- über Integrations- bis zu Systemtests erlaubt. Weiterhin ist ein hohes Maß an Automatisierung der Testabläufe notwendig, um die notwendige Anzahl der Regressionstests durchführen zu können.

Abbildung 2.13.: Scrum

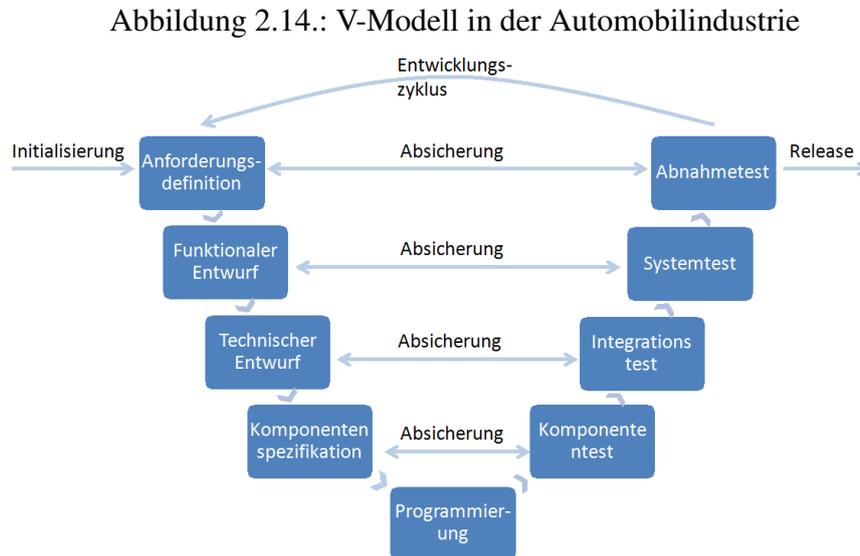


2.2.5. Entwicklung im Automotive-Bereich

Im Automotive-Bereich werden die Entwicklungsprozesse am V-Modell ausgerichtet. Dies ist unter anderem für die Erfüllung der Anforderungen nach Automotive SPICE [32, 33] notwendig. Automotive SPICE fordert das V-Modell zwar nicht explizit, jedoch die gesetzte Anforderung an den zu bewertenden Entwicklungsprozess sind an das V-Modell angelehnt.

Bedingt durch die hohe und stetig steigende Komplexität der Funktionen der Systeme im Fahrzeug ist ein iteratives Vorgehen notwendig. Es werden dabei typischerweise verschiedene Musterstände mit stetig zunehmender Funktion entwickelt und getestet. Die einzelnen Entwicklungszyklen für jeden neuen Musterstand werden dabei entsprechend des

V-Modelles durchgeführt. Damit ist im Automotive-Bereich eine Mischung aus sequenziellen V-Modell und iterativ-inkrementellen Prozess aktueller Stand der Technik. In Abbildung 2.14 ist dieses Vorgehen dargestellt. Vor allem die Synchronisation zwischen der



Entwicklung von Hardware, Software und Mechanik stellt dabei große Anforderungen an den Entwicklungsprozess. Da die Softwareentwicklung typischerweise schnelleren Entwicklungszyklen unterliegt als die Hardware und die Mechanik, ist die Sicherstellung der Kompatibilität zwischen verschiedenen Software-, Hardware- und Mechanik-Versionen eine große Herausforderung. Die Entwicklung der Software, Hardware und Mechanik verläuft in der Praxis in getrennten Teams bzw. in getrennten Abteilungen. Dazu kommt meist noch eine Anzahl verschiedener Varianten eines Systems, so dass ein gesteigerter Aufwand in das Versions- und Varianten-Management investiert werden muss. Weiterhin ist die Entwicklung immer komplexerer Systeme aus mehreren ECUs und deren Test aus System Sicht mit einem hohen Aufwand verbunden.

Neben der Entwicklung einzelner ECUs bzw. von Systemen aus mehreren ECUs, welche typischerweise bei den Zulieferern stattfindet, muss die Entwicklung und der Test des gesamten Fahrzeuges überwacht und durchgeführt werden. Dieser Prozess wird durch den Fahrzeughersteller, der auch als Original-Equipment-Manufacturer (OEM) bezeichnet wird, realisiert. Dabei kommt ebenfalls das V-Modell zum Einsatz.

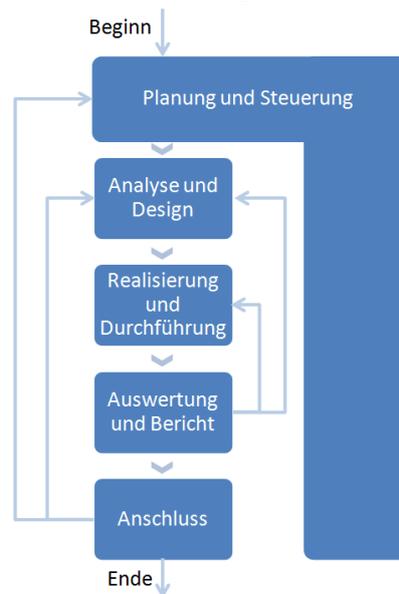
Zusammenfassend ist damit festzustellen, dass bei der Entwicklung im Automotive-Bereich ein vielfältiges Zusammenspiel zwischen den Entwicklungsprozessen des OEM und des Zulieferers realisiert werden muss. Neben diesem Zusammenspiel ist auch die Synchronisation zwischen der Hardware, Software und Mechanik Entwicklung für die einzelnen Komponenten, sowie für das gesamte Fahrzeug eine sehr komplexe Aufgabenstellung. Diese komplexen Entwicklungsprojekte sind nur durch die Verwendung einer breiten und fortschrittlichen Tool-Unterstützung, z. B. durch Anforderungsmanage-

ment, Versionsmanagement und Test Tools, zu bewältigen. Weiterhin werden fortschrittliche Entwicklungsmethoden, wie zum Beispiel Hardware-Software-Co-Design, zumindest punktuell eingesetzt [34].

2.2.6. Der fundamentale Testprozess

Der fundamentale Testprozess nach ISTQB [25, S.18-20] ist unabhängig von den verschiedenen Entwicklungsmodellen zu betrachten. Er beschreibt das Vorgehen für den Test, beginnend bei der Testplanung über die Testdurchführung bis zur Testauswertung und dem Testabschluss. Dies ist in Abbildung 2.15 dargestellt. Das Vorgehen lässt sich dabei in die verschiedenen Entwicklungsmodelle integrieren. Je nach Modell sind die Testumfangs und die Anzahl der Testdurchführungen unterschiedlich. In der praktischen Umsetzung ist damit eine Abstimmung zwischen dem Entwicklungs- und dem Testprozess notwendig.

Abbildung 2.15.: Fundamentalen Testprozesses nach ISTQB [25, S.20]



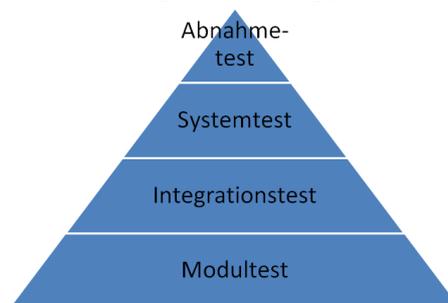
Dieser Prozess bietet den Vorteil, dass die Aufgaben und notwendigen Tätigkeiten sowie die zugehörigen Techniken für den Test präziser beschrieben sind, als es in den Entwicklungsmodellen der Fall ist. Er beschreibt dabei ein deutlich prozessorientiertes und detaillierteres Vorgehen, als es die Entwicklungsprozesse tun.

2.2.7. Test im Automotive-Bereich

Der Test im Automotive-Bereich ist stark mit dem Entwicklungsprozess verbunden. Wie in Abschnitt 2.2.5 beschrieben, erfolgt die Entwicklung im Automotive-Bereich entspre-

chend dem V-Modell. Daraus ergibt sich die grundlegende Unterteilung des Testprozesses in Modultest, Integrationstest, Systemtest und Abnahmetest. Die Abfolge der Tests wird häufig in Form einer Pyramide, wie dies in Abbildung 2.16 veranschaulicht ist, dargestellt. Im Automotive-Bereich fügen sich, beim Übergang vom Zulieferer zum OEM, mehrere dieser Pyramiden aufeinander. Dies verdeutlicht den hohen Testaufwand, den die Entwicklung eines komplexen Systems, wie einem Auto, benötigt. Diese Begriffe bezeichnen dabei keinen absoluten Inhalt, da der Testfokus je nach Entwicklungsebene (z. B. Einzelne ECU, Teilsystem oder Fahrzeug) unterschiedlich ist. Auf Seiten des Zulieferers bezieht sich der Modultest auf eine Modul der ECU. Dabei steht der Fokus auf der Software und damit auf einem Softwaremodul. Diese Tests erfolgen mit spezieller Software, wie zum Beispiel Tessy [35]. Für den OEM ist der Modultest der Test eines Teilsystems bzw. einer ECU, nach Erhalt vom Zulieferer.

Abbildung 2.16.: Testpyramide



Beim Integrationstest des Zulieferers wird typischerweise das Zusammenspiel der einzelnen Software-Komponenten in einer simulierten Umgebung (z. B. SIL oder HIL) getestet. Auf Seiten des OEM hingegen wird das Zusammenspiel einzelner ECUs bzw. Teilsysteme getestet. Dies erfolgt in simulierten Umgebungen (z. B. HIL).

Der Systemtest beim Zulieferer umfasst den Test der ECU bzw. des Teilsystems mit allen realen Komponenten – Hardware, Software und Mechanik. Dabei wird die Erfüllung der Anforderungen des OEMs geprüft. Diese Tests erfolgen auf HIL-Prüfständen. Für den OEM bedeutet Systemtest den Test der Elektronik und der damit verbundenen elektromechanischen Komponenten des Fahrzeuges. Dies wird durch Erprobungsfahrten realisiert. Der Abnahmetest für den Zulieferer ist oft identisch mit dem Modultest des OEMs. Es wird dabei – meist in einer simulierten Umgebung wie einem HIL-System – die Funktion der gelieferten Komponente getestet.

Neben der dargestellten, hohen Komplexität und dem Testaufwand stellen auch enge Zeitpläne große Anforderungen an den Test. In der Praxis stehen nur wenige Tage für den abschließenden Test eines neuen Releases einer ECU zur Verfügung. Es werden zwar entwicklungsbegleitend immer wieder Tests durchgeführt, aber der Auslieferungsstand der ECU für ein Release wird erst spät fertig gestellt. Daher ist es notwendig, ein hohes Maß an Automatisierung der Tests zu realisieren, um die notwendige Testabdeckung bezüglich der Anforderungen vor der Auslieferung zu erreichen. Um dies zu ermöglichen, sind

neben Testsystemen für den automatisierten Test (z. B. SIL, MIL und HIL) auch entsprechende Testautomatisierungs-Software zwingend erforderlich. Durch die zuvor genannten Rahmenbedingungen ergeben sich verschiedene Anforderungen an das Testsystem:

1. Die Anbindung an das Requirements- und Versionsmanagement-System. Bedingt durch die große Anzahl von Anforderungen sowie die notwendige Nachverfolgbarkeit der Tests und der Testabdeckung der Anforderungen, ist ein automatisierter Abgleich der Testergebnisse notwendig. Durch die Anzahl der Softwarevarianten ist eine Integration der Tests mit dem Versionsmanagement-System unabdingbar.
2. Die Austauschbarkeit von Testfällen zwischen Zulieferer und OEM. Dies ist vor allem bei den Integartions- und Abnahmetests notwendig. Wobei der Austausch ausführbarer Testfälle eine drastische Reduzierung des Aufwandes gegenüber dem Austausch von Testspezifikationen ermöglicht.
3. Die unterschiedlichen zeitlichen Anforderungen an das Testsystem und die Testausführung. Die hardwarenahen Tests beim Zulieferer erfordern Testabläufe mit zeitlichen Auflösung von wenigen ms. Dies wird durch Testsysteme mit einer garantier- ten, deterministischen Ausführung (häufig mit einer Wiederholung der Ausführung im Raster von 1 ms). Die Tests bei den OEMs werden aus der Sicht des Fahrers, und damit mit für Menschen wahrnehmbaren zeitlichen Auflösung von 200 ms und größer, durchgeführt.
4. Eine unterschiedliche Abstraktion der Testfälle. Es müssen sich alle Testfälle, beginnend beim Modultest, welcher einzelne Signale bzw. Variablen prüft, bis hin zum Abnahmetest, welcher das komplexe Verhalten eines Steuergerätes testet, einfach und übersichtlich abbilden lassen.

2.2.8. Spezielle Trends im Automotive-Bereich

Der Entwicklungs- und Testprozess im Automotive-Bereich wird vor allem durch den stark steigenden Funktionsumfang und die immer kürzeren Entwicklungszeiten geprägt [36]. Um diesen Herausforderungen entgegen zu treten, wird verstärkt auf die Standardisierung der Systeme und der Prozesse Wert gelegt. Dabei ist auf der Seite der Entwicklung AUTOSAR als standardisierter Baukasten für die Steuergeräte-Software zu nennen. Ebenfalls auf Basis des AUTOSAR-Standards wird ein immer stärkerer Einsatz von Entwicklungswerkzeugen, zur Beschleunigung der Entwicklung, vorangetrieben. Auf Seiten des Tests ist ebenfalls ein verstärkter Einsatz von Tools, wie dies in Abschnitt 3.2 näher erläutert wird, zu verzeichnen.

Basierend auf der Struktur des AUTOSAR-Standards findet eine getrennte Entwicklung der Software-Komponenten der Steuergeräte statt. Dabei wird die Basis-Software – auch LowLevel-Software genannt – durch den Zulieferer entwickelt bzw. konfiguriert. Dieser Teil der Software umfasst auch das AUTOSAR Runtime Environment (RTE), welches

die Schnittstelle zu den Software-Komponenten – auch HighLevel-Software genannt – bildet. Diese werden meist durch den OEM bereit gestellt. Hierbei trägt der Zulieferer immer häufiger die Verantwortung für die Integration der Software-Komponenten und die Absicherung der Gesamt-Software bestehend aus Basis-Software mit RTE und Software-Komponenten. Dies führt zu gesteigerten Testanforderungen beim Zulieferer.

Zusammen wird eine weitere Vereinheitlichung der Entwicklungsprozesse angestrebt. Dabei steht die bessere Abstimmung zwischen Software-, Hardware- und Mechanik-Entwicklung im Mittelpunkt. Es wird die Betrachtung von den einzelnen Komponenten hin zum System verlagert. Dies wird auch durch die Ausrichtung des Automotive SPICE Standard von der Software zum System verdeutlicht. Ziel dieser Maßnahmen ist eine Verkürzung der Entwicklungszeit und damit eine Verringerung der Entwicklungskosten.

Beispielhaft sei hier auch die Arbeit [37] zu nennen, welche das Ziel verfolgt eine bessere Abstimmung zwischen IC-Entwicklung und Fahrzeugentwicklung zu schaffen.

2.3. Software-Test

Der Software-Test, im Sinne der Beschreibung in Abschnitt 2.2.1, stellt bei der Entwicklung und bei dem Test heutiger eingebetteter Systeme einen zentralen Punkt dar. Eine grundlegende Definition des Testbegriffes für diese Arbeit ist in Abschnitt 2.2.1 zu finden. Bedingt durch den stark steigenden Umfang der Software in eingebetteten Systemen – wie zum Beispiel in automotive ECUs – steigt der Anteil der Software-Tests am Gesamttestaufwand [38]. Ziel des Software-Tests ist die Sicherstellung einer möglichst fehlerfreien Software des eingebetteten Systems.

Auf den verschiedenen Teststufen – Modultest, Integrationstest, Systemtest und Abnahmetest, wie sie in Kapitel 2.2.7 erläutert wurden – kommen dabei verschiedene Testmethoden und Testsysteme zum Einsatz. In den folgenden Abschnitten sollen die verschiedenen Arten des Software-Tests, mit dem Fokus auf automatisierte Testausführung in simulierter Umgebung, dargestellt werden. Neben diesen Testarten werden in der Praxis noch weitere Systeme, z. B. für den Softwaremodultest, eingesetzt.

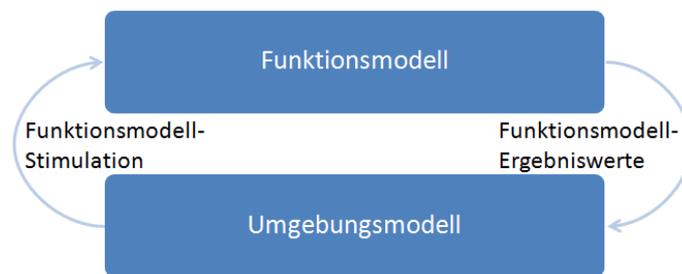
2.3.1. Arten des Software-Tests

Für die Anwendung dieser Testmethoden im Bereich der Entwicklung eingebetteter Systeme ist es notwendig, eine ausführbare und damit testbare Version des zu testenden Systems zu realisieren. Dies kann auf verschiedene Arten und in unterschiedlichen Stadien der Entwicklung erfolgen. Dabei kommt in jedem Fall eine Umgebungssimulation für das zu testende eingebettete System zum Einsatz. In den folgenden Abschnitten werden die verschiedenen Ausprägungen dieses X In The Loop-Ansatzes dargestellt.

2.3.1.1. Model In The Loop

Der Model In The Loop-Test (MIL-Test) kommt in einem frühen Entwicklungsstadium zum Einsatz. Es wird dabei ein Modell – beispielsweise unter der Verwendung von Matlab/Simulink oder ETAS ASCET – der Funktion des eingebetteten Systems erstellt. Weiterhin wird ein Modell der Umgebung erstellt, welches noch sehr einfach gehalten ist. Beispielsweise werden Sensoren und Aktoren durch einfache Modelle realisiert. Für die Testdurchführung werden nun das Funktionsmodell des eingebetteten Systems und das Umgebungsmodell verbunden – wie dies in Abbildung 2.17 veranschaulicht ist – und ausgeführt. Dies begründet auch den Namen Model In The Loop. Die Simulation erfolgt auf Desktop-Rechnern. Es kommt noch keine reale Hardware des eingebetteten Systems zum Einsatz.

Abbildung 2.17.: Model In The Loop – Verbindung des Funktions- und Umgebungsmodelles



MIL-Tests ermöglichen die frühzeitige Prüfung der logischen Funktion eines eingebetteten Systems. Es dient vor allem der Prüfung der Anforderungen und der Erkennung von Fehlern in der Spezifikation. Weiterhin ist es möglich auf Basis des Funktionsmodelles Entwurfsentscheidungen – z. B. über die Aufteilung der Funktionen in Hard- und Software – für das eingebettete System zu treffen. Außerdem lassen sich auf Basis der Tests Pfadüberdeckungen und damit eine Testabdeckung auf Modellebene ermitteln.

Der MIL-Test ermöglicht keine Aussagen über die Ausführungen auf der späteren Hardware. Auch ist ein Test von hardwarenahen Regelungen nur schwer oder gar nicht möglich.

2.3.1.2. Software In The Loop

Der Software In The Loop-Test (SIL-Test) folgt auf den MIL-Test. Für den SIL-Test wird die Software des eingebetteten Systems verwendet. Diese kann aus dem Funktionsmodell des MIL-Tests oder manuell erstellt worden sein. Es kommt hier verbreitet C als Programmiersprache zum Einsatz, da diese Sprache als Standard für die Softwareentwicklung bei Steuergeräten anzusehen ist. Das Umgebungsmodell entspricht dem des MIL-Tests. Je nach Reifegrad der Software und der Fehlererkennungsmechanismen kann es notwendig

sein, die Simulation der Aktoren und Sensoren zu verfeinern. Die Ausführung von Software und Umgebungsmodell erfolgt ebenfalls auf Desktop-Rechnern. Es kommt noch keine reale Hardware des eingebetteten Systems zum Einsatz.

Der SIL-Test ermöglicht einen frühzeitigen Test der Software des eingebetteten Systems. Es ist mittels Whitebox-Tests eine weitreichende Prüfung der logischen Funktion des eingebetteten Systems möglich. Die Bestimmung der Code-Abdeckung durch den Test ist in dieser Umgebung einfach möglich.

Mit dem SIL-Test ist es jedoch nicht möglich, Aussagen über das Echtzeitverhalten auf der realen Hardware zu treffen.

2.3.1.3. Prozessor In The Loop

Der Prozessor In The Loop-Test (PIL-Test) folgt typischerweise dem SIL-Test. Dabei wird die Software des eingebetteten Systems auf dem realen Prozessor der Ziel-Hardware ausgeführt. Es werden dabei Evaluation Boards verwendet, welche den Zugriff auf alle Pins des Prozessors erlauben. Das Evaluation Board wird mit einem Simulationsrechner verbunden, auf welchem das Umgebungsmodell zyklisch – mit typischen Wiederholungszeiten von 1 ms – ausgeführt wird.

Der PIL-Test erlaubt die Prüfung der Software unter Echtzeitbedingungen. Es sind damit nun Aussagen über das Laufzeitverhalten möglich. Weiterhin ist der Test hardwarenaher Softwareteile und Regelungen realisierbar. Durch den direkten Zugriff auf die Pins des Prozessors ist auch der weitreichende Test von Fehlererkennungsmechanismen, wie zum Beispiel redundante Messungen in sicherheitskritischen Systemen, möglich. Dies ist auf der realen Hardware oft wegen Schutzbeschaltungen nicht vollständig umsetzbar. Die Tests sind als Blackbox-Tests angelegt, da eine direkte Beobachtung des Programmablaufes nicht ohne weiteres möglich ist. Mittels der heutigen Debug-Schnittstelle der Mikrocontroller bzw. über das Protokoll XCP ist ein Zugriff in die Software während der Ausführung möglich. Damit sind auch Greybox- und Whitebox-Tests umsetzbar.

Mit dem PIL-Test ist der Test der Integration mit der realen Peripherie nicht möglich. Bedingt durch eine große Überdeckung mit HIL-Test, wird der PIL-Test in der Praxis oft nicht durchgeführt. Es findet oft ein Übergang direkt vom SIL-Test zum HIL-Test statt.

2.3.1.4. Hardware In The Loop

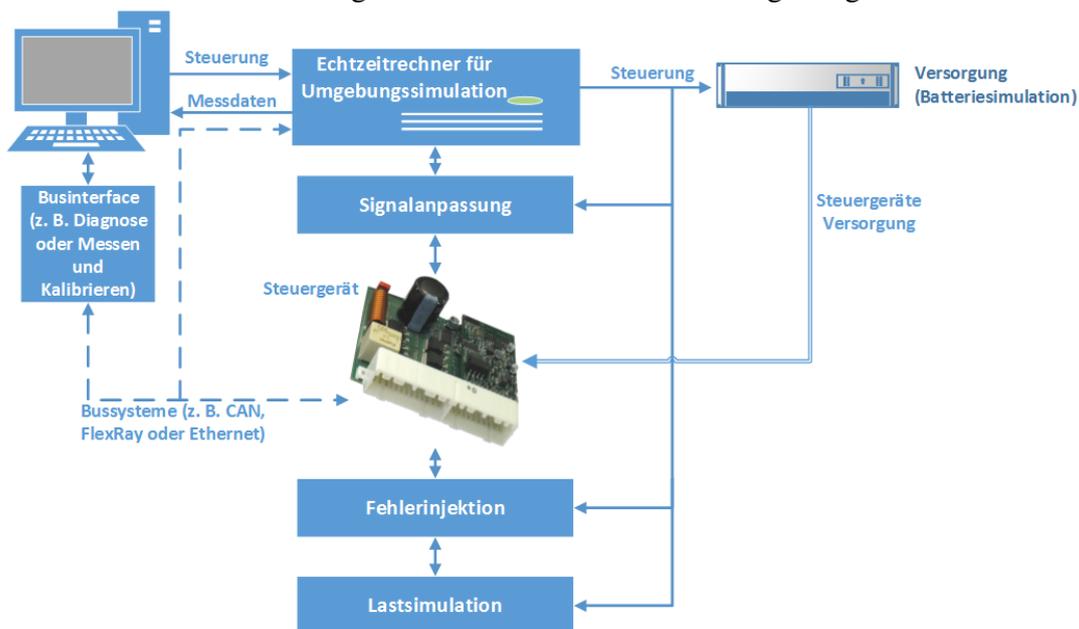
Der Hardware In The Loop-Test (HIL-Test) wird nach dem PIL- bzw. SIL-Test durchgeführt. Dabei wird die Software des eingebetteten Systems auf dem realen Zielsystem ausgeführt. Die Kontaktierung erfolgt dabei über die vorgesehenen Anschlüsse des realen Systems. Nur für spezielle Tests werden extra angepasste Versionen der Ziel-Hardware mit zusätzlichen Anschlüssen verwendet. Dies ist zum Beispiel beim Ersatz von realen Sensoren durch Simulationen der Fall. Die Ziel-Hardware wird mit einem Simulationsrechner verbunden, auf welchem das Umgebungsmodell nun zyklisch – mit typischen Wiederholungszeiten von 1 ms – ausgeführt wird.

Der HIL-Test erlaubt die Prüfung der Hardware und Software unter Echtzeitbedingungen. Es sind damit nun Aussagen über das Laufzeitverhalten möglich. Weiterhin ist der Test hardwarenaher Softwareteile und Regelungen realisierbar. Die Tests sind als Blackbox-Tests angelegt, da eine direkte Beobachtung des Programmablaufes nicht ohne weiteres möglich ist. Mittels der heutigen Debug-Schnittstelle der Mikrocontroller bzw. über das Protokoll XCP ist ein Zugriff in die Software während der Ausführung möglich. Damit sind auch Greybox- und Whitebox-Tests umsetzbar.

Mit dem HIL-Test ist der Test der Integration der Software mit der realen Hardware möglich. Es ist damit die Realisierung von Modultests über Integrations- und Systemtests bis zu Abnahmetests möglich.

Die Abbildung 2.18 zeigt ein typisches HIL-Testsystem für den Test eines eingebetteten Systems. HIL-Testsysteme werden unter anderem von der dSPACE GmbH [39], der MicroNova AG [40] und der iSyst Intelligente Systeme GmbH [41] angeboten. Diese Systeme werden oft auch als Komponenten-HIL bezeichnet.

Abbildung 2.18.: Aufbau der HIL-Testumgebung



Die Ablaufsteuerung der HIL-Tests erfolgt durch den angeschlossenen Steuer-PC, welcher Windows-basiert ist. Dieser ermöglicht ebenfalls das Aufzeichnen des Bustransfers (z. B. CAN, FlexRay oder K-Line). Der PC ist mit einem Echtzeitsystem (z. B. dSPACE [42], National Instruments [43] oder MathWorks [44]) verbunden. Die Echtzeit-Hardware führt ein Umgebungsmodell aus, welches beispielsweise mit Matlab/Simulink oder Labview erstellt wird. Dieses Modell simuliert die Einsatzumgebung des Steuergerätes. Es werden, je nach Anforderung, Bussysteme, Sensoren und / oder Aktoren simuliert. Es ist auch möglich, dass Teile der Umgebung (z. B. Sensoren oder Aktoren) als reale Kompo-

nenen vorhanden sind und als Lastbox verbaut werden.

Zur Anpassung der Signalpegel und der nötigen Signalleistung werden die Signale der Echtzeit-Hardware über Signalkonditionierungskomponenten geleitet. Dies dient auch dem Schutz der Echtzeit-Hardware gegenüber fehlerhaften Signalen aus dem zu testenden Steuergerät.

Aus der vorangegangenen Beschreibung ergeben sich die folgenden Problemstellungen. Die für die Umgebungssimulation eingesetzten Echtzeitsysteme stellen mit ihren technischen Möglichkeiten einen Flaschenhals dar. Vor allem beim Messen und Bewerten schneller Signalverläufe sind die Zykluszeiten der Modellberechnung von meist 1 ms ein Problem.

Zur Bewertung der Genauigkeit schneller Regler mit einigen 100 Hz Grenzfrequenz ist eine Abtastrate von 1000 Hz zu gering um das Abtasttheorem nach Nyquist-Shannon einzuhalten. Hierbei ist zu beachten, dass die doppelte Grenzfrequenz für die Abtastfrequenz in Bereich des Tests zu gering ist. Für eine aussagekräftige Bewertung ist der empfohlene Wert von 10x der Grenzfrequenz für die Abtastfrequenz anzunehmen. Selbst mit der Verwendung von DMA-Transfers und Abtastintervallen von 100 μ s sind schnelle Signalverläufe wie zum Beispiel PWM-Ansteuerungen mit 50 kHz und mehr nicht mehr sinnvoll zu messen und zu bewerten. Es besteht die Möglichkeit der Verwendung von FPGA-Karten (z. B. dSPACE [45] und National Instruments [46]) um zeitkritische Messungen bzw. Simulationsteile auszuführen. Diese Karten verteuern die Testsysteme deutlich und sind bisher auch nur vereinzelt im Einsatz.

Auf der anderen Seite stellt der PC für die Testausführung ein Problem dar. Die im Einsatz befindlichen Windows-PCs sind mit Scheduling-Zeiten von mehreren 10 ms nicht in der Lage die notwendigen zeitlichen Anforderungen zu erfüllen. Durch unterschiedliche Auslastung des Systems kann es zum einem Jitter in der zeitlichen Ausführung der Test kommen. Dies beeinträchtigt die Reproduzierbarkeit der Tests [47].

2.3.2. Testautomatisierung im Software-Test

2.3.2.1. Übersicht über die Testautomatisierung im Software-Test

Zur Testautomatisierung, und dort vor allem im Bereich der HIL-Tests, gibt es viele verschiedene kommerzielle wie auch firmeninterne Lösungen. Als Beispiel für kommerzielle Tools sind hier zu nennen: EXAM der Firma MicroNova [40], AutomationDesk der Firma dSPACE [48], ECU-TEST der Firma TraceTronic [49] und iTestStudio der Firma iSyst [50]. Als firmeninterne Lösungen sind hier beispielhaft TA2 und TA3 der Continental AG (vormals Siemens VDO) zu nennen. Die Lösungen werden bei verschiedenen Firmen eingesetzt. Die Tools werden häufig um eigene Komponenten erweitert, da nicht alle Anforderungen durch die Produkte abgedeckt werden.

Die Implementierung der Tests erfolgt durch Testingenieure. Es gibt auch die Möglichkeit, Testfälle in UML (Sequenzdiagramms in EXAM), als Flussdiagramm (Art eines Flussdiagrammes in AutomationDesk) oder als eine grafische Ablaufbeschreibung

(grafische Darstellung der Testschritte bei ECU-TEST) zu realisieren. Die grafischen Testabläufe werden für die Ausführung in Python-Code übersetzt (z. B.: EXAM und AutomationDesk) und abgearbeitet. Eine automatisierte Erstellung der Tests aus der Testspezifikation erfolgt bisher nicht. Die Verknüpfung der Anforderungen – welche in Anforderungsmanagement-Tools verwaltet werden – mit den eigentlichen Testfällen erfolgt manuell (z. B.: mittels Excel-Listen). Es gibt zwar Ansätze einer automatisierten Verknüpfung der Testfälle und auch der Ergebnisse mit den Anforderungsmanagement-Tools zu realisieren. Diese Anbindungen sind sehr spezifisch auf ein Projekt oder eine Firma ausgelegt.

Weiterhin bieten die meisten Testautomatisierungs-Tools keine oder nur eine rudimentäre Anbindung an Versionsmanagement-Tools zur Versionierung der Tests. Bei datenbankbasierten Tools wie EXAM ist eine Versionierung nur sehr schwer möglich. Die Schnittstellen zu den Mess-, Kalibrier- und Diagnosetools unterscheiden sich auch von Testautomatisierungs-Tool zu Testautomatisierungs-Tool. Bei der Umsetzung der Tests ist man so an ein spezielles Testautomatisierungs-Tool gebunden. Es gibt Bestrebung die Schnittstelle zwischen HIL-System und Software-System zu vereinheitlichen (siehe dazu Kapitel 3.2).

2.3.2.2. Zielsetzung der Testautomatisierung

Mit der Automatisierung der Software-Tests werden verschiedene Ziele verfolgt. Das Hauptziel ist dabei die Beschleunigung der Testausführung. Vor allem für die wiederholte Absicherung des Systems während der Entwicklungszyklen, was auch als Regressionstest bezeichnet wird, ist eine automatisierte Ausführung der Tests unabdingbar. Des Weiteren ist die Absicherung von mehreren tausend Anforderungen über mehrere Releases einer Software ohne ein hohes Maß an Automatisierung nicht möglich. Außerdem ermöglicht ein automatisierter Testablauf eine erhöhte Reproduzierbarkeit der Testergebnisse. Die Fehleranfälligkeit gegenüber der manuellen Ausführung der Testfälle ist geringer. Mit steigender Anzahl der Wiederholungen der Tests kann auch eine Einsparung beim Testaufwand erzielt werden. Die initiale Erstellung der Tests beansprucht mehr Zeit, als eine einmalige, manuelle Ausführung von Tests. Bei mehrfacher Ausführung der gleichen Tests, ist eine Automatisierung hingegen lohnenswert. Dies zeigt gleichzeitig auch die Grenzen des automatisierten Tests auf. Für einmalig auszuführende Tests ist eine Automatisierung meist nicht sinnvoll. Ebenso ist die Testautomatisierung auch nur dort sinnvoll einsetzbar, wo das Testsystem alle nötigen Testschritte automatisiert ausführen kann. Damit ergeben sich weitreichende Anforderungen an das Testsystem. Es müssen alle Vorgaben und Rückmessungen des zu testenden Systems automatisiert zugänglich sein.

Zusammenfassend ist zu sagen, dass der automatisierte Test eine höhere Testabdeckung und eine schnellere Testausführung gegenüber der manuellen Ausführungen bietet. Der für die Automatisierung notwendige höhere Initialaufwand sowie die erweiterten Anforderungen an das Testsystem sind nur sinnvoll, wenn die Tests tatsächlich mehrfach ausgeführt werden sollen.

2.3.2.3. Anforderungen an die Testautomatisierung

Aus den in Abschnitt 2.3.2.1 und 2.3.2.2 erläuterten Gegebenheiten für den automatisierten Test ergeben sich eine Reihe von Anforderungen an ein Testautomatisierungssystem:

1. Die Wiederholbarkeit der Testfälle. Um ein Vertrauen in die Testautomatisierung und die Testergebnisse zu schaffen, ist es notwendig, dass die Tests reproduzierbar gleiche Ergebnisse liefern. Um dies zu ermöglichen, muss die Testautomatisierung einerseits eine reproduzierbare Ausführungszeit der Testfälle sicherstellen. Dies ist notwendig, um vom Ausführungstiming abhängige unterschiedliche Testergebnisse zu vermeiden. Andererseits ist das Timing auch von der Implementierung der Testfälle abhängig und erfordert für die Testbeschreibung sinnvolle Beschreibungsmöglichkeiten. Diese müssen dem Testentwickler eine übersichtliche Darstellung der Testabläufe bereitstellen und gleichzeitig die notwendigen Freiheiten für eine flexible Testimplementierung bieten.
2. Die Austauschbarkeit der Testfälle in ausführbarer Form, wie dies auch schon in Abschnitt 2.2.7 erläutert wurde. Zur Umsetzung dieser Anforderung ist eine Abstraktion der Testfälle vom eigentlichen Testsystem für die Testausführung zu realisieren. Diese Abstraktion darf nicht die Wiederholbarkeit der Testfälle beeinträchtigen. Es ist daher notwendig eine Testbeschreibung mit einer definierten Schnittstelle zur Testausführung zu realisieren. Weiterhin sollte auch das zeitliche Verhalten Teil der Schnittstellenbeschreibung sein.
3. Im Automotive-Bereich ist ein weites Spektrum an zeitlichen Anforderungen gegeben. Die Zeitabläufe, welche im Test zu prüfen sind, beginnen bei ca. 100 μ s, wie sie bei der Regelung von Elektroantrieben auftreten, gehen über Tests im 1 ms Bereich, wie sie bei Bussignalen auf CAN und FlexRay auftreten, bis hin zu Tests aus Fahrersicht mit einer Zeitauflösung von mehr als 200 ms. Zur Umsetzung dieser zeitlichen Anforderungen ist eine definierte Verbindung zwischen Testsystem und Testautomatisierungssystem notwendig. Die Untergrenze der erreichbaren Zeiten werden dabei durch die Erfassungsmöglichkeiten des Testsystems und die Ablaufgeschwindigkeit der Testautomatisierung bestimmt.
4. Ein möglichst geringer Implementierungs- und Wartungsaufwand für die Tests. Nur mit einer effektiven Testimplementierung sind einerseits die engen Zeitvorgaben während der Entwicklung einzuhalten und andererseits eine Einsparung bei den Testaufwänden zu erzielen. Es ist daher notwendig dem Testentwickler eine möglichst effektive und auf die Umsetzung von Tests ausgelegte Testbeschreibung in der Testautomatisierung zur Verfügung zu stellen.
5. Die Robustheit der Testausführung. Um eine möglichst hohe Effektivität des automatisierten Tests zu erreichen, muss die Testausführung rund um die Uhr, ohne eine Aufsicht, lauffähig sein. Im Zusammenspiel mit dem in Entwicklung befindlichen Testobjekt kann es während der Testausführung auch zu Fehlern kommen. Das Testautomatisierungssystem muss dabei sicherstellen, dass nur der jeweils be-

troffene Testfall abbricht. Ein Fehler während der Ausführung eines Testfalls darf nicht zum Abbruch aller weiteren Tests führen. Weiterhin muss sichergestellt werden, dass für jeden Testfall ein definierter Ausgangszustand hergestellt wird. Nur unter diesen Voraussetzungen ist ein automatisierter Test mit voller Effektivität einsetzbar.

6. Eine übersichtliche und flexible Testdokumentation. Um die Tests und Testergebnisse nachvollziehbar darzustellen, ist eine übersichtliche Struktur der Testergebnisse zu realisieren. Es muss die Darstellung der Testergebnisse an die jeweiligen Bedürfnisse des Testprojektes angepasst werden können. Außerdem muss die Testdokumentation eine schnelle Übersicht über die Testergebnisse ermöglichen und Metriken für den Vergleich verschiedener Testdurchläufe zur Verfügung stellen. Weiterhin ist es notwendig, dass die Ergebnisse nicht nur in menschenlesbarer Form zur Verfügung stehen. Für den automatisierten Abgleich der Testergebnisse mit dem Anforderungsmanagement-Tools ist eine einfache maschinelle Verarbeitung notwendig.

2.4. Fertigungsendtest

Der Fertigungsendtest – auch Bandendetest genannt – wird am Ende der Produktion durchgeführt. Er dient der Absicherung der Funktion jedes produzierten eingebetteten Systems. Meist werden während dieses Tests auch Kalibrierungen – z. B. für Strommessungen – durchgeführt. Im Gegensatz zu den entwicklungsbegleitenden Tests wird nicht die gesamte Funktionalität des eingebetteten Systems bzw. der Software getestet.

Ziel ist eine möglichst hohe Abdeckung bei der Funktionalität des eingebetteten Systems. Dies wird durch den Einsatz von Funktionstests mit spezieller Testsoftware realisiert. Diese Software steuert die einzelnen Funktionen des eingebetteten Systems an und die Wirkung wird über Messsysteme aufgenommen und bewertet. Die Serien-Software wird erst am Ende des Fertigungsendtest in das eingebettete System programmiert.

Die Testdurchführung wird durch die Taktzeit der Fertigung zeitlich beschränkt. Es stehen im Automotive-Bereich oft nur 90 s für den Test zur Verfügung.

Die Erstellung der Tests erfolgt unabhängig von den entwicklungsbegleitenden Tests. Dies führt oft dazu, dass Erfahrungen und möglich Fehlerquellen, welche während der Entwicklung aufgefallen sind, nicht im Fertigungsendtest berücksichtigt werden. Des Weiteren werden die Tests manuell geplant und erstellt. Eine quantitative Aussage über die erreichte Testabdeckung ist oft nicht möglich.

Zu Verbesserung dieser Tests wird eine weitgehende Automatisierung der Testerstellung angestrebt. Dies würde auch zu einer deutlichen Reduzierung des Entwicklungsaufwandes für diese Tests beitragen [51].

3. Stand der Technik

In diesem Kapitel soll eine Übersicht über den Stand der Technik im Bezug auf die Inhalte dieser Arbeit gegeben werden. Es werden dabei die verwendeten Testbeschreibungsmethoden sowie ihre Vor- und Nachteile erläutert. Im Anschluss werden verschiedene Testautomatisierungssysteme mit ihren Eigenschaften beschrieben. Weiterhin werden aktuelle Ansätze für die Testdokumentation sowie die Einbindung der Testautomatisierung in den Entwicklungsprozess dargestellt.

3.1. Testbeschreibungsmethoden

In den folgenden Unterabschnitten werden Methoden der Testbeschreibung getrennt nach programmatischer und grafischer Darstellung erläutert. Es werden dabei die Haupteigenschaften sowie die Vor- und Nachteile des jeweiligen Ansatzes dargestellt. Hierbei sind die Vor- und Nachteile hauptsächlich auf den Automotive-Bereich bezogen, da in diesem Bereich die meisten Erfahrungen zur Verfügung standen. Die Betrachtung ist aber nicht ausschließlich auf diesen Bereich beschränkt.

3.1.1. Programmatischer Ansatz

In diesem Abschnitt werden verschiedene Testbeschreibungsmethoden vorgestellt, welche eine textuelle Beschreibung einsetzen.

3.1.1.1. Python

Python [52] ist eine interpretierende höhere Programmiersprache, deren Entwicklung durch die Python Software Foundation (PSF) gefördert wird. Sie ist eine dynamische Sprache (Skriptsprache). Diese Sprache verwendet dynamische Typisierung und unterstützt verschiedene Programmierparadigmen, wie zum Beispiel objektorientierte und funktionale Programmierung.

Da die Skripte direkt ausführbar sind, ist eine schnelle und einfache Entwicklung, wie die direkte Funktionsprüfung von Testfällen, möglich. Python bietet ein breites Spektrum von Standardbibliotheken und flexibel einsetzbaren Datentypen – wie zum Beispiel Mengen und assoziative Arrays. Die Umsetzung von Tests wird ebenfalls durch eine mächtige Stringverarbeitung unterstützt. Es ist keine direkt für den Test entwickelte Sprache. Dies

hat zur Folge, dass Funktionen zur Bewertung von Werten und Signalen sowie für die Ausgabe in den Test-Report geschaffen werden müssen. Dies kann eine erhöhte Fehleranfälligkeit nach sich ziehen.

Im Gebiet der Testautomatisierung im Automotiv Bereich ist Python als ein Quasi-Standard anzusehen, da es bei den verbreiteten Testsystemen der Firma dSPACE, sowie bei Testautomatisierungs-Tools – wie TA2, TA3, AutomationDesk, EXAM und iTestStudio – Verwendung findet.

Vorteile für den Test sind:

- Direkte Ausführung und Erprobung der Testfälle bei der Entwicklung, ohne dass eine Übersetzung notwendig ist
- Ausdrucksstarke Sprachkonstrukte
- Gute Lesbarkeit der Sprache
- Unterstützung verschiedene Programmierparadigmen, wie zum Beispiel objektorientierte und funktionale Programmierung
- Unterstützung verschiedener Betriebssystem (z. B. Windows und Linux)
- Große Standardbibliothek
- Einfache Erweiterbarkeit durch eigene Bibliotheken und einfache Anbindung vorhandener Programme
- Breiter Einsatz im Gebiet der Testautomatisierung im Automotive-Bereich

Nachteile für den Test sind:

- Dynamische Typisierung kann zu schwer wartbarem Code führen
- Strukturierung des Programmes über Einrückungen ist Fehleranfällig
- Keine direkt für den Test entwickelte Sprache
- Durch Mischung verschiedener Programmierparadigmen entstehen schnell nicht wartbare Programme

3.1.1.2. C#

C# [53] ist eine compilierte höhere Programmiersprache, welche von Microsoft entwickelt wurde und als ISO-Standard unter der Bezeichnung ISO/IEC 23270 [53] standardisiert. Es gibt neben dem Standard proprietäre Erweiterungen durch Microsoft. Die Sprache verwendet statische Typisierung und unterstützt neben der objektorientierten auch strukturierte, imperative und deklarative Programmierung. Der Programm-Code wird durch den Compiler in einen Zwischencode umgewandelt und in einer Laufzeitumgebung (Common Language Infrastructure (CLI)) ausgeführt. C# ist keine direkt für den Test entwickelte Sprache. Dies hat zur Folge, dass Funktionen zur Bewertung von Werten und Signalen sowie für die Ausgabe in den Test-Report geschaffen werden müssen. Dies

kann eine erhöhte Fehleranfälligkeit nach sich ziehen.

Die unter Windows verfügbare .Net-Laufzeitumgebung für C# und die einfache Programmierung – auch von Oberflächen – haben zu einem Einsatz für firmeninterne Lösungen zur Testautomatisierung geführt.

Vorteile für den Test sind:

- Ausdrucksstarke Sprachkonstrukte
- Gute Lesbarkeit der Sprache
- Unterstützung verschiedene Programmierparadigmen
- Unterstützung verschiedener Betriebssystem (z. B. Windows und Linux)
- Einfache Erstellung von Benutzeroberflächen
- Vorhandensein der Laufzeitumgebung unter Windows

Nachteile für den Test sind:

- Programme müssen vor der Ausführung kompiliert werden
- Durch proprietäre Erweiterungen ist die Ausführbarkeit der Programme auf verschiedenen Plattformen (z. B. Linux und MacOS) nicht sichergestellt
- Keine direkt für den Test entwickelte Sprache
- Durch Mischung verschiedener Programmierparadigmen entstehen schnell nicht wartbare Programme

3.1.1.3. Open Test Exchange (OTX – ISO 13209)

OTX [54] ist im Gegensatz zu den vorher genannten Sprachen speziell für den Test im Automotive-Bereich entwickelt wurden. Sie wird unter der Bezeichnung ISO 13209 [54] als internationaler Standard gepflegt und steht aktuell (Stand September 2013) in Version 1.0.0 zur Verfügung. Es ist dabei nicht festgelegt, ob die Sprache kompiliert oder interpretiert wird. Es ist eine entsprechende Laufzeitumgebung notwendig. Die Sprache ist für die prozedurale Programmierung mit statischer, starker Typisierung entworfen.

Die Sprache ist für den Testaustausch und die Ausführung von Tests der Fahrzeugdiagnose im Automotive-Bereich vorgesehen. Sie basiert auf XML und bietet neben der textuellen Darstellung auch eine grafische Bearbeitungsmöglichkeit.

Vorteile für den Test sind:

- Sprache speziell für Testaustausch und Testausführung
- Textuelle und grafische Bearbeitung der Testfälle
- Unterstützung verschiedener Betriebssystem (z. B. Windows und Linux)

Nachteile für den Test sind:

- Einschränkung auf Testfälle für die Fahrzeugdiagnose
- Standard nicht frei verfügbar
- Standard noch nicht weit verbreitet und wenige Erfahrungswerte über praktischen Einsatz

3.1.1.4. Testing and Test Control Notation Version (TTCN-3)

TTCN-3 [55] ist ebenfalls eine speziell für den Test entwickelte Programmiersprache und hat ihre Ursprünge in der Telekommunikationsbranche. Sie wird als offener Standard von der European Telecommunication Standards Institute (ETSI) gepflegt und es werden regelmäßig neue Versionen veröffentlicht. Aktuell ist Version 4.5.1 (Stand September 2013) des Sprachkerns (ES 201 873-1 [56]). Die Sprache selbst kann mit der Version 1 (TTCN-1) und Version 2 (TTCN-2) auf eine lange Geschichte zurück blicken, welche mit der Standardisierung von TTCN als ISO TC 97/SC 16 im Jahr 1983 begann. Es ist dabei nicht festgelegt, ob die Sprache kompiliert oder interpretiert wird. Es ist eine entsprechende Laufzeitumgebung notwendig, welche ebenfalls Teil des Standards ist (ES 201 873-4 [57], ES 201 873-5 [58] und ES 201 873-6 [59]). Die Sprache ist für die prozedurale Programmierung mit statischer, starker Typisierung entworfen.

Für die Sprache stehen, durch ihre lange Anwendung im Bereich des automatisierten Tests, viele praktische Erfahrungen zur Verfügung [60]. Neben der textuellen Darstellung ist mit dem Graphical Format (GFT) auch eine standardisierte grafische Bearbeitungsmöglichkeit für Testfälle gegeben. Diese Möglichkeit ist parallel zur textuellen Programmierung einsetzbar.

Vorteile für den Test sind:

- Ausdrucksstarke Sprachkonstrukte
- Sprache speziell für Testausführung
- Spezielle Sprachkonstrukte – wie zum Beispiel einen Datentyp für Testbewertungen (verdict) und Strukturen für Testfälle (testcase) – für den Test
- Unterstützung verschiedener Betriebssystem (z. B. Windows und Linux)
- Offener Standard
- Langjährige Erfahrungen
- Standardisierte Laufzeitumgebung und Schnittstellen
- Grafische und textuelle Darstellung der Testfälle

Nachteile für den Test sind:

- Geringe Erfahrungen im Automotive-Bereich
- Verfügbare Tools sind auf Telekommunikation ausgerichtet

3.1.1.5. Extensible Markup Language (XML)

XML [61] ist selbst keine Programmiersprache. Es ist eine Auszeichnungssprache für die hierarchische und strukturierte Ablage von Daten in Textdateien. Die XML-Spezifikation wird durch das World Wide Web Consortium (W3C) gepflegt und legt eine Metasprache für die Definition anwendungsspezifischer Sprachen fest. Es existieren eine große Anzahl von Dateiformaten, welche auf XML aufsetzen.

Für die Anwendung im Bereich des Tests sei hier OTX (siehe Abschnitt 3.1.1.3) genannt. Weiterhin setzen einige firmeninterne Tools, wie zum Beispiel TA2, XML für die Testbeschreibung ein.

Vorteile für den Test sind:

- Weitreichende Definitionsmöglichkeiten der Struktur
- Unterstützung verschiedener Betriebssystem (z. B. Windows und Linux)
- Offener Standard

Nachteile für den Test sind:

- Keine Programmiersprache, sondern eine Metasprache zur Sprachdefinition
- Kein einheitlicher Standard für Tests

3.1.1.6. Weitere programmatische Testbeschreibungsmethoden

Neben den in den vorangegangenen Abschnitten beschriebenen programmatischen Testbeschreibungsmethoden, gibt es noch eine Vielzahl von weiteren Testbeschreibungsmethoden und -standard. Diese sollen im Folgenden der Vollständigkeit halber vorgestellt werden, da diese für den hier beschriebenen automatisierten Software-Test nicht einsetzbar sind:

- IEEE 1445 – Standard for Digital Test Interchange Format (DTIF) [62]
- IEEE 1450 – Standard Test Interface Language (STIL) [63]
- IEEE 1671 – Automatic Test Markup Language (ATML) [64]
- TAP – The Test Anything Protocol [65]
- TET– Test Environment Toolkit [66]
- HIL-API [67]

Der Standard IEEE 1445 bezieht sich auf den Austausch von Testfällen. Er ist für den Austausch von Testfällen zwischen digital automated test program generator (DATPG) und automatic test equipment (ATE) vorgesehen.

Der Standard IEEE 1450 ist ein allgemeines Datenformat für Testfälle für ATE im Bereich der Halbleiterindustrie.

Der Standard IEEE 1671 ist ein XML basiertes Format für den Austausch von Testinformationen für ATE im Produktionstest von Hardware.

TAP ist ein Standard für die Automatisierung von Software-Test und bietet ein sehr einfaches Funktionsinterface für die Testdurchführung und Testbewertung. Der Standard bietet keine Möglichkeit der Gliederung von Testfällen und keinerlei Meta-Informationen (z. B. Anforderungs-ID).

TET stellt ein Werkzeug zur Testausführung und zum Testmanagement dar. Es ist aber nur für den Test von Software auf eingebetteten Systemen mit Betriebssystemen nach dem Standard POSIX (Standard for Embedded Realtime Systems POSIX 1003.13) oder Embedded Linux Systemen ausgelegt.

Standards, welche sich direkt mit der Anwendungsdomäne HIL-Test beschäftigen, existieren nur in Form der HIL-API des ASAM (Association for Standardisation of Automation and Measuring Systems) Vereins. Diese API stellt eine einheitliche Schnittstelle zu dem Testsystem dar. Es werden keine Vorgaben für die Testbeschreibung gemacht. Außerdem wird nur eine Untermenge der Möglichkeiten heutiger HIL-Systeme durch die Schnittstelle abgebildet.

3.1.2. Modellbasierter Ansatz

In diesem Abschnitt werden verschiedene Testbeschreibungsmethoden vorgestellt, welche eine modellbasierte Beschreibung einsetzen.

3.1.2.1. Unified Modeling Language (UML)

UML [68, 69] ist eine graphische Modellierungssprache, welche von der Object Management Group (OMG) sowie der ISO (ISO/IEC 19505) standardisiert ist. Die Sprache dient der Spezifikation, Konstruktion und Dokumentation von Systemen und kommt stark bei der Modellierung von Software-Systemen zum Einsatz. Neben verschiedenen Diagrammartent definiert der UML-Standard auch Begriffe und Bezeichnungen.

Der Standard bietet dabei eine große Anzahl von verschiedenen Diagrammen, von denen im Folgenden einige beispielhaft genannt werden sollen:

- Aktivitätsdiagramm
- Anwendungsfalldiagramm
- Sequenzdiagramm
- Kommunikationsdiagramm
- Klassendiagramm

In der Praxis kommt, je nach Problemstellung, nur ein Teil der möglichen Diagramme zum Einsatz. Beispielhaft für die Anwendung im Test sei hier EXAM (siehe auch Kapitel 3.2.2.2) genannt. Für die Formulierung der Testabläufe werden UML-

Sequenzdiagramme, wie in Abbildung 3.1 dargestellt, verwendet. Diese erlauben es, die zeitliche Abfolge der Testschritte zu definieren.

Weiterhin kommt UML für den modellbasierten Entwurf von Software zum Einsatz. Es wird dabei das gewünschte Verhalten der Software mittels UML-Diagrammen modelliert. Dies kann als Spezifikation für die Softwareentwicklung dienen. Es kann gegebenenfalls auch direkt Code aus dem Modell generiert werden. Die Generierung von Testfällen aus solchen Entwurfsmodellen kommt ebenfalls zum Einsatz [70]. Für den Test ist die Güte des Modelles sehr wichtig. Weiterhin darf der Test nicht das selbe Modell, wie die Entwicklung verwenden, da sonst Fehler im Modell unentdeckt bleiben. Weiterhin stellt die Wahl der sinnvollen Testfälle bei der automatischen Generierung eine große Herausforderung dar, da sehr schnell viele Testfälle generiert werden können. Dies kann zu einer sehr hohen Ausführungszeit führen [70].

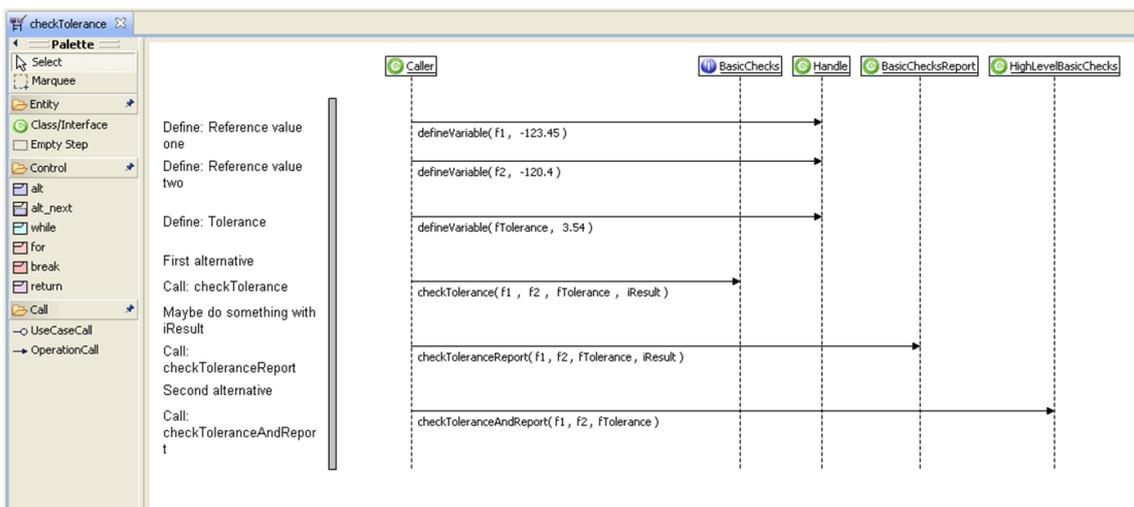
Vorteile für den Test sind:

- Sehr große Anzahl von Ausdrucksmitteln für die Modellierung
- Breiter Einsatz in der Entwicklung, vor allem in der Softwareentwicklung
- Automatische Generierung von Code bzw. Tests
- Unabhängig vom Betriebssystemen einsetzbar

Nachteile für den Test sind:

- Großer Einarbeitungsaufwand durch die hohen Freiheitsgrade
- Modelle können unübersichtlich werden
- Hoher Pflegeaufwand für Modelle (z. B. ein Modell für die Entwicklung und ein Modell für den Test)
- Speicherung der Modelle in proprietären Formaten erschwert Austausch

Abbildung 3.1.: Beispiel Sequenzdiagrammen von EXAM



3.1.2.2. Matlab/Simulink

Simulink [71] ist eine Software der Fa. The MathWorks und dient der Systemsimulation. Es ist ein Zusatz zur Mathematiksoftware MATLAB [72]. Die Modellierung von technischen, physikalischen oder finanzmathematischen Problemen erfolgt in einer hierarchischen Struktur mittels grafischer Blöcke, wie dies in Abbildung 3.2 beispielhaft dargestellt ist. Die Darstellung der Modellierung folgt dabei keinem allgemeinen Standard, wie zum Beispiel UML. Es ist eine proprietäre Darstellungsform.

Simulink bietet eine große Anzahl verschiedener Blöcke. Je nach Problemstellung können auch zusätzliche Blockset, welche in Simulink Funktionsbibliotheken darstellen, von The Mathworks oder anderen Anbietern bezogen werden. Es besteht auch die Möglichkeit selbst Blöcke, zum Beispiel über die Einbindung von Matlab- oder C-Code, zu erstellen. Es ist dabei möglich, aus den Modellen Code für verschiedene Plattformen, wie Mikrocontroller oder FPGAs, zu generieren, wobei dies nicht gleichzeitig möglich ist. Auf dieser Basis wird Simulink für die Modellierung bzw. Programmierung von Software-Komponenten, wie zum Beispiel Reglern, eingesetzt. Im Automotive-Bereich wird Simulink für die Erstellung von Umgebungsmodellen für den Test, wie sie beispielsweise bei HIL-Tests zum Einsatz kommen, verwendet. Eine direkte Generierung von Testfällen erfolgt dabei nicht.

Weiterhin besteht über die Erstellung von m-Skripten in Form von m-Dateien die Möglichkeit zur Programmierung von Abläufen in MATLAB. Diese m-Dateien lassen sich in der MATLAB-Konsole ausführen oder in Simulink-Modelle integrieren.

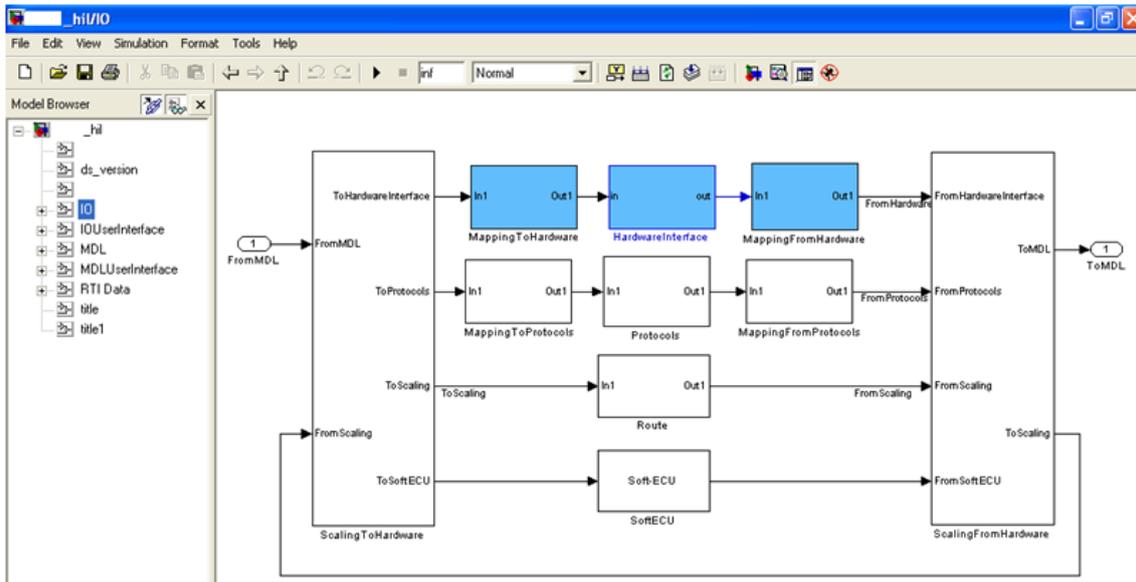
Vorteile für den Test sind:

- Sehr große Anzahl von Blocksets für verschiedene Problemstellungen für die Modellierung
- Weite Verbreitung in der Entwicklung, vor allem in der Softwareentwicklung
- Automatische Generierung von Code
- Einsatz unter Windows und Linux möglich

Nachteile für den Test sind:

- Großer Einarbeitungsaufwand durch die hohen Freiheitsgrade
- Modelle können unübersichtlich werden
- Hoher Pflegeaufwand für Modelle (z. B. ein Modell für die Entwicklung und ein Modell für den Test)
- Hoher Portierungsaufwand durch den Einsatz unterschiedlicher Blockset für unterschiedliche Testsysteme

Abbildung 3.2.: Beispiel Simulink Modell



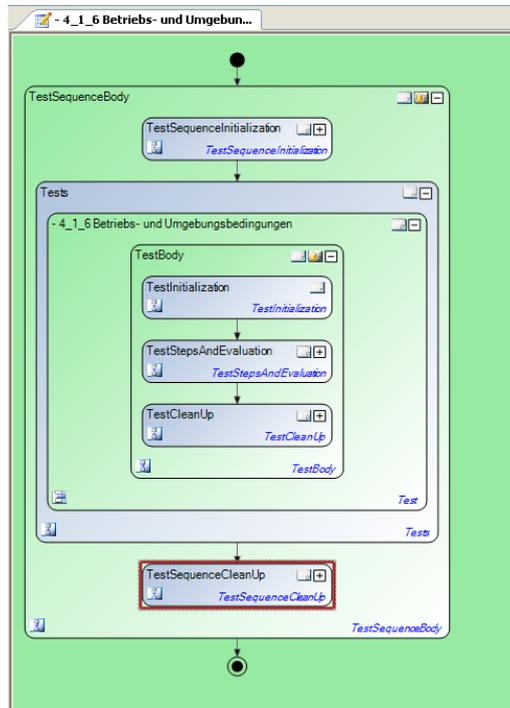
3.1.2.3. Weitere modellbasierte Testbeschreibungsmethoden

Neben den in den vorangegangenen Abschnitten genannten modellbasierten Testbeschreibungsmethoden gibt es noch Weitere, welche in diesem Abschnitt erläutert werden. Dabei liegt der Fokus auf im Automotive-Bereich eingesetzten Testbeschreibungsmethoden.

Als erstes ist hier Labview (Laboratory Virtual Instrumentation Engineering Workbench) [73] der Firma National Instruments zu nennen. Dieses grafische Programmiersystem erlaubt die signalfluss-basierte Modellierung eines Systems. Es kommt dabei vor allem in der Mess-, Regel- und Automatisierungstechnik zum Einsatz. Im Automotive-Bereich wird es vor allem für Messsysteme und für Umgebungsmodelle der Testsysteme eingesetzt. Bedingt durch die signal-basierte Darstellung, können die Modelle schnell an Übersichtlichkeit verlieren.

Als zweites ist hier AutomationDesk der Firma dSPACE zu nennen, welches nochmals genauer in Kapitel 3.2.2.1 erläutert wird. In diesem Tool werden die Testabläufe grafisch dargestellt, wie dies in Abbildung 3.3 zu sehen ist. Diese Darstellung ist an Ablaufdiagramme angelehnt. Sie folgt dabei keinen der Standards, wie zum Beispiel DIN 66001 oder DIN 66261. Auf weitere, meist firmenspezifische Lösungen soll hier aus Platzgründen nicht weiter eingegangen werden.

Abbildung 3.3.: Beispiel AutomationDesk Testablauf



3.1.3. Grenzen und Probleme verfügbarer Testbeschreibungsmethoden

Wie in den vorangegangenen Abschnitten beschrieben wurde, existieren verschiedene programmatische und modellbasierte Testbeschreibungsmethoden. Sie sind jeweils für unterschiedliche Teststufen und Testarten geeignet. Keine der Testbeschreibungsmethoden deckt alle Anforderungen vom Modultest bis zum Fertigungsendtest ab und verbindet dabei auch den modellbasierten und programmatischen Ansatz vollständig.

Vor allem die modellbasierten Ansätze für die Testbeschreibung sind nur bedingt für Modultests bzw. Signaltests auf hardwarenaher Ebene geeignet. Der Aufwand für die Modellierung steht hier in keinem Verhältnis zur Zeit der Testdurchführung. Weiterhin können komplexere Testabläufe mit verschachtelten Schleifen unübersichtlich werden.

Bei den programmatischen Testbeschreibungsmethoden stellt die Portierbarkeit der Testfälle zwischen verschiedenen Testsystemen eine große Schwierigkeit dar. Auch ist der Einsatz nicht in allen Teststufen gleichermaßen möglich bzw. sinnvoll. Vor allem der Test von komplexen Systemverhalten führt schnell zu einem hohen Programmier- und Pflegeaufwand für die Testfälle.

3.2. Testautomatisierungssysteme

In diesem Abschnitt wird der aktuelle Stand der Technik im Bereich der Testautomatisierungssysteme, mit dem Fokus auf Systeme im Automotive-Bereich, erläutert. Dabei werden, neben den allgemeinen Standards, auch verschiedene verbreitete Werkzeuge dargestellt und ihre Vor- und Nachteile kompakt erläutert.

3.2.1. Standards für Testausführungssysteme

Im Bereich der Testausführungssysteme gibt es eine große Überschneidung mit den Standards für die Testbeschreibung. Viele der in Abschnitt 3.1 beschriebenen Standards stellen, neben der Testbeschreibung, auch die Testausführung zur Verfügung. Aus diesem Grund soll hier nur kurz auf wichtige Aspekte der Testausführung im Bezug auf diese Arbeit eingegangen werden. Dafür werden weiterhin nur die wichtigsten Standards betrachtet.

3.2.1.1. Python

Python [52] ist als Programmiersprache kein Standard für die Ausführung von Testfällen im eigentlichen Sinne. Es soll hier betrachtet werden, da es vor allem im Automotive-Bereich, für die Testausführung verbreitet zum Einsatz kommt. Durch die im Abschnitt 3.1.1.1 beschriebenen Eigenschaften lässt sich Python sehr einfach und flexibel für die Ausführung von Tests einsetzen und bietet einige wichtige Vorteile.

Als ersten sei hier die direkte Ausführung der Testskripte zu nennen. Damit ist eine schnelle Entwicklung und Erprobung der Tests möglich. Es treten dabei keine großen Wartezeiten, wie zum Beispiel das Übersetzen des Codes, auf.

Als zweiter Punkt sei die einfache Anbindung an verschiedenen Testsysteme zu nennen. Durch den direkten Zugriff auf verschiedenste Schnittstellen aus Python heraus, ist die schnelle Anbindung verschiedenster Testsysteme möglich.

Es ergeben sich einige Nachteile beim Einsatz von Python für die Testausführung:

1. Der fehlende zeitliche Determinismus bei der Ausführung. Dabei ist nicht nur Python selbst, sondern auch das verwendete Betriebssystem dafür mit verantwortlich. Die zeitlichen Schwankungen bei der Testausführung können in der Praxis zu Problemen bei der Reproduzierbarkeit der Tests führen. Dies birgt vor allem im Bereich des automatisierten Tests große Probleme.
2. Die Portierbarkeit der Tests. Auf der einen Seite kann der Einsatz betriebssystemspezifischer Bibliotheken zu Problemen bei der Portierung der Tests von einem System auf ein anderes führen. Auf der anderen Seite ist die Anbindung von Testsystemen über eigene Schnittstellen problematische. Da Python selbst keine Standards für derartige Schnittstellen besitzt, ist der Anwender selbst für die Spezifikation,

Implementierung und Verwendung von Schnittstellen zu Testsystemen verantwortlich. Die Portierbarkeit hängt dabei von der Güte der jeweiligen Implementierungen ab. Da die Schnittstellen zu Testsystemen keinen Bestandteil der Sprache darstellen, ist ein Austausch von Testfällen zwischen verschiedenen Testsystemen und Testautomatisierungslösungen, auch wenn sie alle Python verwenden, nur schwer möglich.

3.2.1.2. OTX – ISO 13209

OTX [54] standardisiert, neben der Testbeschreibung, auch die Testausführung. Neben den in Abschnitt 3.1.1.3 beschriebenen Eigenschaften, kommen für die Testausführung noch weitere Eigenschaften zum Tragen.

Diese stellen folgende wichtige Vorteile dar:

1. Die standardisierte Schnittstelle zu den Testtools. Es werden dabei die Schnittstellen zu Diagnosetools (DIAGCOM) aus dem Automotive-Bereich sowie für das Benutzerinterface (HMI) betrachtet.
2. Die Standardisierung der Testausführung. Als Teil des Standards ist das Ausführungsverhalten des OTX Systems festgelegt.
3. Die Austauschbarkeit der Testfälle. Die Austauschbarkeit der Testfälle war ein zentrales Ziel bei der Erstellung des OTX Standards und wird schon durch den Namen *Open Test Exchange* zum Ausdruck gebracht.

Neben den genannten Vorteilen zeigt sich ein vorrangiger Nachteil für die Testausführung auf Basis des OTX Standards. Der Standard betrachtet einzig und allein die Beschreibung und Ausführung von Tests der Fahrzeugdiagnose im Automotive-Bereich. Es werden zwar Erweiterungen in andere Bereiche diskutiert [74], aber aktuell sind keine Arbeiten in diese Richtung bekannt.

3.2.1.3. TTCN-3

TTCN-3 [55] standardisiert, neben der Testbeschreibung, ebenfalls die Testausführung. Neben den in Abschnitt 3.1.1.4 beschriebenen Eigenschaften, kommen für die Testausführung noch weitere Eigenschaften zum Tragen. Als ein wichtiger Vorteil sind die standardisierten Schnittstellen TTCN-3 Runtime Interface (TRI) [58] und TTCN-3 Control Interface (TCI) [59] zu nennen. Das TRI bildet dabei die Schnittstelle zur Testplattform und damit zum System auf welchem die Tests ausgeführt werden. Weiterhin bildet es auch die Schnittstelle zum Device Under Test (DUT).

Das TCI bildet die Schnittstelle zum Testmanagement.

Dies beinhaltet die folgenden Punkte:

- Ablaufsteuerung der Testfälle
- Handling mehrerer verbundener Testsysteme
- Codierung und Decodierung DUT spezifischer Daten
- Anbindung des Loggings bzw. des Test-Reports

Als zweites sei die standardisierte Ausführung der Tests zu nennen. Das Ausführungsverhalten eines TTCN-3 Systems ist im Standard beschrieben und damit eindeutig festgelegt. Als drittes sei die Austauschbarkeit der Testfälle zu nennen. Der TTCN-3 Standard stellt die Austauschbarkeit von Testfälle explizit als Ziel dar. Durch die Standardisierung der Schnittstellen sowie des Ausführungsverhaltens ist der Austausch möglich.

Neben den genannten Vorteilen zeigen sich auch Nachteile für die Testausführung auf Basis des TTCN-3 Standards:

1. Der hohe Abstraktionsgrad der Tests. Bedingt durch die Standardisierung der Schnittstellen, sind die Testfälle weit vom Testsystem abstrahiert. Dies kann einerseits das Verständnis zwischen Ursache und Wirkung der Anweisungen eines Testfallen in Verbindung mit dem realen Testsystem erschweren. Andererseits führen die zusätzlichen Softwareschichten für die Abstraktion zu einer höheren Laufzeit.
2. Die fehlende Standardisierung des zeitlichen Verhaltens eines TTCN-3 Testsystems. Die zeitliche Dauer einzelner Kommandos ist nicht spezifiziert. Dies hängt damit stark vom jeweiligen Testsystem ab, was beim Austausch von Testfällen zwischen verschiedenen Testsystemen zu Problemen und Portierungsaufwand führen.

3.2.2. Kommerzielle Produkte

In den folgenden Abschnitten wird ein Überblick, über eine Auswahl kommerzieller Produkte zur Testautomatisierung, gegeben. Die hier vorgestellten Tools haben eine weite Verbreitung im Automotive-Bereich.

3.2.2.1. dSPACE AutomationDesk

Das Testautomatisierungs-Tool AutomationDesk [75] wird durch die Firma dSPACE entwickelt und vertrieben. Es ist für die Automatisierung von HIL-Tests vorgesehen. Dabei liegt der Fokus auf der Unterstützung der von dSPACE angebotenen HIL-Systeme. Es werden keine anderen Testsysteme unterstützt. AutomationDesk wird als Teil des dSPACE Releases installiert. Tests werden als Ablaufgraphen grafisch programmiert, wie dies in Abbildung 3.3 dargestellt ist.

Im Lieferumfang von AutomationDesk sind die Anbindungen an verschiedene Testtools,

wie zum Beispiel DTS6, DTS7, dSPACE ControlDesk und Calibration Tools, enthalten. Die Erweiterung des AutomationDesk kann über Bibliotheken erfolgen. Diese Bibliotheken können aus Blöcken der anderen AutomationDesk-Bibliotheken oder auch aus Python-Klassen bestehen. Die nutzerspezifischen Bibliotheksblöcke lassen sich wie Blöcke aus den AutomationDesk-Bibliotheken verwenden.

Die Strukturierung der Tests erfolgt in Testprojekten und Testfällen. Innerhalb der Testfälle werden die Testabläufe in grafischer Form erstellt. Die verwendeten Variablen werden den einzelnen Blöcken im Testablauf zugeordnet. Dabei ist zu beachten, dass immer die in der Hierarchie der Blöcke nächstgelegene Variable eines Namens verwendet wird. Dies kann bei größeren Testfällen zu unübersichtlichen Strukturen führen. Allgemein ist zu sagen, dass der Datenfluss bei AutomationDesk nur schwer zu verfolgen ist. Eine klare Strukturierung der Testfälle ist daher sehr wichtig.

Für die Testausführung lassen sich einzelne Testfälle oder ein Testprojekt ausführen. Dabei wird jeweils ein Testbericht erzeugt. Dieser kann im HTML- oder PDF-Format generiert werden. Die Testberichte sind nur in geringem Maße anpassbar und enthalten, je nach AutomationDesk-Version, keine (AutomationDesk 2.X) oder nur eine geringe Anzahl von Metriken (ab AutomationDesk 3). Ein Nachtest einzelner Testfälle, mit Übernahme der Ergebnisse in einen gesamten Testbericht, ist nicht möglich. Auch die Ausführung mehrerer Testprojekte ist nur unter Zuhilfenahme eines zusätzlichen Tools namens *Operator* möglich. Der Operator verwendet dabei die Möglichkeit AutomationDesk als COM-Objekt aus anderen Applikation fernzusteuern.

Die Speicherung der Testprojekte und Bibliotheken erfolgt binär. Für die Bearbeitung von Projekten im Team steht eine Anbindung an Versionsmanagement-Werkzeuge zur Verfügung, wobei die Granularität auf Testprojekte und Bibliotheken beschränkt ist. Ein Vergleich von verschiedenen Versionen und das Zusammenführen dieser, ist auf Grund des Binärformates nur schwer möglich.

Vorteile:

- Einfache Einbindung mit dSPACE-System
- Einfache Installation

Nachteile:

- Eingeschränkte Debug-Möglichkeit
- Geringer Funktionsumfang des Test-Reports
- Beschränkte Unterstützung von Testtools (z. B. CANape)
- Hoher Ressourcenbedarf
- Schlechte Anbindung an das Versionsmanagement

3.2.2.2. MicroNova EXAM

Das Testautomatisierungs-Tool EXAM [40] wird durch die Firma MicroNova AG im Auftrag der Volkswagen AG (VW) entwickelt. Es wird ab Version 2.0 als Freeware über die Internetseite www.exam-ta.de verbreitet. EXAM 1.0 wurde nur VW intern verwendet [76]. Am 14.09.2009 wurde EXAM 2.0 offiziell freigegeben. EXAM 2.0 ist für den Einsatz in größeren Projekten und Projektteams vorgesehen. Es besteht je nach Konfiguration aus 3 oder 4 Teilkomponenten.

Bei einer Konfiguration für eine größere Anzahl von Testingenieuren bzw. getrennten Personen für die Testimplementierung und die Testausführung, kommt die Konfiguration mit den folgenden 4 Komponenten zum Einsatz:

- EXAM modeller zur Erstellung der Testabläufe
- EXAM generator zur Erzeugung des Python-Codes für die Testausführung
- EXAM testrunner zur Ausführung der Tests
- EXAM reportmanager zur Bearbeitung und Erzeugung des Test-Reports

Für kleinere Testprojekte werden die Komponenten EXAM modeller und EXAM generator, zur Komponente EXAM modeller_SA (SA = Stand Alone) vereint. In diesem Fall übernimmt der EXAM modeller die Erzeugung der Python-Code. Es werden damit die folgenden 3 Komponenten verwendet:

- EXAM modelle_SA zur Erstellung der Testabläufe und zur Erzeugung des Python-Codes für die Testausführung
- EXAM testrunner zur Ausführung der Tests
- EXAM reportmanager zur Bearbeitung und Erzeugung des Test-Reports

Ab der Version 3 von EXAM wurden die Komponenten EXAM modeller, EXAM testrunner und EXAM reportmanager in eine gemeinsame Oberfläche integriert.

EXAM basiert auf dem Eclipse-Framework und stellt nur die Umgebung für die Erstellung von Tests in Form von UML-Sequence-Diagrammen zur Verfügung. Die Ausführung der Tests und die Erzeugung des Test-Reports sind ebenfalls Bestandteil des Funktionsumfangs. Es stellt in seiner Grundausstattung keinerlei Möglichkeiten zur Anbindung von Testtools oder zur weiteren Verarbeitung von Werten zur Verfügung. Diese Funktionalität wird über die sogenannte Core-Bibliothek realisiert, welche von der VW AG als Open Source über die Webseite <http://www.exam-ta.de/> zur Verfügung gestellt wird. Diese Bibliothek bietet sehr umfangreiche Basisfunktionen beginnend bei der Anbindung von Testtools bis hin zur Anbindung von Klimakammern und Oszilloskopen. Als primäres Einsatzszenario ist EXAM für die Zusammenarbeit größerer Testteams bzw. Testabteilungen vorgesehen. Die gesamte Datenbasis, angefangen bei den Bibliotheken über die Testabläufe bis hin zu den Testergebnissen, wird in 2 Datenbanken verwaltet. Eine Datenbank umfasst die Bibliotheken und Testfälle. Die andere Datenbank umfasst die Testergebnisse. Es sind auch Mechanismen zur Abstimmung von Änderungen zwischen verschiede-

nen Nutzern implementiert. Eine Anbindung an das Requirementsmanagement-Werkzeug Doors ist ebenfalls vorgesehen. Dazu ist eine eigene Komponente mit der Bezeichnung EXAM DOORS SessionManager vorhanden. Eine Anbindung an Versionsmanagement-Werkzeuge ist hingegen nicht möglich.

Vorteile:

- Umfangreiche Core-Bibliothek
- Umfangreiche Möglichkeiten für die Ausgabe von Daten in den Test-Report
- Umfangreiche Möglichkeiten zur Erzeugung von Reports in verschiedenen Formaten

Nachteile:

- Hoher Installationsaufwand
- Overhead bei der Arbeit mit 4 eigenständigen Programmkomponenten
- Umfang der nötigen Infrastruktur

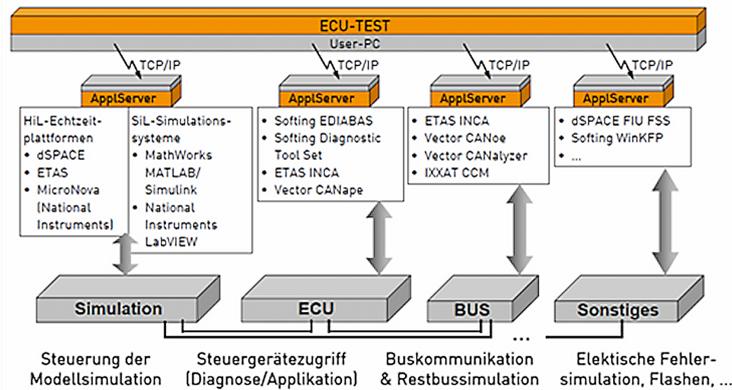
3.2.2.3. TraceTronic ECU-Test

Die Entwicklung und der Vertrieb von ECU-TEST [49] erfolgen durch die TraceTronic GmbH in Dresden. Es handelt sich bei ECU-TEST, im Gegensatz zu EXAM, um ein kommerzielles Produkt. Es sind Lizenzkosten pro Softwarelizenz zu entrichten. TraceTronic bietet neben dem Testautomatisierungs-Tool auch Dienstleistungen für die Anpassung von ECU-TEST sowie Schulungen für ECU-TEST an. Das Testautomatisierungs-Tool ECU-TEST wird bei verschiedenen Automobilherstellern für den HIL-Test von einzelnen Steuergeräten und auch im Bereich von Verbundtests eingesetzt.

Das Testautomatisierungs-Tool besitzt eine modulare Struktur, mit der es möglich ist, Testtools auf verschiedenen Rechnern über das Netzwerk anzubinden. Der mögliche Aufbau ist in Abbildung 3.4 zu sehen [77]. Dies ermöglicht die Verteilung der Testsoftware und damit der Systembelastung auf verschiedene Systeme. Damit besteht die Möglichkeit einer einfachen Skalierung des Testsystems. Es ist aber auch möglich, alle Testtools auf einem Test-PC zu betreiben.

Die Implementierung der Testfälle erfolgt in einer grafischen Ablaufbeschreibung. ECU-TEST unterstützt sowohl den Test einzelner Steuergeräte als auch den Test von Verbundaufbauten mehrerer Steuergeräte. Im Rahmen einer Evaluierung konnte nur der Test eines einzelnen Steuergerätes erprobt werden. Es stand kein Aufbau zum Test mehrerer Steuergeräte zur Verfügung. Mit der grafischen Erstellung von Testfällen zielt ECU-TEST laut TraceTronic auch auf Testentwickler ohne Programmiererfahrung. Eine Anbindung an Versions- und Requirementsmanagement-Werkzeuge scheint nicht direkt durch das Testautomatisierungs-Tool unterstützt zu werden. Es besteht über die Einbindung des SVN-Clients Tortoise eine Anbindung an SVN als Versionsmanagement-Werkzeug. Weiterhin besteht die Möglichkeit in der XML-Datei bzw. dem Eigenschaftsdialog eines

Abbildung 3.4.: Aufbau des ECU-TEST Applikation Servers [77]



Testfalls ein frei belegbares Feld für die Anbindung an das Requirementsmanagement-System zu verwenden.

Vorteile:

- Signalanalyse auf Basis der *real time linear temporal logic*
- Flexible und transparente Verteilung der Testsoftware auf verschiedene Computersysteme
- Große Anzahl unterstützter Testtools
- Kombinierbarkeit der Testkonfiguration und der Testbenchkonfiguration

Nachteile:

- Geringer Funktionsumfang des Test-Reports
- Schlechte Dokumentation
- Unvollständige Unterstützung des Funktionsumfangs des meisten Testtools

3.2.2.4. Weitere kommerzielle Produkte

In diesem Abschnitt werden weitere kommerzielle Produkte kurz erläutert, welche im thematischen Bezug zu dieser Arbeit stehen.

TTworkbench [78] ist ein Tool der Firma Testing Technologies. Das Tool bietet einen sehr großen Funktionsumfang für die Erstellung und Ausführung von Testfällen unter Verwendung von TTCN-3. Es ist auf Linux und Windows lauffähig. Die vorhandenen Anbindungen an Testplattformen sind auf den Telekommunikation-Sektor ausgerichtet.

PROVetech [79] ist ein Tool der Firma MBtech. Das Tool dient der Testautomatisierung im Automotive-Bereich. Es wird hierbei Basic als Sprache zur Testerstellung eingesetzt. Es steht eine integrierte Entwicklungsumgebung, mit einer Vielzahl von Bibliotheken zur Anbindung von Testsystem und Testtools, zur Verfügung.

OpenTTCN3 [80] ist ein Tool der Firma OpenTTCN Ltd. Es steht eine Entwicklungs- und Ausführungsumgebung für Testfälle in der Sprache TTCN-3 zur Verfügung. Die vorhandenen Anbindungen an Testplattformen sind auf den Telekommunikation-Sektor ausgerichtet.

Espresso ist ein kommerzieller TTCN-3 Compiler der Firma Metarga GmbH, welcher auf Arbeiten des Fraunhofer FIRST [81] basiert. Das Tool verwendet eine Umsetzung von TTCN-3 nach C# und kompiliert den C#-Code zu einem ausführbaren Test. Das Tool benötigt Windows als Plattform.

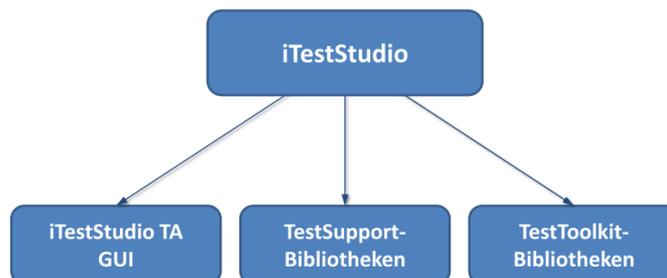
3.2.3. Firmeninterne Entwicklungen

In den folgenden Abschnitten wird eine Übersicht über eine Auswahl von firmeninternen Tools zur Testautomatisierung gegeben. Dabei ist zu beachten, dass hier nur eine kleine Auswahl betrachtet werden kann, da in vielen Firmen die verschiedenen Abteilungen jeweils eigene Lösungen verwenden und weiterentwickeln.

3.2.3.1. iSyst iTestStudio

Das Testautomatisierungs-Tool iTestStudio wurde und wird durch die Firma iSyst Intelligente Systeme GmbH entwickelt. Es kommt im Rahmen von Testdienstleistung in verschiedenen Projekten und bei verschiedenen Firmen zum Einsatz. Im Gegensatz zu den meisten kommerziellen Tools, wie sie in Kapitel 3.2.2 erläutert wurden, besteht das iTestStudio aus einer einfachen Oberfläche für die Testausführung und einer Sammlung von Bibliotheken für die Testerstellung, die Testausführung und das Test-Reporting. Die Abbildung 3.5 gibt eine Übersicht über die Komponenten.

Abbildung 3.5.: Aufbau des iTestStudio



Die Implementierung der Tests erfolgt in Python. Das iTestStudio bietet dafür keine eigene Entwicklungsumgebung an. Es kommt in den meisten Fällen Eclipse mit der Erweiterung PyDev als Entwicklungsumgebung zum Einsatz. Es kann auch jede andere Entwicklungsumgebung für Python eingesetzt werden. Für die Unterstützung der Testentwicklung

stehen eine Reihe von Bibliotheken zur Testbewertung und zur Anbindung an Testtools, wie zum Beispiel Vector CANape, dSPACE ControlDesk, Softing INCA oder National Instruments LabView, zur Verfügung. Diese werden als TestToolkit bezeichnet. Eigene Bibliotheken können mit Python erstellt werden. Mittels der Testsupport-Bibliotheken wird eine Umgebung für die Testausführung und die Erstellung von Test-Reports zur Verfügung gestellt. Es ist dabei möglich die erstellten Tests in Form von Python-Skripten einzeln auszuführen.

Die Strukturierung der Tests erfolgt in Form von Testserien, welche mehrere Testsequenzen enthalten können. Die Testsequenzen wiederum enthalten eines oder mehrere Python-Skripte zur Ausführung. Das Ergebnis jedes einzelnen Test-Skriptes wird am Ende gespeichert und zusammengefasst. Daraus wird ein Test-Report im XLS-, HTML- oder PDF-Format generiert. Es ist dabei auch möglich einzelne Test-Skripte nach zu testen und die Ergebnisse in einen kompletten Test-Report zu integrieren. Damit ist es auch möglich Tests auf verschiedenen Testsystemen parallel laufen zu lassen und einen gemeinsamen Testbericht zu erzeugen.

Die Test-Reports enthalten eine umfangreiche Darstellung von Metriken sowie Meta-Informationen zu jedem einzelnen Test-Skript.

Vorteile:

- Ausführbarkeit einzelner Test
- Debug-Möglichkeiten durch Python als Sprache
- Gute Erweiterungsmöglichkeiten
- Einfache Anbindung verschiedener Testtools

Nachteile:

- Keine integrierte Entwicklungsumgebung
- Geringer Dokumentationsumfang

3.2.3.2. Continental TA2/TA3

Die Testautomatisierungs-Tools TA2/TA3 der Firma Continental stammen historisch gesehen aus der Firma Siemens VDO, welche jetzt zu Continental gehört. Die beiden Tools wurden über lange Zeit getrennt entwickelt, verwenden aber mittlerweile die gleiche Codebasis. Dabei stellt TA3 die eigentliche Basis dar, und TA2 verwendet nur ein eigenes Frontend für die Testkonfiguration und Testausführung

In TA3 stehen eine Reihe von Bibliotheken für die Anbindung von Testtools, wie zum Beispiel dSPACE ControlDesk, Vector CANape oder Softing INCA, zur Verfügung. Eigene Bibliotheken können mittels Python erstellt werden. Die Erstellung der Tests erfolgt in Python-Skripten auf Basis des Test Automation Frameworks. Die Testskripte müssen eine feste Struktur aus Testvorbereitung, Testausführung und Testauswertung in Form von getrennten Funktionen aufweisen. TA3 bieten keine eigene Entwicklungsumgebung. Es

wird meist Eclipse mit der Erweiterung PyDev eingesetzt.

Die generierten Test-Reports bestehen aus einer Vielzahl von Dateien, welche während des Testablaufes in verschiedenen Ordner gespeichert werden. Die Menge der ausgegebenen Informationen lässt sich über Report-Level steuern.

Basierend auf den Bibliotheken von TA3 verwendet TA2 ein eigenes Frontend, welches es ermöglicht eine Anzahl von Standard-Testfällen mittels XML-Dateien zu parametrieren und auszuführen. Die Konfiguration der XML-Dateien erfolgt in grafischer Form unter Verwendung des Tools *Authentic* der Firma Altova. Es ist bei TA2 nicht vorgesehen eigene Tests in Form von Python-Skripten zu erstellen.

Vorteile:

- Debug-Möglichkeiten durch Python als Sprache
- Gute Erweiterungsmöglichkeiten
- Einfache Anbindung verschiedener Testtools

Nachteile:

- Keine integrierte Entwicklungsumgebung
- Geringer Dokumentationsumfang

3.2.4. Freie Entwicklungen

In den folgenden Abschnitten wird eine Übersicht über eine Auswahl von freien Tools zur Testautomatisierung gegeben.

3.2.4.1. TRex

TRex [82] ist ein Plugin für Eclipse, entwickelt von der Universität Göttingen. Das Plugin dient der Entwicklung von TTCN-3 Tests (inklusive Refactoring und Metriken). Neben der Unterstützung der Testentwicklung in TTCN-3 bietet TRex auch weitreichende Möglichkeiten zur statischen Analyse des Codes. Es eignet sich als IDE (Integrated Development Environment) für die Entwicklung von TTCN-3 Testfällen. Es erfolgt eine stetige Weiterentwicklung. Der Code ist im Rahmen des Projektes zugänglich. TRex bietet selbst keine Möglichkeit zur Ausführung von TTCN-3 Tests. Es besteht die Möglichkeit TTCN-3 Ausführungsumgebungen anzubinden. Dies ist für die Tools OpenTTCN und Danet TTCN-3 Toolbox realisiert.

3.2.4.2. LoongTesting

LoongTesting [83] ist ein TTCN-3 Compiler der University of Science and Technology of China. Die Plattform bietet eine Umsetzung von TTCN-3 nach C++. Das Tool benötigt

Windows als Plattform. Der Compiler selbst ist Closed-Source. Der Umfang der gebotenen Dokumentation ist gering.

Die vorhandenen Anbindungen an Testplattformen sind auf den Telekommunikation-Sektor ausgerichtet.

3.2.4.3. Weitere freie Entwicklungen

Das BroadBit Test Tool (BTT) [84] ist ein Open-Source-TTCN-3 Interpreter auf Basis von PHP. Es wird nur eine Untermenge von TTCN-3 unterstützt. Weiterhin ist das Tool nur für den Test von Netzwerkprotokollen ausgelegt. Eine Erweiterung auf andere Anwendungsgebiete wird durch fehlende Dokumente erschwert.

PicoTTCN [85, 86] ist ein Open-Source TTCN-3 Compiler, welcher im Rahmen des Projektes Go4IT [87] entwickelt wurde. Das Projekt wurde seit 2008 nicht mehr aktualisiert und der Compiler wurde bisher nicht fertig gestellt. Eine Dokumentation ist nicht zugänglich.

Ttthreeparser [88] wurde in der Version 1.4 geprüft. Es ist ein auf Java und ANTLR basierender Parser, welcher keinerlei Logik zur Ausführung von TTCN-3 mitbringt. Weiterhin existiert keine Dokumentation. Es wird nur die Version 1.0.1 (aktuell Version 4.5.1) von TTCN-3 aus dem Jahr 2001 unterstützt.

TET (Test Environment Toolkit) [66] stellt ein Werkzeug zur Testausführung und zum Testmanagement dar. Es ist nur für den Test von Software auf eingebetteten Systemen mit Betriebssystemen, nach dem Standard POSIX (Standard for Embedded Realtime Systems POSIX 1003.13) oder Embedded Linux Systemen, ausgelegt.

3.2.5. Grenzen und Probleme verfügbarer Testautomatisierungs-Tools

Wie in den vorangegangenen Abschnitten beschrieben wurde, existieren viele verschiedenen Testautomatisierungs-Tools und ein weites Spektrum an Anforderungen an die Testautomatisierungs-Tools, welche von keinem Tool vollständig erfüllt werden (siehe dazu auch Abschnitt 3.2). Jedes Testautomatisierungs-Tool verwendet proprietäre Schnittstellen sowie Speicherformate für die Tests und Test-Reports. Dies verhindert zum Großteil eine Wiederverwendung von Testfällen. Weiterhin ist es nicht möglich, Test-Reports verschiedener Testautomatisierungs-Tools gemeinsam zu verarbeiten.

Selbst bei Verwendung derselben Testautomatisierungs-Tools in verschiedenen Projekten ist die Wiederverwendung der Testfälle schwierig. Einerseits gibt es Kompatibilitätsprobleme zwischen verschiedenen Versionen des selben Tools. Andererseits muss oft das Zeitverhalten der Testfälle auf die verwendete PC-Hardware angepasst werden, da die verwendeten, Windows-basierten PC-Systeme kein akkurates Zeitverhalten liefern. Dies bedeutet, dass die Ausführung des selben Testfalles je nach Auslastung des PC-Systems ein unterschiedliches zeitliches Verhalten und gegebenenfalls unterschiedliche Testergeb-

nisse liefert.

Ein weiteres Problem stellen die Schnittstellen zu den Mess-, Kalibrier- und Diagnose-tools bzw. Testtools im Allgemeinen dar. Die Änderung eines der Testtools bzw. auch dessen Version macht meist eine Anpassung der Testfälle nötig. Weiterhin sind die meisten für die Tests notwendigen Testtools nicht in ihrem vollen Funktionsumfang unterstützt. Als Beispiel kann hier ECU-TEST 4.5 dienen, welches den Echtzeitfetch des dSPACE-Systems nicht unterstützt und auch keine zyklisches Messen mit CANape bereitstellt. Bei EXAM 2.0 gibt es Probleme, wenn das dSPACE-System mehr als eine Prozessorkarte enthält. In diesem Fall ist der Echtzeitstimulus nicht nutzbar.

Die Testausführung auf dem in Kapitel 2.3.1.4 beschriebenen HIL-Systemen erfolgt in zwei verschiedenen Situationen. Auf der einen Seite werden einzelne Testfälle aus dem jeweiligen Tool, wie zum Beispiel Eclipse, EXAM oder ECU-TEST, zur Testentwicklung ausgeführt. Auf der anderen Seite wird, meist über Nacht oder auch über längere Zeiträume, eine größere Anzahl von Testfällen automatisiert hintereinander ausgeführt. Dies wird entwicklungsbegleitend oder im Vorfeld von Abgaben für Releases durchgeführt.

Die Art der Gruppierung und Ausführung mehrerer Testfälle ist toolabhängig. ECU-TEST und AutomationDesk bieten eine einstufige Organisation. Es ist möglich einzelne Testfälle in Gruppen zusammenzufassen. Bei AutomationDesk und ECU-TEST werden diese Gruppen als Testprojekte bezeichnet.

EXAM bietet hingegen ein zweistufiges System. Dort können Testfälle (TestCases) in sogenannte TestSuites zusammengefasst werden. TestSuites wiederum können in sogenannte TestCampaigns zusammengefasst werden. Die Steuerung des Testablaufs erfolgt durch die Testautomatisierung bzw. die einzelnen Testfälle auf dem PC-System des HIL-Systems. Es erfolgt im Normalfall keine Ablaufsteuerung durch das verwendete Echtzeitsystem. Weiterhin werden für die Testabläufe nötige Busschnittstellen an das PC-System angebunden und über entsprechende Software, wie zum Beispiel Vector CANape oder Vector CANalyzer, Signale der Bussysteme gemessen. Es wird auch abgesetztes Testequipment (z. B. Diagnosesystem oder Busanbindung) über Ethernet angebunden.

Die Bewertung von Signalverläufen erfolgt nach der Durchführung eines Testfalls. Es wird während des Testfalls der Verlauf von einem oder mehreren Signalen auf die Festplatte aufgezeichnet. Nach Abschluss des Tests werden die Daten automatisch bewertet. Die mangelhafte Anbindung der Versionsmanagement-Werkzeuge erschwert die Reproduzierbarkeit der Tests. Es ist sinnvoll zu jedem versionierten Softwarestand des Produktes auch den zugehörigen Stand der Tests im Versionsmanagement-Werkzeug abzulegen. Die Tests sollten weiterhin in für den Menschen lesbarer Form (z. B. Python oder XML) vorliegen um auch die Möglichkeit des Vergleichs (Erstellung eines Diff) zwischen verschiedenen versionierten Ständen zu ermöglichen. EXAM verwendet für die Speicherung der Testfälle und Bibliotheken eine Datenbank, die sich nur schwer zurück auf einen vorhergehenden Stand setzen lässt. AutomationDesk verwendet für die Speicherung der Projekte ein proprietäres, binäres Format.

Die Testautomatisierungs-Tools bieten oft auch keine – oder nur eine rudimentäre Unterstützung der Entwicklung von Testfällen. Es fehlt oft eine Möglichkeit zum Debuggen

bzw. zum schrittweisen Ausführen der Testfälle. Als Beispiele sind hier EXAM 2 ohne jegliche Debug-Möglichkeit und AutomationDesk mit nur sehr beschränkter schrittweise Ausführung von Testfällen zu nennen. Bei den Testautomatisierungs-Tools mit einstufiger Gruppierung von Tests, wie AutomationDesk oder ECU-TEST, ist es schwierig, mehrere Testprojekte automatisiert auszuführen. Es ist so meist nötig, eine große Anzahl von Tests in ein Testprojekt zu packen, was meist der Übersichtlichkeit des Testprojektes und des Test-Reports abträglich ist.

3.3. Testdokumentation

3.3.1. Standards

Zur Dokumentation des Testprozesses sind verschiedene Dokumente, wie zum Beispiel Master Test Plan, Level Test Plan oder Level Test Design, notwendig. Für die Arten und Inhalte der Dokumente existieren verschiedene Vorgaben, wie zum Beispiel IEEE 829, ISO/IEC 29119 oder ISO/IEC 15504 (SPICE). Dabei kommen die Vorgaben nach IEEE 829 [89] verbreitet zum Einsatz. Dieser Standard wird durch die ISO/IEC 29119 [90] abgelöst. Die Inhalte sind dabei vergleichbar.

Der Standard beschreibt dabei die folgenden Dokumente:

- Master Test Plan
- Level Test Plan
- Level Test Design
- Level Test Case
- Level Test Procedure
- Level Test Log
- Anomaly Report
- Level Interim Test Status Report
- Master Test Report

Im Automotive-Bereich kommen weitere Vorgaben zum Tragen. Hierbei sei der Standard ISO/IEC 15504 (SPICE) in der Adaption Automotive SPICE zu nennen. Dieser Standard beschreibt die Bewertung von Entwicklungs- und Testprozessen. Im Rahmen dieser Bewertung werden auch entsprechende Vorgaben bezüglich der im Testprozess zu erstellenden Dokumente und deren Inhalte gemacht.

Im Bezug auf diese Arbeit sei hier vor allem die Forderung nach einer durchgehenden Nachverfolgbarkeit – Traceability – von den Anforderungen über die Testspezifikation bis zu den Testergebnissen zu nennen. Des Weiteren sind hier Angaben zum Autor des Test, zu Änderungen und Version des Tests zu berücksichtigen.

3.3.2. Inhalte

Für die Betrachtung des Inhaltes der Testautomatisierung und des damit verbundenen Test-Reports bzw. Testprotokolls – wie er innerhalb dieser Arbeit betrachtet wird – sind die Vorgaben für den Level Test Log und den Master Test Report nach IEEE 829 zu betrachten. Dabei ist herauszustellen, dass sich die genannten Angaben nicht ausschließlich auf den automatisierten Test beziehen. Die hier dargestellten Anforderungen und Standards gelten für alle Testaktivitäten. Es sind ebenfalls die Vorgaben bezüglich der Nachverfolgbarkeit nach Automotive SPICE zu berücksichtigen. Daraus ergeben sich die folgenden notwendigen Punkte für den Test-Report:

- Eindeutige Kennung des Test-Reports
- Beschreibung des Testfalls und der Testschritte
- Angabe der Ergebnisse von Testschritten
- Zuordnung des Testfälle zu Anforderungen (z. B. eindeutige ID)
- Autor des Tests
- Version des Testfalls inklusive Angaben zu Änderungen
- Metriken über die Anzahl der Testfälle und die Ergebnisse

3.3.3. Formate

Die in den vorhergehenden Abschnitten genannten Standards machen hierbei keine Angaben über das Format bzw. über die Art der Dokumentation. Die Dokumentation kann dabei digital in Dateien oder Datenbanken bzw. auch handschriftlich erfolgen. Die Form der Darstellung ist dabei freigestellt. Es ist ebenfalls möglich, dass mehrere der genannten Dokumente in einem physikalischen Dokument zusammengefasst sind, solange alle notwendigen Informationen enthalten sind.

Weiterhin wurden Standards für Dateiformate zur Speicherung von Testergebnissen bzw. von Testdokumentationen recherchiert. Dabei hat sich gezeigt, dass dazu keine anerkannten Standards auffindbar sind. Einzig das TAP [65] beschreibt ein einfaches Textformat.

Das Format ist wie folgt gegliedert:

```
<Bewertung> <Nummer des Testschrittes> – <Beschreibung>
```

Die folgenden Zeilen zeigen beispielhaft mögliche Ergebnisse eines Tests unter Verwendung von TAP:

```
ok 1 – Input file opened
not ok 2 – Invalid input.
```

Der Standard bietet keine Möglichkeit der Gliederung von Testfällen und keinerlei Meta-Informationen (z. B. Anforderungs-ID¹).

3.3.4. Praktische Umsetzungen

Wie in den vorangegangenen Kapiteln schon angedeutet, unterscheidet sich der Umfang der Testauswertung und des Test-Reports von Testautomatisierungs-Tool zu Testautomatisierungs-Tool. Dabei ist auch eine klare Abweichung von den verfügbaren Standards festzustellen. Die Bandbreite reicht von einem Test-Report zur Dokumentation des Testablaufes ohne die Möglichkeit, Bilder einzubinden, wie es ECU-TEST bietet, bis hin zu umfangreich bearbeitbaren Test-Reports wie dies EXAM bietet.

Die verbreitetsten Dateiformate zur Ausgabe von Testberichten sind HTML (z. B. AutomationDesk, ECU-TEST, iTestStudio) und PDF (z. B. AutomationDesk, EXAM). Weiterhin gibt es teilweise die Möglichkeit, den Test-Report in Form einer XML-Datei zu erhalten. Bei AutomationDesk wird die XML-Datei als Basis für die Erzeugung des HTML- oder PDF-Reports verwendet. Bei EXAM besteht die Möglichkeit, XML-Dateien zu exportieren. Diese XML-Dateien sind nicht standardisiert. Jeder Toolhersteller hat seine eigene XML-Datenstruktur spezifiziert.

Der Abgleich der Ergebnisse mit Requirementsmanagement-Werkzeugen ist nicht vorgesehen. Es sind zwar zum Teil Schnittstellen vorhanden, aber nicht einsatzfähig bzw. auf ein spezielles Tool zugeschnitten. Die zur Verfügung gestellten Test-Reports und deren Funktionsumfang unterscheiden sich von Testautomatisierungs-Tool zu Testautomatisierungs-Tool. Es stellt schon ein Problem dar, eine Abbildung (z. B. grafische Darstellung eines Signalverlaufes) in einen Test-Report einzubinden (z. B. ECU-TEST und EXAM). Auch die Visualisierung von Verläufen aufgezeichneter Signale ist zum Teil nicht möglich (z. B. ECU-TEST). Selbst die komfortable Ausgabe von mehrzeiligen Texten zur Beschreibung des Testfalls oder des eigentlichen Testschrittes ist schwierig (z. B. ECU-TEST). Die erstellten Test-Reports erscheinen durch den Einsatz vieler Farben unübersichtlich und sind teilweise schwer lesbar (z. B. EXAM (PDF), ECU-TEST (HTML), AutomationDesk (HTML)). Weiterhin führt die Verwendung von JavaScript und Frames bei dem HTML-Test-Report zu Problemen mit verschiedenen Browsern und zu Kollisionen mit den Sicherheitseinstellungen der Browser auf Firmenrechnern.

¹Anforderungs-ID – Eindeutige Identifikationsnummer für eine Anforderung

Ein weiteres Problem, welches bei dem HTML-Report auftritt, ist die Versionierung. Der HTML-Report besteht aus mehreren Dateien (HTML-Dateien und teilweise Bilddateien). Diese Dateien werden meist mit dynamisch erzeugten Namen versehen. Es ist damit nicht möglich, diese Test-Reports direkt in ein Versionsmanagement-Werkzeug einzuchecken und von dort aus auch lesbar zu halten. Das lesbar halten, bedeutet dabei die korrekte Anzeige im Browser sowie die korrekte Funktion von Links. Es bleibt nur der Weg die Dateien zu packen und anschließend in das Versionsmanagement-Werkzeug ein zu pflegen. Dies hat den Nachteil, dass der Test-Report nicht direkt aus dem Versionsmanagement-Werkzeug zu öffnen ist.

Besteht ein Test aus mehreren Testsequenzen (EXAM: TestSuite) so ist es entweder nicht möglich, mehrere Testsequenzen automatisiert auszuführen, oder es wird für jede Testsequenz ein eigener Test-Report erzeugt. Dies erschwert einerseits die Strukturierung von Tests und andererseits die Auswertung der Ergebnisse des gesamten Tests. Weiterhin erschweren mehrere Test-Reports das Versionsmanagement noch weiter. Die Test-Reports bieten nur eine rudimentäre statistische Auswertung des Testablaufes. Eine Auswertung der Ergebnisse mit der Anzahl der PASSED und FAILED der durchgeführten Testfälle ist normalerweise vorhanden. Eine Auswertung der Laufzeiten der einzelnen Testfälle existiert nicht. Dies wäre für die Planung von Tests mit Laufzeiten von mehreren Tagen nötig. Bei der Testdurchführung auf realen Systemen kann es durch Umgebungseinflüsse (z. B.: Installation von Updates auf dem Test-PC oder Abstürzen von Komponenten der Testsoftware) nötig sein einzelne Testfälle nach zu testen. Es ist meist nicht möglich, diese einzelnen Tests in einen Gesamttestreport zu integrieren. Es ist nötig, jeden Einzelnen nach zu testenden Testfall mit einem eigenen Bericht, im Versionsmanagement-Werkzeug einzupflegen. Es geht damit die Zusammengehörigkeit, des Test-Reports verloren.

Es besteht nur vereinzelt die Möglichkeit, die Ergebnisse der Tests mit den Anforderungen in Requirementsmanagement-Werkzeug zu verknüpfen und die Ergebnisse der Testfälle zurück in das Requirementsmanagement-Werkzeug abzugleichen. Die Automatisierung dieses Vorgehens ist meist nicht gegeben oder bestenfalls durch firmeninterne Speziallösungen realisiert.

3.4. Anbindung der Testautomatisierung an den Entwicklungsprozess

Wie schon in Kapitel 2.2 dargestellt, existiert eine enge Verknüpfung zwischen den Entwicklungs- und Testaktivitäten. Es ist daher notwendig, eine Integration des Testprozessen und damit der Testautomatisierung in den Entwicklungsprozess zu realisieren. Um dies transparent zu ermöglichen, ist die Integration der Testaktivitäten und der damit verbundenen Toollandschaft in die Toollandschaft der Entwicklung notwendig. In den folgenden Abschnitten werden die zentralen Schnittstellen hierfür dargestellt.

3.4.1. Anbindung an das Requirementsmanagement

Die für die Erstellung von Tests wichtigste Schnittstelle, ist die Schnittstelle zum Anforderungsmanagement bzw. Requirementsmanagement. Die Anforderungen stellen die Basis für Testfälle – vor allem auf der Ebene des Systemtests – dar. Des Weiteren erfordert die in Kapitel 3.3.1 erläuterte Nachverfolgbarkeit eine Verknüpfung von den Anforderungen mit den Testfällen und den Ergebnissen. Diese Verknüpfung sollte in möglichst automatisierter Form erfolgen.

Die Verwaltung von Anforderungen erfolgt in Requirementsmanagement-Werkzeugen wie zum Beispiel IBM Doors, MKS Integrity oder HP Quality Center. Der Austausch der Anforderungen zwischen OEM und Zulieferer erfolgt in Textdokumenten wie zum Beispiel Microsoft Word oder PDF. Diese Dokumente werden aus dem Requirementsmanagement-Werkzeug generiert und automatisiert bzw. manuell wieder eingepflegt. In seltenen Fällen kommt das Austauschformat ReqIF (Requirements Interchange Format) zum Einsatz.

Da es für die Anbindung von Testtools an das Requirementsmanagement keine einheitlichen Schnittstellen gibt, wird die Anbindung projektspezifisch bzw. firmenspezifisch realisiert. Es werden dabei aus der Struktur der Anforderungen entsprechende Testfälle im Testautomatisierungs-Tool erstellt, wie die zum Beispiel in Kapitel 3.2.2.2 für EXAM beschrieben wird. Eine automatisierte Verknüpfung der Testergebnisse mit den Anforderungen erfolgt dabei nicht.

Die geforderte Nachverfolgbarkeit wird in der Praxis über manuell geführte Listen, z. B. in Microsoft Excel, realisiert.

3.4.2. Anbindung an das Changemanagement

Das Changemanagement bzw. Änderungsmanagement ist für die Dokumentation, von im Test gefundenen Fehlern bzw. Anomalien, notwendig. Das Changemanagement erfolgt ebenfalls toolbasiert mit z. B. MKS Integrity oder Trac. Eine automatisierte Verknüpfung zwischen dem Test bzw. den Testergebnissen und dem Changemanagement ist nicht bekannt und erscheint auch nicht sinnvoll, da die Ergebnisse bewertet werden müssen. Nach der Bewertung ist eine manuelle Überführung der gefundenen Anomalien in das Changemanagement möglich.

3.4.3. Anbindung an das Versionsmanagement

Das Versionsmanagement ist zur Definition und Wiederherstellung eindeutiger Versionen einer Software oder eines Tests notwendig. Die Verwaltung von Softwareentwicklungsprojekten mittels Versionsmanagement-Werkzeugen ist Stand der Technik. Es kommen hier Tools wie MKS Source Integrity, Microsoft Team Foundation Server oder Apache Subversion zum Einsatz. Ziel ist es definierte Versionen von Dateien zu speichern und

diese jeder Zeit wieder herstellen zu können. Außerdem ist es möglich definierte Stände (Baselines) über eine Vielzahl von Dateien zu realisieren. Dies wird für die Zuordnung von Dateiversionen der Quelltextdateien zu Softwareständen verwendet. Dies ist für die Reproduzierbarkeit und Nachverfolgbarkeit unabdingbar.

Die genannten Punkte gelten hierbei nicht nur für die Softwareentwicklung. Es ist für den Test, vor allem beim Einsatz automatisierter Tests, notwendig definierte Versionen und Stände zu speichern und zu reproduzieren. Zum Beispiel um Tests für ältere Stände der Software wiederholen zu können oder Änderungen von Tests zu dokumentieren.

Wie in dem Kapitel 3.2 beschrieben, besitzen die meisten Testautomatisierungs-Tools keine Schnittstellen zum Versionsmanagement. Die Einbindung kann daher nur über die gespeicherten Dateien der Testautomatisierungs-Tools erfolgen. Dabei erschweren binäre Formate die Verwendung des Versionsmanagement, da zum Beispiel keine sinnvollen Vergleiche mittels Diff-Tools möglich sind.

Die Integration zwischen Testautomatisierungssystemen und Versionsmanagement ist bei textbasierten Dateiformaten, wie sie beispielsweise der Quelltext in Python darstellt, am einfachsten, da das Versionsmanagement für die Verwaltung von Quelltext entwickelt wurde.

Für die effektive Integration in den Entwicklungsprozess ist eine möglichst transparente Integration des Tests in das Versionsmanagement der Entwicklung notwendig, um eine gemeinsame Versionierung zu ermöglichen.

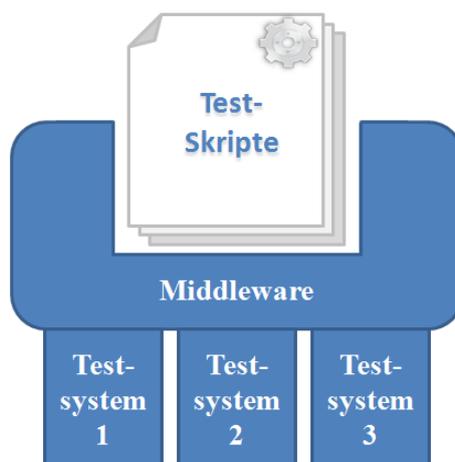
4. Konzept der Middleware

Die konzeptionellen Überlegungen zur modularen Middleware für den automatisierten Test werden in diesem Kapitel dargelegt. Aus den in Kapitel 2 und 3 angeführten Anforderungen und Problemen werden in diesem Kapitel die Zielsetzungen für die modulare Middleware abgeleitet und die Realisierungsmöglichkeiten erarbeitet. Dabei wird im ersten Teil ein allgemeines Konzept für die Aufteilung der Funktionalitäten in einzelne Module erarbeitet. Danach werden die Konzepte der einzelnen Module ausgearbeitet und im Anschluss das Gesamtkonzept erläutert.

4.1. Allgemeines Konzept

In diesem Abschnitt wird das allgemeine Konzept für die modulare Middleware dargestellt, welches aus den Anforderungen und Problemen, die in Kapitel 2 und 3 dargestellt wurden, erarbeitet wurde. Es sind dabei vor allem der modulare Ansatz für die Middleware sowie die explizite Einbeziehung des Requirements- und Versionsmanagement zu beachten. Des Weiteren steht die Abstraktion zwischen Testbeschreibung und der Testausführung auf der einen Seite und dem Testsystem auf der anderen Seite im Mittelpunkt, wie dies in Abbildung 4.1 verallgemeinert dargestellt ist.

Abbildung 4.1.: Allgemeiner Funktion der Middleware



4.1.1. Vorbetrachtung

In den Kapiteln 2 und 3 wurden eine Vielzahl von Aspekten bezüglich des Software-Tests und der Testautomatisierung dargestellt. Im ersten Schritt sollen die wichtigsten Punkte an dieser Stelle nochmals zusammengefasst werden und die Aufgaben der Middleware herausgearbeitet und abgegrenzt werden.

Als erstes ist hierfür der Entwicklungsprozess zu beleuchten, da die Tests in diesem Kontext eingebettet sind. Je nach eingesetztem Entwicklungsprozess, wie diese in Kapitel 2.2 dargestellt sind, unterscheiden sich die Anzahl der Testdurchführungen. Von der – eher theoretischen – einmaligen Testdurchführung, wie sie im Wasserfall-Modell vorgesehen ist, bis hin zu einer sehr häufigen Testdurchführung, wie sie in den Iterativ-Inkrementellen Modellen vorgesehen ist, muss eine sinnvolle Unterstützung durch das Testautomatisierungssystem gegeben sein. Auch die Anzahl der Teststufen (z. B. Modultest, Integrations-test, Systemtest und Abnahmetest) und deren Testumfänge sind unterschiedlich. Weiterhin soll auch die Wiederverwendbarkeit der Testfälle im Fertigungsendtest möglich sein. Des Weiteren ist aus der Betrachtung des Produktlebenszyklus in Kapitel 2.2.2 die Lebensdauer bzw. die Einsatzdauer eines Testautomatisierungssystems und des Testsystems zu berücksichtigen. Hier besteht die Notwendigkeit, Tests bis zum Ende der Produktionsdauer eines Produktes wiederholen zu können. Sollte ein Testsystem nicht mehr verfügbar sein, so erlaubt eine einfache Portierbarkeit der Testfälle eine schnelle Wiederherstellung der Testbarkeit.

Es lassen sich daraus die folgenden Aufgaben ableiten:

- Unterstützung einer effizienten Testerstellung und Testanpassung, um einen schnellen Nutzen aus der Automatisierung der Tests, auch bei wenigen Testdurchführungen zu erzielen
- Unterstützung der Testerstellung für verschiedene Teststufen
- Unterstützung der Testerstellung für verschiedene Testsysteme

Weiterführend sind die Aufgaben aus dem Testprozess, wie sie in Kapitel 2.2.6 beschrieben sind, zu untersuchen. Hier stehen, neben den Aufgaben aus dem Entwicklungsprozess, vor allem Aufgaben des Testmanagement im Mittelpunkt. Um eine sinnvolle und effektive Testdurchführung zu gewährleisten, ist eine hohe Flexibilität der Testorganisation und Testausführung notwendig. Bei der Durchführung von Regressionstests muss es zum Beispiel möglich sein, nur einen Teil der Testfälle auszuführen. Weiterhin ist es notwendig auch definierte ältere Stände der Testfälle ausführen zu können. Für eine effektive Auswertung und Verwaltung der Testergebnisse ist es notwendig, dass die Ergebnisse in Form von möglichst einem Test-Report vorhanden sind. Weiterhin müssen die Test-Reports in maschinenlesbarer Form für die automatisierte Weiterverarbeitung vorliegen. Es ist auch eine menschenlesbare Form für die Auswertung und Bewertung notwendig.

Die sich hieraus ergebenden Aufgaben sind im Folgenden aufgelistet:

- Unterstützung einer flexiblen Zusammenstellung und Gliederung der Testfälle
- Unterstützung einer Anbindung an Versionsmanagement-Systeme, für das Management verschiedener Testfallversionen
- Unterstützung der Abspeicherung von Testergebnissen in eine Test-Report-Datenbasis, welche maschinelle verarbeitet werden kann
- Unterstützung der Integration von manuellen Testergebnissen in die Test-Report-Datenbasis
- Unterstützung der Generierung menschenlesbarer Testports auf Basis einer gemeinsamen Datenbasis

Als weiterer Schritt sind die formalen Anforderungen an die Prozesse, wie sie durch Automotive SPICE in Kapitel 2.2.5 gegeben sind, zu berücksichtigen. Als zentrale Punkte sind dabei die Nachverfolgbarkeit sowie die konsistente Dokumentation zu nennen. Bei der Nachverfolgbarkeit ist die – möglichst automatisierte – Verknüpfung der Anforderungen mit der Implementierung und dem Test bis hin zu den Testergebnissen gemeint. Dies kann nicht alleine durch das Testautomatisierungssystem erreicht werden. Es müssen hierfür die notwendigen Schnittstellen zu dem Requirementsmanagement-System sowie die notwendigen Informationen im Test und im Test-Report bereitstellen können, um die automatisierte Nachverfolgbarkeit möglich zu machen. Die konsistente Dokumentation bezieht sich einerseits auf die eindeutige Zuordnung des Test-Reports zu einem Testobjekt und andererseits auf die Inhalte, wie zum Beispiel Versionsnummer eines Tests, Name des Testerstellers, Name des Testdurchführers, Änderungshistorie des Tests und weitere Angaben, wie sie in Kapitel 3.3.2 erläutert sind. Daraus wurden die folgenden Aufgaben abgeleitet:

- Unterstützung einer automatisierten Schnittstelle zum Requirementsmanagement-System inklusive Abgleich der Testergebnisse
- Unterstützung der Verarbeitung von Anforderungs-IDs im Test und im Test-Report
- Unterstützung von Meta-Daten, wie zum Beispiel Versionsnummer eines Tests, Name des Testerstellers, Name des Testdurchführers und Änderungshistorie des Tests im Test-Report und der Testverwaltung

Im nächsten Schritt sind die formalen Anforderungen für die Dokumentation, wie sie in Kapitel 3.3.1 beschrieben ist, zu betrachten. Hierbei gibt es Überschneidungen mit den Anforderungen aus Automotive SPICE. Zusätzlich zu den dortigen Anforderungen sind weitere Informationen, wie eine Beschreibung jedes Testschrittes mit einer Angabe der Bewertung (z. B. Passed oder Failed) notwendig. Des Weiteren müssen zu jedem Testfall die Testziele angegeben sein.

Damit ergeben sich diese Zielsetzungen:

- Unterstützung von Beschreibungen auf verschiedenen Gliederungsebenen des Test-Reports.
- Unterstützung der Bewertung jedes Testschrittes

Ebenso sind die Anforderungen des Testentwicklers bzw. des Testingenieurs zu berücksichtigen. Im Gegensatz zu den in den vorangegangenen Abschnitten beschriebenen, technischen Anforderungen kommen nun die praktischen Notwendigkeiten zum Tragen. Für den Testentwickler ist ein möglichst einfacher Umgang mit der Testautomatisierung notwendig. Dies umfasst eine gut bedienbare Entwicklungsumgebung für die Testfälle. Weiterhin ist eine flexible Programmierung der Testfälle notwendig. Dies umfasst eine einfach zu erlernende möglichst vielseitige Programmiersprache. Für die Testentwicklung kommen auch Personen mit Domänenwissen zum Einsatz, welche möglicherweise mit Programmiersprachen nicht umgehen können. Es ist daher auch die grafische Erstellung von Testfällen nötig.

Für die effektive Entwicklung eines Testfalls ist die Möglichkeit zum Debuggen bzw. zur schrittweisen Ausführung von Testfällen notwendig. Die gebotenen Funktionen sollten mit Debuggern aus der Softwareentwicklung vergleichbar sein.

Bei der Durchführung automatisierter Tests ist auch eine hohe Robustheit des Testautomatisierungssystems notwendig. Ein fehlerhafter Testfall oder eine unerwartete Reaktion des DUT dürfen nicht zu einem Abbruch des gesamten Tests führen, da die automatisierten Tests typischerweise ohne Aufsicht ablaufen.

Die daraus ermittelten Aufgaben sind wie folgt:

- Unterstützung von programmatischer und modellbasierter bzw. grafischer Testentwicklung
- Unterstützung einer einfach zu erlernenden Programmiersprache mit großem Funktionsumfang
- Unterstützung des Debugging von Testfällen
- Unterstützung der Ausführung einzelner Testfälle
- Unterstützung eines robusten Fehlerhandlings von Fehlern des Testsystems und des DUT durch das Testautomatisierungssystem

Weiterhin werden die Anforderungen des Testmanagers, zusätzlich zu den allgemeinen Anforderungen des Testmanagements, betrachtet. Für die Erfüllung der praktischen Aufgaben des Testmanagers ist es notwendig eine schnelle Übersicht über die Testergebnisse zu erlangen. Daher ist es notwendig, dass der Testport Metriken über die Anzahl der Testfälle bzw. Testschritte sowie über die Anzahl der Passed und Failed zur Verfügung stellt. Weiterhin muss eine schnelle Zuordnung zu Anforderungen und möglichen Safety-Einstufungen möglich sein, was ebenfalls über Metriken realisierbar ist. Als letztes ist die Angabe und Auswertung der Testlaufzeiten notwendig, um dem Testmanager die Möglichkeit der Planung weiterer Testausführungen zu erlauben.

Daraus haben sich die folgenden Aufgaben ergeben:

- Unterstützung von Metriken in den Test-Reports (z. B. Übersichten über die Anzahl der Testfälle und Anzahl der Passed und Failed)
- Unterstützung der Erweiterbarkeit der Metriken
- Unterstützung der Laufzeitmessung der Testfälle inklusive Angaben im Test-Report

Als letzter Schritt sind die Anforderungen des Softwareentwicklers zu betrachten. Auf den ersten Blick scheint es hier keinen direkten Zusammenhang zu geben. Entsprechend der Vorgaben der Entwicklungs- und Testprozesse endet die Betrachtung mit dem Testbericht bzw. mit der Aufnahme der Fehler in ein Defect Management System, welches häufig ein Teil des Requirements und Changemanagement Systems ist. Für die Reproduktion und Behebung der Fehler ist es notwendig, dass auch der Softwareentwickler in der Lage ist die Testabläufe anhand des Test-Reports nachzuvollziehen. Daher ist es aus Sicht des Softwareentwicklers notwendig, dass der Test-Report eine entsprechend sinnvolle Strukturierung sowie eine Beschreibung der Testschritte und Testfälle zulässt.

Es wurden damit die folgenden Aufgaben ermittelt:

- Unterstützung einer ausführlichen Testschritt- und Testfallbeschreibung
- Unterstützung einer flexiblen Ausgabe von Informationen in den Test-Report

Zum Abschluss soll hier eine genauere Abgrenzung zwischen dem betrachteten Testautomatisierungssystem und dem Testsystem gemacht werden. Die in Abschnitt 2.3.1.4 beschriebenen hohen zeitlichen Anforderungen, wie sie zum Beispiel durch FPGA-Karten realisiert werden, werden hier als Teil des Testsystems betrachtet. Das Testsystem muss die entsprechenden Anforderungen an das Zeitverhalten, welche durch das DUT gestellt werden, erfüllen. Das Testautomatisierungssystem hingegen muss die notwendigen Zeitbedingungen für die Testausführung realisieren können.

4.1.2. Anforderungen

Auf Basis der im vorangegangenen Kapitel formulierten Aufgaben werden in diesem Kapitel konkrete Anforderungen an die modulare Middleware formuliert. Im Mittelpunkt stehen dabei die folgenden, zentralen Punkte und Neuheiten dieser Arbeit:

- Modularität des Systems
- Durchgängigkeit der Nachverfolgbarkeit
- Umfangreiche Testdokumentation bzw. Testreporting
- Flexibilität bezüglich der umsetzbaren Testfälle und der unterstützten Testsysteme

In den folgenden Abschnitten werden die Aufgaben Kategorien zugeordnet und einzeln, im Bezug auf die abzuleitenden Anforderungen, betrachtet. Die große Zahl abgeleiteter Anforderungen soll dabei das große Spektrum zu erfüllender Aufgaben und eine mög-

lichst detaillierte Zielsetzung aufzeigen. Die Zuordnung zu den Kategorien ist dabei nicht immer eindeutig möglich. Die Aufgaben werden daher der Kategorie zugeordnet, wo die stärkste Zugehörigkeit besteht.

4.1.2.1. Testerstellung und -entwicklung

Als erste Kategorie soll hier die *Testerstellung und -entwicklung* betrachtet werden. In dieser Kategorie sind die folgenden Aufgaben zugeordnet worden, welche im Anschluss einzeln betrachtet werden:

- Unterstützung einer effizienten Testerstellung und Testanpassung, um einen schnellen Nutzen aus der Automatisierung der Tests auch bei wenigen Testdurchführungen zu erzielen
- Unterstützung einer einfach zu erlernenden Programmiersprache großen Funktionsumfangs
- Unterstützung der Testerstellung für verschiedene Teststufen
- Unterstützung von programmatischer und modellbasierter bzw. grafischer Testentwicklung
- Unterstützung des Debugging von Testfällen

Die Aufgabe *Unterstützung einer effizienten Testerstellung und Testanpassung, um einen schnellen Nutzen aus der Automatisierung der Tests auch bei wenigen Testdurchführungen zu erzielen* und *Unterstützung einer einfach zu erlernenden Programmiersprache großen Funktionsumfangs* sind eng miteinander verbunden und sollen hier gemeinsam betrachtet werden. Eine effiziente Testerstellung und Testanpassung geht einher mit einer gut lesbaren und verständlichen Programmiersprache. Dies bedingt damit auch eine einfache Erlernbarkeit der Sprache. Ein großer Funktionsumfang geht wiederum einher mit einer schnellen Anpassbarkeit von Testfällen, da die notwendigen Funktionen durch die Sprache bereitgestellt werden. Daraus wurden die folgenden, in Tabelle 4.1 dargestellten, Anforderungen abgeleitet.

Tabelle 4.1.: Anforderungsliste – Testerstellung und -entwicklung Teil I

Anforderungsnummer	Beschreibung
Anforderung 1	Verwendung einer höheren Programmiersprache mit einfacher Syntax
Anforderung 2	Verwendung einer Programmiersprache mit großem Funktionsumfang

Die Aufgabe *Unterstützung der Testerstellung für verschiedene Teststufen* erfordert die Erstellung von Testfällen in verschiedenen Abstraktionsgraden. Für den Modultest sind

einzelne Werte in hoher zeitlicher Auflösung zu erfassen und auszuwerten. Bei Systemtests sind komplexe Zusammenspiele vieler Signale und ihrer Verläufe zu erfassen, wobei im Automotive-Bereich geringere zeitliche Anforderungen gestellt werden, da es um die Erfassung von durch den Fahrer erlebbare Symptome geht. Des Weiteren stellt diese Aufgabe auch Anforderung an die Anbindung der Testsysteme. Dies wird in Kapitel 4.1.2.5 diskutiert. Hier ließen sich damit die weiteren, in Tabelle 4.2 aufgeführten, Anforderungen ermitteln.

Tabelle 4.2.: Anforderungsliste – Testerstellung und -entwicklung Teil II

Anforderungsnummer	Beschreibung
Anforderung 3	Verwendung einer Programmiersprache mit verschiedenen Abstraktionsgraden
Anforderung 4	Verwendung einer Programmiersprache zur Auswertung einzelner Werte und von Signalverläufen
Anforderung 5	Verwendung einer Programmiersprache mit der Unterstützung einer hohen Ausführungsgeschwindigkeit

Die Aufgabe *Unterstützung von programmatischer und modellbasierter bzw. grafischer Testentwicklung* erfordert es, dass eine bidirektionale Umwandlung von grafischen Modellen in ausführbaren Quelltext und von Quelltext in eine grafische Darstellung möglich ist. Für eine effiziente Entwicklung ist der gleichzeitige Einsatz beider Methoden bei der Entwicklung notwendig. Es wurden die weiteren, in Tabelle 4.3 aufgeführten, Anforderungen formuliert.

Tabelle 4.3.: Anforderungsliste – Testerstellung und -entwicklung Teil III

Anforderungsnummer	Beschreibung
Anforderung 6	Verwendung einer Programmiersprache mit der Möglichkeit zur modellbasierten bzw. grafischen Darstellung und Änderung des Programmablaufes
Anforderung 7	Verwendung von gängigen Darstellungsmethoden, wie zum Beispiel UML
Anforderung 8	Parallele Verwendung von verschiedenen Darstellungsmethoden

Die Aufgabe *Unterstützung des Debugging von Testfällen* fordert die Möglichkeit zur schrittweisen Ausführung von Testfällen mit den Fähigkeiten eines modernen Debuggers aus der Softwareentwicklung, wie zum Beispiel Haltepunkte, Ansicht und Änderung der

Variablen im Speicher. Damit ließen sich die weiteren, in Tabelle 4.4 aufgeführten, Anforderungen bestimmen.

Tabelle 4.4.: Anforderungsliste – Testerstellung und -entwicklung Teil IV

Anforderungsnummer	Beschreibung
Anforderung 9	Verwendung einer Programmiersprache mit Debug-Möglichkeit
Anforderung 10	Verwendung einer Entwicklungsumgebung mit Debug-Möglichkeit

4.1.2.2. Testausführung

Als zweite Kategorie soll hier die *Testausführung* betrachtet werden. In dieser Kategorie sind die folgenden Aufgaben zugeordnet worden, welche im Anschluss einzeln betrachtet werden:

- Unterstützung eines robusten Fehlerhandlings durch das Testautomatisierungssystem
- Unterstützung der Ausführung einzelner Testfälle
- Unterstützung der Bewertung jedes Testschrittes
- Unterstützung der Laufzeitmessung der Testfälle inklusive Angaben im Test-Report
- Unterstützung einer flexiblen Ausgabe von Informationen in den Test-Report

Die Aufgabe *Unterstützung eines robusten Fehlerhandlings durch das Testautomatisierungssystem* erfordert, auf der einen Seite eine detailliert spezifizierte Programmiersprache (z. B. definiertes Verhalten bei der Verwendung falscher Datentypen) mit einer umfangreichen Fehlerbehandlung bzw. Exception-Handling. Auf der anderen Seite besteht auch für die Testausführung im Testautomatisierungssystem die Aufgabe möglichst fehlertolerant zu sein. Selbst unbehandelte Exceptions in einzelnen Tests dürfen den Gesamt Ablauf der Tests nicht gefährden. Weiterhin muss die Testautomatisierungsumgebung die Möglichkeit bieten, das Testsystem einfach in einen definierten Zustand zu versetzen. All dies hat zum Ziel, fehlerhafte Bewertungen durch den Test zu vermeiden und eine stabile, automatisierte Ausführung der Tests zu gewährleisten.

Daraus wurden die weiteren, in Tabelle 4.5 aufgeführten, Anforderungen erkannt.

Tabelle 4.5.: Anforderungsliste – Testausführung Teil I

Anforderungsnummer	Beschreibung
Anforderung 11	Verwendung einer Programmiersprache mit umfangreichen Exception-Handling
Anforderung 12	Verwendung einer Programmiersprache zur einfache Umsetzung von Testfällen
Anforderung 13	Behandlung von Fehlern und Exceptions im Testautomatisierungssystem
Anforderung 14	Bereitstellung von Funktion(en) um das Testsystem in den Default-Zustand zu versetzen
Anforderung 15	Automatische Herstellung des Default-Zustandes nach Abbruch eines Testfalls (z. B. durch unbehandelte Exception)

Die Aufgabe *Unterstützung der Ausführung einzelner Testfälle* erfordert die Unterstützung durch das Testautomatisierungssystem. Schon beim Design des Testautomatisierungssystem und der zugehörigen Bibliotheken muss berücksichtigt werden, dass Testfälle einzeln lauffähig sein müssen. Es muss daher eine Unabhängigkeit vom Testmanagement des Testautomatisierungssystem realisiert werden. Es ergaben sich damit die weiteren, in Tabelle 4.6 aufgeführten, Anforderungen.

Tabelle 4.6.: Anforderungsliste – Testausführung Teil II

Anforderungsnummer	Beschreibung
Anforderung 16	Bibliotheken sind unabhängig vom Testautomatisierungssystem lauffähig
Anforderung 17	Testfälle sind unabhängig vom Testmanagement des Testautomatisierungssystem lauffähig
Anforderung 18	Ein Testfall muss ein eigenständig ausführbares Programm darstellen

Die Aufgabe *Unterstützung der Bewertung jedes Testschrittes* bezieht sich auf der einen Seite auf die Programmierung der Testfälle. Hier müssen Möglichkeiten für die Bewertung und Ausgabe der Bewertung in den Test-Report für jeden Testschritt gegeben sein. Auf der anderen Seite muss der Test-Report die Bewertung und die Darstellung der Bewertung einzelner Testschritte ermöglichen, wobei diese Anforderungen in Kapitel 4.1.2.3 betrachtet werden. Hierbei kann ein Testschritt die Bewertung eines einzelnen Zahlenwertes darstellen. Daraus ließen sich die weiteren, in Tabelle 4.7 aufgeführten, Anforderungen bestimmen.

Tabelle 4.7.: Anforderungsliste – Testausführung Teil III

Anforderungsnummer	Beschreibung
Anforderung 19	Bereitstellung von Funktionen zur Bewertung einzelner Testschritte
Anforderung 20	Bereitstellung von Funktionen zur Ausgabe der Bewertung einzelner Testschritte in den Test-Report

Die Aufgabe *Unterstützung der Laufzeitmessung der Testfälle inklusive Angaben im Test-Report* erfordert die Umsetzung während der Testausführung. Es müssen dabei beim Starten und beim Beenden eines Testfalls die aktuelle Zeit bestimmt werden. Weiterhin muss diese Information in den Test-Report ausgegeben werden, siehe dazu die Betrachtung in Kapitel 4.1.2.3. Es wurden damit die weiteren, in Tabelle 4.8 aufgeführten, Anforderungen erarbeitet.

Tabelle 4.8.: Anforderungsliste – Testausführung Teil IV

Anforderungsnummer	Beschreibung
Anforderung 21	Erfassung der Laufzeit einzelner Testfälle durch das Testautomatisierungssystem
Anforderung 22	Weitergabe der Laufzeit an den Test-Report

Die Aufgabe *Unterstützung einer flexiblen Ausgabe von Informationen in den Test-Report* steht in engem Zusammenhang mit den Aufgaben für den Test-Report in Kapitel 4.1.2.3. Für die Testausführung ist es wichtig, dass die notwendigen Schnittstellen für die Weitergabe der Informationen an des Test-Reportings berücksichtigt werden. Daraus wurden die weiteren, in Tabelle 4.9 aufgeführten, Anforderungen abgeleitet.

Tabelle 4.9.: Anforderungsliste – Testausführung Teil V

Anforderungsnummer	Beschreibung
Anforderung 23	Bereitstellung von Funktionen für die Weitergabe von Informationen von der Testausführung zum Test-Reporting
Anforderung 24	Berücksichtigung von Informationen für Testbewertung, Testbeschreibungen, Messdaten und Meta-Daten (z. B. Eindeutige ID für die getestete Anforderung)

4.1.2.3. Test-Reporting

Als dritte Kategorie soll hier die *Testausführung* betrachtet werden. In dieser Kategorie sind die folgenden Aufgaben zugeordnet worden, welche im Anschluss einzeln betrachtet werden:

- Unterstützung von Beschreibungen auf verschiedenen Gliederungsebenen des Test-Reports
- Unterstützung einer ausführlichen Testschritt- und Testfallbeschreibung
- Unterstützung vom Meta-Daten, wie zum Beispiel Versionsnummer eines Tests, Name des Testerstellers, Name des Testdurchführers und Änderungshistorie des Tests im Test-Report und der Testverwaltung
- Unterstützung der Verarbeitung von Anforderungs-IDs im Test und im Test-Report
- Unterstützung der Abspeicherung von Testergebnissen in eine Test-Report-Datenbasis, welche maschinell verarbeitet werden kann
- Unterstützung der Integration von manuellen Testergebnissen in die Test-Report-Datenbasis
- Unterstützung der Generierung menschenlesbarer Test-Reports auf Basis einer gemeinsamen Datenbasis
- Unterstützung von Metriken in den Test-Reports (z. B. Übersichten über die Anzahl der Testfälle und Anzahl der Passed und Failed)
- Unterstützung der Erweiterbarkeit der Metriken

Die Aufgaben *Unterstützung von Beschreibungen auf verschiedenen Gliederungsebenen des Test-Reports* und *Unterstützung einer ausführlichen Testschritt- und Testfallbeschreibung* erfordert die Möglichkeit der Ausgabe von Beschreibungen auf verschiedenen Gliederungsebenen, bis hin zum einzelnen Testschritt. Dabei ist es für einen Testbericht sinnvoll, dass jede Beschreibung auch mit einer Bewertung versehen werden kann. Damit konnten die weiteren, in Tabelle 4.10 aufgeführten, Anforderungen bestimmt werden.

Die Aufgabe *Unterstützung vom Meta-Daten, wie zum Beispiel Versionsnummer eines Tests, Name des Testerstellers, Name des Testdurchführers und Änderungshistorie des Tests im Test-Report und der Testverwaltung* und *Unterstützung der Verarbeitung von Anforderungs-IDs im Test und im Test-Report* stellen zusammen Anforderungen an die Berücksichtigung von Meta-Daten, welche nicht direkt zur Testbeschreibung und Bewertung gehören, dar. Hieraus ließen sich die weiteren, in Tabelle 4.11 aufgeführten, Anforderungen ableiten.

Die Aufgaben *Unterstützung der Abspeicherung von Testergebnissen in eine Test-Report-Datenbasis, welche maschinell verarbeitet werden kann* und *Unterstützung der Integration von manuellen Testergebnissen in die Test-Report-Datenbasis* sind ebenfalls gemeinsam zu betrachten. Sie stellen weitere Anforderungen an die Ablage der Testergebnisse

Tabelle 4.10.: Anforderungsliste – Test-Reporting Teil I

Anforderungsnummer	Beschreibung
Anforderung 25	Bereitstellung verschiedener Gliederungsebenen im Test-Report
Anforderung 26	Bereitstellung von Beschreibungen auf den verschiedenen Gliederungsebenen im Test-Report
Anforderung 27	Bereitstellung von Bewertungen zu den Beschreibungen auf allen Gliederungsebenen
Anforderung 28	Bereitstellung eines Bewertungsschemas mit Priorisierung (z. B. Error, Failed, Passed und Info)

Tabelle 4.11.: Anforderungsliste – Test-Reporting Teil II

Anforderungsnummer	Beschreibung
Anforderung 29	Bereitstellung einer erweiterbaren Schnittstelle für Meta-Daten
Anforderung 30	Speicherung der Meta-Daten im Testbericht

sowie an die zur Verfügung zu stellenden Schnittstellen für die Test-Reports. Darauf aufbauend ließen sich die weiteren, in Tabelle 4.12 aufgeführten, Anforderungen ermitteln.

Tabelle 4.12.: Anforderungsliste – Test-Reporting Teil III

Anforderungsnummer	Beschreibung
Anforderung 31	Bereitstellung eines Speicherformates für das Test-Reporting, welches maschinell verarbeitbar ist
Anforderung 32	Bereitstellung eines Speicherformates für das Test-Reporting, welches manuell ergänzbar ist

Die Aufgabe *Unterstützung der Generierung menschenlesbarer Testports auf Basis einer gemeinsamen Datenbasis* stellt die zentrale Zielsetzung für das Test-Reporting dar. Als Ergebnis ist ein für Menschen lesbarer Test-Reports notwendig. Je nach Bedarf müssen dabei verschiedene Formate unterstützt werden. Basierend hierauf ließen sich die weiteren, in Tabelle 4.13 aufgeführten, Anforderungen erkennen.

Tabelle 4.13.: Anforderungsliste – Test-Reporting Teil IV

Anforderungsnummer	Beschreibung
Anforderung 33	Bereitstellung einer Umwandlung des maschinell verarbeitbaren Test-Report in Menschenlesbarer Form
Anforderung 34	Unterstützung verschiedener Formate, wie zum Beispiel PDF, HTML und XLS, für den Test-Report

Die Aufgaben *Unterstützung von Metriken in den Test-Reports* (z. B. *Übersichten über die Anzahl der Testfälle und Anzahl der Passed und Failed*) und *Unterstützung der Erweiterbarkeit der Metriken* beziehen sich beide auf die Bereitstellung von Auswertungen und Metriken innerhalb des Test-Reports. Es ist dabei zu bearbeiten, dass je nach Erfordernis, verschiedene Metriken notwendig sind. Daraus konnten die weiteren, in Tabelle 4.14 aufgeführten, Anforderungen erkannt werden.

Tabelle 4.14.: Anforderungsliste – Test-Reporting Teil V

Anforderungsnummer	Beschreibung
Anforderung 35	Bereitstellung von Funktionen zur Erstellung von Metriken
Anforderung 36	Bereitstellung von Funktionen zur Konfiguration der Metriken

4.1.2.4. Testmanagement

Als vierte Kategorie soll hier die *Testmanagement* betrachtet werden. In dieser Kategorie sind die folgenden Aufgaben zugeordnet worden, welche im Anschluss einzeln betrachtet werden:

- Unterstützung einer flexiblen Zusammenstellung und Gliederung der Testfälle
- Unterstützung einer Anbindung an Versionsmanagement-Systeme für das Management verschiedener Testfallversionen
- Unterstützung einer automatisierten Schnittstelle zum Requirements-management-System inklusive Abgleich der Testergebnisse

Die Aufgabe *Unterstützung einer flexiblen Zusammenstellung und Gliederung der Testfälle* steht einerseits in Verbindung mit der Gliederung des Test-Reports, wie dies in Kapitel 4.1.2.3 erläutert ist. Andererseits ist die Zusammenstellung der Testfälle in ihrer Reihenfolge sowie die Gliederung ein Teil des Testmanagement des Testautomatisierungssystems. Damit ließen sich die weiteren, in Tabelle 4.15 aufgeführten, Anforderungen bestimmen.

Tabelle 4.15.: Anforderungsliste – Testmanagement Teil I

Anforderungsnummer	Beschreibung
Anforderung 37	Bereitstellung von Funktionalität zum Ausführen der Testfälle in beliebiger Reihenfolge
Anforderung 38	Bereitstellung von Funktionalität zur hierarchischen Gliederung der Testfälle in mehreren Ebenen

Die Aufgabe *Unterstützung einer Anbindung an Versionsmanagement-Systeme für das Management verschiedener Testfallversionen* ist für das Management der Testausführung im Bezug auf verschiedene Versionen des zu testenden DUT notwendig. Das Testmanagement sowie die Testausführung müssen die Ausführung älterer Testfälle in der früheren Reihenfolge und Gliederung ermöglichen. Daraus ergaben sich die weiteren, in Tabelle 4.16 aufgeführten, Anforderungen.

Tabelle 4.16.: Anforderungsliste – Testmanagement Teil II

Anforderungsnummer	Beschreibung
Anforderung 39	Speicherung der Ausführungsreihenfolge der Testfälle
Anforderung 40	Bereitstellung einer Schnittstelle zum Versionsmanagement-System
Anforderung 41	Sicherstellung der Ausführbarkeit älterer Versionsstände von Testfällen inklusive der früheren Ausführungsreihenfolge

Die Aufgabe *Unterstützung einer automatisierten Schnittstelle zum Requirementsmanagement-System inklusive Abgleich der Testergebnisse* bezieht sich einerseits auf die Testausführung und das Test-Reporting, welche die notwendigen Informationen für die Verknüpfung mit dem Requirementsmanagement-System verarbeiten müssen. Auf der anderen Seite muss das Testmanagement eine Schnittstelle zum Requirementsmanagement-System bieten, um einen bidirektionalen Abgleich der Informationen für die Nachverfolgbarkeit zu ermöglichen. Hier waren die weiteren, in Tabelle 4.17 aufgeführten, Anforderungen zu ermitteln.

Tabelle 4.17.: Anforderungsliste – Testmanagement Teil III

Anforderungsnummer	Beschreibung
Anforderung 42	Bereitstellung einer bidirektionalen Schnittstelle zu Requirementsmanagement-Systemen
Anforderung 43	Bereitstellung von Funktionen zur Verknüpfung von Testfällen mit Anforderungen (z. B. Anforderungs-ID)
Anforderung 44	Bereitstellung von Funktionen zum Abgleich von Testergebnissen mit Anforderungen
Anforderung 45	Bereitstellung von Funktionen zum Abgleich von Beschreibungen (z. B. Testfallbeschreibung)

4.1.2.5. Testsystemanbindung

Als fünfte und letzte Kategorie soll hier die *Testsystemanbindung* betrachtet werden. In dieser Kategorie ist die folgende Aufgabe zugeordnet worden:

- Unterstützung der Testerstellung für verschiedene Testsysteme in den verschiedenen Teststufen

Die Aufgabe *Unterstützung der Testerstellung für verschiedene Testsysteme in den verschiedenen Teststufen* bezieht sich vor allem auf die Testfallerstellung und Testausführung des Testautomatisierungssystems. Für eine Unterstützung verschiedener Testsysteme, sowie eine schnelle Anbindung neuer Testsysteme in verschiedenen Teststufen, sind einheitliche Schnittstellen zum Testautomatisierungssystem zu schaffen. Basierend darauf waren die weiteren, in Tabelle 4.18 aufgeführten, Anforderungen zu bestimmen.

Tabelle 4.18.: Anforderungsliste – Testsystemanbindung Teil I

Anforderungsnummer	Beschreibung
Anforderung 46	Bereitstellung einheitlicher Schnittstellen für Testsysteme
Anforderung 47	Bereitstellung abstrahierter Funktionen für den Testsystemzugriff auf Testfallebene

4.1.3. Modulkonzept

Zur Realisierung der Zielsetzung der Entwicklung einer flexibel einsetzbaren Middleware, hat sich früh die Notwendigkeit für einen modularen Ansatz gezeigt. Die

im Kapitel 3.2 dargestellten monolithischen Systeme mit den damit verbundenen Problemen zeigen die Notwendigkeit einer modularen Lösung klar auf.

Ebenso sind durch die Anforderungen verschiedener Entwicklungsprozesse, wie sie in Kapitel 2.2 erläutert wurden, sowie die Integration in die Tool-Landschaften modular austauschbare Komponenten der Middleware notwendig. Hier sei beispielhaft die Anbindung an Requirementsmanagement-Systeme zu nennen, bei denen es eine Vielzahl von zu unterstützender Systemen und Schnittstellen gibt.

Durch die Betrachtung und Kategorisierung der Anforderungen in Kapitel 4.1.2 wurden wichtige funktionale Gruppen für die Middleware herausgearbeitet. Bei genauerer Betrachtung lassen sich daraus drei Hauptkomponenten bzw. Module für die Middleware ableiten, welche zwingend zur Erfüllung der Aufgaben einer Testautomatisierung notwendig sind, aber selbst eine abgeschlossene Aufgabe erfüllen und somit auch autark einsetzbar sind.

Das erste abgeschlossene Modul ist die Testbeschreibung bzw. das Test-Beschreibungs-Modul, welches in Kapitel 4.2 genauer erläutert wird. Dieses Modul umfasst die Erstellung von Tests in programmatischer, wie auch grafischer Form.

Das zweite Modul ist die Testausführung bzw. das Test-Ausführungs-Modul, welches in Kapitel 4.3 beschrieben ist. Dieses Modul umfasst die Ausführung der Tests sowie alle notwendigen Schnittstellen zur Anbindung der Testsysteme.

Das dritte Modul ist die Testdokumentation bzw. das Test-Report-Generator-Modul, wie es in Kapitel 4.4 dargestellt ist. Dieses Modul umfasst die Erstellung von Test-Reports sowie die Generierung von Metriken inklusive des Abgleiches mit dem Requirementsmanagement-System.

Neben der Aufteilung der Anforderungen auf einzelne Module sind die Schnittstellen zwischen den Modulen ein wichtiger Teil des Konzeptes. Es stehen hierfür zwei prinzipielle Lösungsmöglichkeiten zur Verfügung. Die erste Lösungsmöglichkeit ist eine Funktionsschnittstelle zwischen den jeweiligen Modulen. Die zweite Lösungsmöglichkeit ist der Einsatz von Dateien als Speicher für die auszutauschenden Informationen. Es wurden die zentralen Eigenschaften dieser beiden Lösungsmöglichkeiten erarbeitet. Diese sind in Tabelle 4.19 gegenübergestellt.

Tabelle 4.19.: Gegenüberstellung der Lösungsmöglichkeiten für die Modulschnittstellen

Kriterien	Funktions-schnittstelle	Datei-basierte Schnittstelle
Betriebssystemeinabhängigkeit	-	++
Direktheit des Datenaustausches	++	0
Datenaustausch über Systemgrenzen	-	++
Flexibilität	+	++

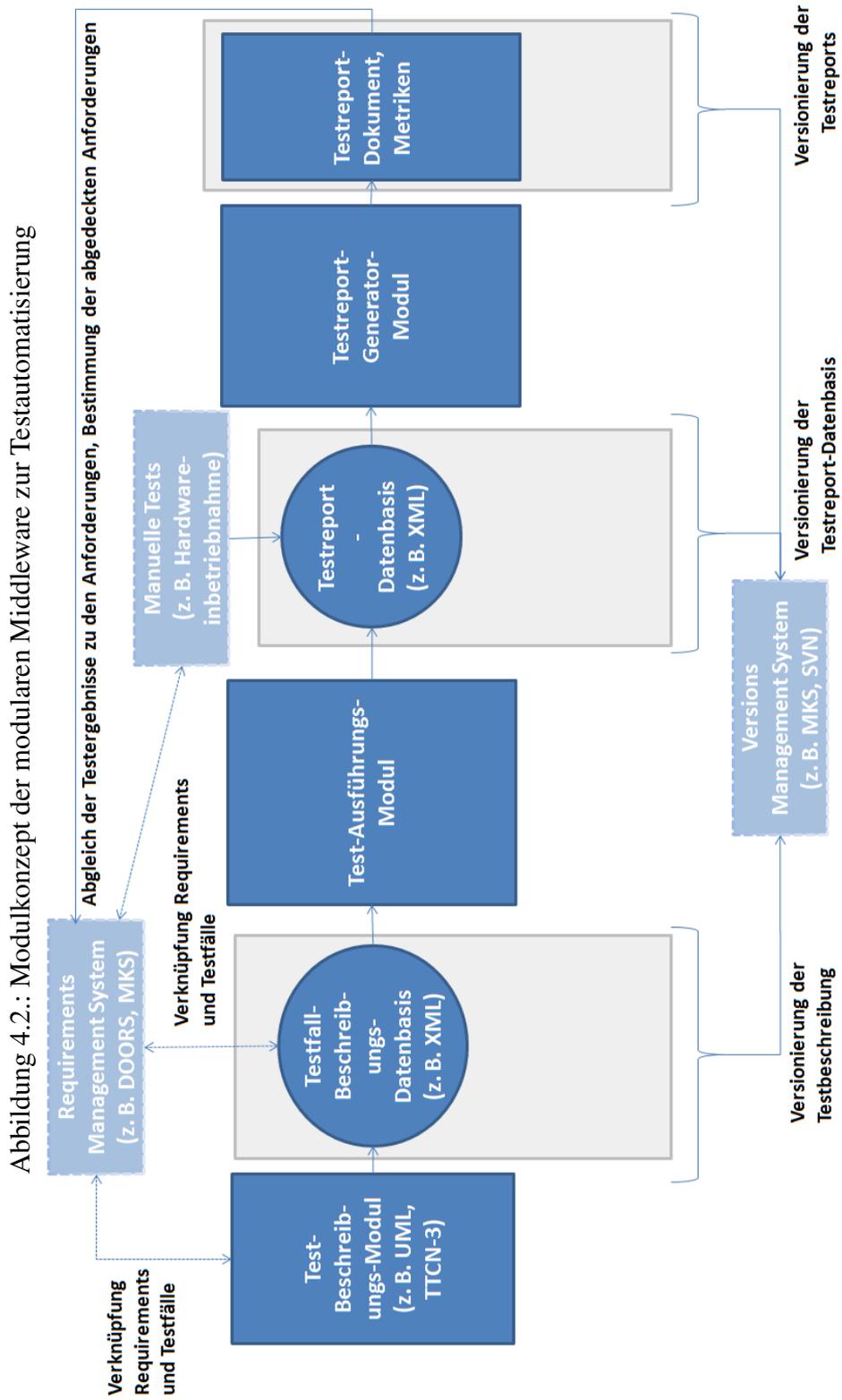
Der vorrangigste Nachteil einer Funktionsschnittstelle stellt die Abhängigkeit vom jeweiligen Betriebssystem dar. Eine typische Realisierung einer Funktionsschnittstelle erfolgt

über Bibliotheken (z. B. DLL-Datei unter Windows oder SO-Datei unter Linux). Es besteht dabei die Notwendigkeit, diese Schnittstellen für jedes der zu unterstützenden Systeme neu zu erzeugen. Weiterhin müssen die Module auf dem selben System laufen, um Daten austauschen zu können oder es müssen Middleware-Systeme, wie zum Beispiel Corba, für den Aufruf von Funktionen über Systemgrenzen hinweg eingesetzt werden. Eine Datei basierte Schnittstelle ist hier deutlich besser geeignet. Die Erstellung von Dateien ist unter verschiedenen Betriebssystemen mit standardisierten Funktionen, wie beispielsweise *fopen* unter C, möglich. Weiterhin können Dateien zwischen verschiedenen Systemen ausgetauscht werden. Außerdem hat die Verwendung von Dateien als Austauschformat zur Folge, dass diese Dateien direkt in Versionsmanagement-Systeme archiviert werden können und somit auch die Anforderung 40 aus Tabelle 4.16 direkt erfüllbar ist.

4.1.4. Aufbau des Gesamtsystems

Basierend auf den Betrachtungen aus den vorangegangenen Kapitel wurde ein erstes Konzept für die Realisierung der modularen Middleware erstellt. In Abbildung 4.2 ist das Konzept erarbeitet.

In der Darstellung ist, neben den drei Modulen Test-Beschreibungs-Modul, Test-Ausführungs-Modul und Test-Report-Generator-Modul, auch die vorgesehene Anbindung von Requirementsmanagement-Systemen und Versionsmanagement-Systemen sowie die Einbindung manueller Tests ersichtlich. Das hier dargestellte Konzept für das Gesamtsystem wurde im Rahmen der Arbeit entwickelt und stellt den Ausgangspunkt für die weiteren Betrachtungen dar. In den folgenden Kapiteln werden die einzelnen Module der Middleware genauer betrachtet und auf Basis der Anforderungen ein detailliertes Lösungskonzept für jedes einzelne Modul erarbeitet und diskutiert.



4.2. Testbeschreibung

Im Folgenden wird die Testbeschreibung betrachtet. Es werden hier verschiedene Lösungsvarianten diskutiert und das gewählte Lösungskonzept dargestellt.

4.2.1. Zielsetzung

Die Zielsetzung ist die Konzeptionierung eines Test-Beschreibungs-Moduls entsprechend des in Kapitel 4.1.3 und in Kapitel 4.1.4 dargestellten Aufbaues.

4.2.2. Anforderungen

Die Anforderungen wurden in Kapitel 4.1.2 dargelegt und diskutiert. An dieser Stelle soll auf die, für die Testbeschreibung, relevanten Anforderungen verwiesen werden. Die hier zu berücksichtigenden Anforderungen sind in den Tabellen 4.1, 4.2, 4.3 und 4.4 aufgelistet. Des Weiteren sind die Anforderung 38 aus Tabelle 4.15 sowie die Anforderungen 42 und 43 aus Tabelle 4.17 zu berücksichtigen.

4.2.3. Lösungsvarianten

In diesem Abschnitt werden die erarbeiteten Lösungsvarianten für die Realisierung eines Test-Beschreibungs-Moduls diskutiert. Die folgenden Abschnitte beschreiben jeweils eine mögliche Lösungsvariante sowie deren Vor- und Nachteile.

4.2.3.1. Definition einer eigenen Testbeschreibung

Auf Basis der erarbeiteten Anforderungen ist es möglich, eine eigene Testbeschreibungsmethode zu realisieren. Dies wäre zum Beispiel über eine Domain-specific language (DSL) möglich. In dem Beitrag [91] wird diese Lösungsvariante beispielhaft für den Test von AUTOSAR-Software-Komponenten dargestellt.

Die Konzeptionierung und Realisierung einer DSL ist für die Lösung spezieller Problemstellungen, auch im Bereich des Tests, sinnvoll. Im Rahmen dieser Arbeit soll ein möglichst allgemeingültiger Lösungsansatz gefunden werden, welcher nicht nur die Testbeschreibung für einen kleinen Bereich der Testaufgaben abdeckt. Der Aufwand für diese Lösungsvariante ist daher sehr hoch.

Ebenfalls bedingt die Verwendung einer eigenen Sprache einen hohen Einarbeits- und Schulungsaufwand für die Einführung und praktische Anwendung.

4.2.3.2. Anwendung einer bestehenden Testbeschreibung

Auf Basis der erarbeiteten Anforderungen ist es möglich, bestehende Beschreibungsverfahren für Testfälle auf ihre Anwendbarkeit bzw. Adaptierbarkeit für dieses Konzept zu untersuchen. In den Kapiteln 3.1 und 3.2.1 wurden verschiedenste Standards und Methoden für die Testbeschreibung aus unterschiedlichen Bereichen dargestellt.

Die verschiedenen Standards und Methoden wurden, unter Berücksichtigung der Anforderungen, auf ihre Verwendbarkeit im Rahmen dieser Arbeit untersucht. Im Folgenden sollen zentrale Aspekte für die in Kapitel 3.1 dargestellten Standards und Methoden erläutert werden.

C# ist als ausdrucksstarke Programmiersprache für die Umsetzung von Tests einsetzbar. Es hat sich gezeigt, dass die Anforderungen nach einer modellbasierten bzw. grafischen Darstellung (Anforderung 6) und einer parallelen Verwendung der grafischen und textuellen Darstellung (Anforderung 8) von diesem Standard nicht erfüllt werden. Des Weiteren wird ein plattformübergreifender Einsatz durch proprietäre Erweiterungen von Microsoft erschwert.

XML ist als Metasprache für die Definition anwendungsspezifischer Sprachen nicht direkt einsetzbar. Es stellte sich im Rahmen der Untersuchungen heraus, dass die Verwendung von XML eine domainspezifische Umsetzung erfordern würde. Wie schon in Kapitel 4.2.3.1 dargelegt, ist dies mit einem hohen Aufwand verbunden.

Die Standards IEEE 1445, IEEE 1450 und IEEE 1671 sind für die Verwendung mit ATEs im Bereich der Halbleiterindustrie und des Hardwaretests vorgesehen und bieten damit nicht den notwendigen Funktionsumfang für die Tests, wie unter anderem durch Anforderung 2 gefordert. Weiterhin bieten diese Standards auch keine modellbasierte bzw. grafische Darstellung (Anforderung 6) und keine parallele grafische und textuelle Darstellung (Anforderung 8).

UML ermöglicht die grafische Modellierung von Systemen und deren Verhalten, was einen Einsatz im Testbereich als sinnvoll erscheinen lässt, wie dies in den Beiträgen [92] und [93] beispielhaft dargestellt ist. Da UML-Diagramme selbst nicht direkt ausgeführt werden können, muss eine Umsetzung in eine ausführbare Sprache erfolgen. Dies ist nicht Teil des UML-Standards. Damit ist die Realisierung der Anforderungen 1 und 2 nach der Verwendung einer höheren Programmiersprache sowie der Anforderung 9 und 10 nach Debug-Möglichkeiten nur aufwändig realisierbar.

Matlab/Simulink ermöglicht die grafische Modellierung von Systemen und deren Verhalten, was einen Einsatz im Testbereich als sinnvoll erscheinen lässt. Außerdem ist die Generierung von ausführbarem Code möglich. Die verwendete Art der Modellierung ist auf die Realisierung von zyklisch auszuführenden Prozessen, wie zum Beispiel Reglern, ausgelegt. Die Modellierung von sequenziellen Testabläufen ist damit nur schwer möglich. Außerdem ist eine Debug-Möglichkeit, wie in Anforderung 10 gefordert, aus Simulink heraus für generierten Code nicht gegeben. Eine parallele Verwendung der modellbasierten und der direkten textuellen Programmierung, wie in Anforderung 8 gefordert, ist nicht möglich.

Labview ermöglicht ebenfalls die grafische Modellierung von Systemen und deren Verhalten. Eine parallele Verwendung der modellbasierten und der direkten textuellen Programmierung, wie in Anforderung 8 gefordert, ist nicht möglich.

AutomationDesk der Firma dSPACE kommt für die Verwendung in dieser Arbeit nicht in Betracht, da nur eine Unterstützung von Testsystem auf Basis von Komponenten von dSPACE möglich ist. Eine parallele Verwendung der modellbasierten und der direkten textuellen Programmierung, wie in Anforderung 8 gefordert, ist ebenfalls nicht möglich. Die Untersuchungen im Rahmen dieser Arbeit ergaben, dass unter Berücksichtigung der Anforderungen an die Testbeschreibung allein Python, ISO 13209 und TTCN-3 für eine Verwendung zu betrachten sind. In Tabelle 4.20 sind die zentralen Ergebnisse der Untersuchung und die Eigenschaften der drei Beschreibungsmethoden dargestellt.

Tabelle 4.20.: Gegenüberstellung der Testbeschreibungsmethoden – Python, ISO 13209 und TTCN-3

Merkmal	Python	ISO 13209	TTCN-3
Stand der Standardisierung	<ul style="list-style-type: none"> • Seit 1991 in fortwährender Entwicklung • Version 2.7.6 bzw. 3.3.3 (Stand Januar 2014) 	<ul style="list-style-type: none"> • Version 1.0.0 (Stand 2012) 	<ul style="list-style-type: none"> • Seit 1983 in fortwährender Entwicklung • Version 3 seit 2000 veröffentlicht • aktuell: 4.5.1 (2013)
Einsatz-Domänen	<ul style="list-style-type: none"> • Allgemeine Programmiersprache 	<ul style="list-style-type: none"> • Automotive (Fahrzeugdiagnose) 	<ul style="list-style-type: none"> • Mobilfunkkommunikation [60] • Breitbandkommunikation [60] • Middleware Plattformen [60] • Internet Protokolle [60] • Smart Cards [60] • Automotive (AUTOSAR) [60]

4. KONZEPT DER MIDDLEWARE

Darstellungsformen	<ul style="list-style-type: none"> • Textuell – Python 	<ul style="list-style-type: none"> • Textuell – XML • Grafisch 	<ul style="list-style-type: none"> • Textuell – TTCN-3 (Core Language) • Grafisch – Ablaufdiagramm • Tabellarische Darstellung
Ausführbarkeit	<ul style="list-style-type: none"> • Ausführung durch Python Interpreter 	<ul style="list-style-type: none"> • Für Ausführbarkeit in einer Laufzeitumgebung entwickelt 	<ul style="list-style-type: none"> • Nicht vorrangig zur Ausführung entwickelt • Es existieren eine Vielzahl von Ausführungsumgebungen als Interpreter oder Compiler
Austauschbarkeit / Zusammenarbeit	<ul style="list-style-type: none"> • Keine standardisierten Schnittstellen zu Testsystemen und Tools • Einbindung verschiedenster Tools über COM-Interface und DLLs möglich 	<ul style="list-style-type: none"> • Standardisierte Schnittstelle zu Fahrzeugdiagnose vorhanden • Schnittstellen zu anderen Tools geplant • Nutzbarkeit durch mangelnde Umsetzung noch nicht bewiesen 	<ul style="list-style-type: none"> • Komplette Trennung der Testfälle über die TTCN-3 Laufzeitumgebung vom Testsystem. • Standardisierte Schnittstellen zu Adapter und Codecs • Adapter und Codecs müssen für jeweilige Testumgebung erstellt werden

Erweiterbarkeit	<ul style="list-style-type: none"> • Erstellung eigener Bibliotheken (Package) in Python möglich • Erweiterungen auch in C oder C++ möglich 	<ul style="list-style-type: none"> • Modulare Erweiterung vorgesehen, aber Standard nicht Final 	<ul style="list-style-type: none"> • Schnittstellen zu C und JAVA definiert • Plug-In-Konzept für die Anbindung von Testsystemen und Testtools (TTCN-3: Adapter und Codecs)
Debugging	<ul style="list-style-type: none"> • Abhängig von Entwicklungsumgebung möglich 	<ul style="list-style-type: none"> • Abhängig von Ausführungsumgebung möglich 	<ul style="list-style-type: none"> • Abhängig von Ausführungsumgebung möglich

Mit den dargestellten Eigenschaften lässt sich aufzeigen, dass die drei Testbeschreibungsmethoden unterschiedliche Stärken und Schwächen für die Anwendung in der Middleware haben. Um eine sinnvolle Entscheidung über den Einsatz einer der drei Testbeschreibungsmethoden zu treffen, ist die Betrachtung der gestellten Anforderungen notwendig. In der Tabelle 4.21 sind die Eignungen bezogen auf die wichtigsten Anforderungen für die Testbeschreibung mit einer relativen Bewertung – von -- bis ++ – gegenübergestellt.

Tabelle 4.21.: Anforderungserfüllung der Testbeschreibungsmethoden – Python, ISO 13209 und TTCN-3

Anforderungsnummer	Python	ISO 13209	TTCN-3
Anforderung 1	++	+	++
Anforderung 2	++	o	++
Anforderung 3	++	o	++
Anforderung 4	+	o	++
Anforderung 5	o	o	+
Anforderung 6	--	+	++
Anforderung 7	--	+	++
Anforderung 8	--	o	++
Anforderung 9	++	o	+
Anforderung 10	++	o	+
Anforderung 38	--	--	+
Anforderung 42	--	--	--
Anforderung 43	--	--	--

4.2.4. Lösungskonzept

Für das Lösungskonzept stehen die beiden in Kapitel 4.2.3 dargestellten Möglichkeiten zur Auswahl. Für die Umsetzung der Middleware kommt die zweite Lösungsvariante *Anwendung einer bestehenden Testbeschreibung* vorrangig in Betracht, da die Verwendung eines bestehenden Standards ein geringeres Entwicklungsrisiko bedeutet.

Innerhalb dieser Lösungsvariante stehen die drei Testbeschreibungsmethoden Python, ISO 13209 und TTCN-3 als Kandidaten zur Auswahl. Auf Basis der Betrachtungen in Abschnitt 4.2.3.2 und speziell der Gegenüberstellung in Tabelle 4.21 ist die Verwendung von TTCN-3 als Basis für die Testbeschreibung innerhalb der Middleware ausgewählt worden.

Für die Wahl von TTCN-3 spricht als erstes die gleichzeitige Verwendbarkeit von grafischer und programmatischer Entwicklung. Die parallele Verwendung beider Methoden ist schon aus dem Standard heraus vorgesehen. Ebenfalls ist der Einsatz unterschiedlicher grafischer Beschreibungsmethoden – neben dem standardisierten GFT – zu nennen, wie dies unter anderem im Beitrag [94] dargestellt ist.

Des Weiteren haben Arbeiten im Rahmen des TEMEA-Projektes aufgezeigt, dass die Verwendung von TTCN-3 im Bereich des Tests von Steuergeräten im Automotive-Bereich einsetzbar ist. Beispielfhaft sei hier die Arbeit [95] zu nennen.

Wie aus Tabelle 4.21 ersichtlich ist, werden nicht alle Anforderungen für die Middleware von TTCN-3 abgedeckt. Hier sind vorrangig die Anforderungen 42 und 43 für die Anbin-

derung der Requirementsmanagement zu nennen. Zur Realisierung dieser Anforderungen ist es, im Rahmen dieser Arbeit, notwendig die Schnittstellen und Standard-Funktionen von TTCN-3 zu erweitern. Die notwendigen Anpassungen und die Umsetzung sind in Kapitel 5 erläutert.

4.3. Testausführung

4.3.1. Zielsetzung

Die Zielsetzung ist die Konzeptionierung eines Test-Ausführungs-Moduls, entsprechend des in Kapitel 4.1.3 und in Kapitel 4.1.4 dargestellten Aufbaues.

4.3.2. Anforderungen

Die Anforderungen wurden in Kapitel 4.1.2 dargelegt und diskutiert. An dieser Stelle soll auf die, für die Testausführung, relevanten Anforderungen verwiesen werden. Die hier zu berücksichtigenden Anforderungen sind in den Tabellen 4.5, 4.6, 4.7, 4.8 und 4.9 aufgelistet. Ebenso sind die Anforderung 37 aus Tabelle 4.15 sowie die Anforderungen 46 und 47 aus Tabelle 4.18 zu berücksichtigen.

4.3.3. Lösungsvarianten

Für die Testausführung stehen eine Vielzahl verschiedener Lösungsvarianten zur Wahl. Mit der Entscheidung für TTCN-3 als Testbeschreibung ist schon eine Vorentscheidung für die Testausführung getroffen worden. Der TTCN-3 Standard beschreibt, wie in Kapitel 3.2.1.3 dargelegt, ebenfalls die Testausführung sowie die Schnittstellen zum Testsystem und dem DUT. Für diese Arbeit erschien es damit sinnvoll die Testausführung auf Basis des TTCN-3 Standards zu realisieren.

Da der TTCN-3 Standard die genaue Art der Aufführung – Interpreter oder Compiler – offen lässt, stehen mit der Verwendung von TTCN-3 weiterhin verschiedene Lösungsvarianten zur Diskussion, wie diese in den folgenden Abschnitten dargestellt wurden.

4.3.3.1. TTCN-3 Code Konverter

Zur Ausführung der TTCN-3 Testfälle besteht die Möglichkeit, den TTCN-3 Code in eine andere Sprache, wie zum Beispiel C oder C++, zu konvertieren, diesen Code mittels eines verfügbaren Compilers zu compilieren und das Ergebnis auszuführen. Diese Art der Lösung kommt bei verschiedenen TTCN-3 Implementierung, wie zum Beispiel Loong-Testing (siehe Kapitel 3.2.4.2) oder PicoTTCN (siehe Kapitel 3.2.4.3), zum Einsatz.

Aus Sicht der Ausführungsgeschwindigkeit erscheint eine Lösung mittels Code Konverter und Compiler vielversprechend, da ein direkt für die Zielplattform kompiliertes Programm eine hohe Ausführungsgeschwindigkeit zeigt. Auf der anderen Seite ist es bei jeder Änderung des TTCN-3 Codes notwendig, den Zielcode neu zu generieren und zu compilieren. Dies ist vor allem bei der iterativen Entwicklung von Tests zeitaufwändig und störend.

Außerdem führte die Konvertierung des TTCN-3 Codes und das anschließende Compilieren zu Problemen bei der Fehlersuche. Bei dem Auftreten von Fehlern zur Laufzeit ist die Zuordnung zwischen generiertem Code und TTCN-3 Code nur mit großem Aufwand möglich. Die Realisierung von Debug-Möglichkeiten, wie sie in Anforderung 9 und 10 gefordert sind, ist damit nur mit sehr hohem Aufwand möglich.

Des Weiteren zeigt der breite Einsatz von Python in der Testautomatisierung, dass die Verwendung eines Interpreters und die damit erzielbare Ausführungsgeschwindigkeit für den angestrebten Einsatzzweck ausreichend ist.

4.3.3.2. TTCN-3 Interpreter

Zur Ausführung von TTCN-3 Testfällen besteht alternativ die Möglichkeit den TTCN-3 Code direkt zu interpretieren und auszuführen. Hierfür ist es notwendig, einen TTCN-3 Interpreter zu realisieren. Diese Art der Lösung kommt bei verfügbaren TTCN-3 Systemen nur selten zum Einsatz. Einzig das BroadBit Test Tool (siehe Kapitel 3.2.4.3) verwendet diesen Ansatz.

Es zeigte sich, dass im Bezug auf die Zielsetzung dieser Arbeit die Realisierung eines Interpreters einige Vorteile bietet. Als erste sei hier die direkte Ausführung des TTCN-3 Codes zu nennen. Dies ermöglicht eine zeitsparende Entwicklung der Testfälle gegenüber des Code Konverters. Als zweites ermöglicht ein Interpreter und die damit verbundene direkte Ausführung des TTCN-3 Codes die direkte Umsetzung von Debug-Möglichkeiten, wie sie in Anforderung 9 und 10 gefordert sind. Als drittes ermöglicht ein Interpreter ein einfacheres und umfangreiches Fehlerhandling, da zur Ausführungszeit mehr Informationen über den aktuellen Zustand des Programmes und des Testsystemes vorliegen. Des Weiteren können die Fehlerbehandlungsmechanismen im Interpreter erweitert werden und sind anwendbar, ohne den Code der Testfälle anpassen zu müssen oder die Tests neu kompilieren zu müssen.

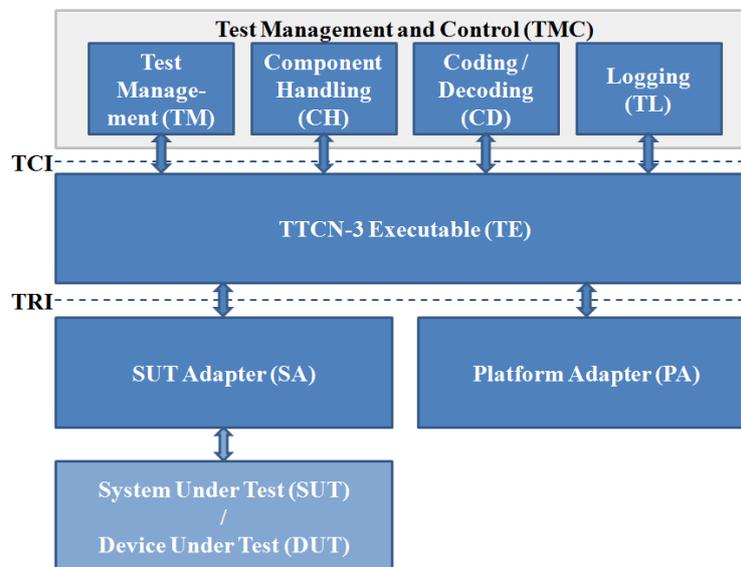
Als Nachteil ist die Ausführungsgeschwindigkeit der eigentlichen Tests zu sehen, welche mit einem klassischen Interpreter langsamer sind, als bei der Ausführung von direkt für die Zielplattform kompilierten Codes [96]. Dies ist dadurch bedingt, dass der Interpreter zur Laufzeit den Quelltext des Tests einliest, analysiert und danach ausführt. Ein Compiler überführt den Quelltext des Tests in Maschinen-Code, welche bei der Testdurchführung direkt ausgeführt wird.

4.3.4. Lösungskonzept

Für das Lösungskonzept stehen die beiden in Kapitel 4.3.3 dargestellten Möglichkeiten zur Auswahl. Für die Umsetzung der Middleware wurde die zweite Lösungsvariante *TTCN-3 Interpreter* ausgewählt, da diese in Bezug auf die gestellten Anforderungen besser anwendbar ist.

Der TTCN-3 Standard in Form der Dokumente [56], [57], [59] und [58] bildet die Basis für die Realisierung eines TTCN-3 Interpreters. In Abbildung 4.3 ist der daraus erarbeitete Aufbau des TTCN-3 Test-Ausführungs-Modul zu sehen.

Abbildung 4.3.: Struktur des Testausführungsmoduls auf TTCN-3-Basis



Dabei beschreiben die Dokumente [56] und [57] die TTCN-3 Sprache und ihre Ausführungssemantik. Diese Punkte werden durch den Interpreter umgesetzt und bilden das *TTCN-3 Executable (TE)*. Die Schnittstelle zum Testsystem (PA) und zum DUT (SA) werden über das TTCN-3 Runtime Interface (TRI) in Dokument [58] beschrieben. Dieses Interface ist ebenfalls durch das Test-Ausführungs-Modul umzusetzen. Die Schnittstelle zu den weiteren Komponenten, wie dem Test Management (TM) oder dem Logging (TL) für die Testdokumentation, bildet das TTCN-3 Control Interface (TCI), welches in dem Dokument [59] beschrieben ist. Dieses Interface ist ebenfalls durch das Test-Ausführungs-Modul umzusetzen. Die konkrete Umsetzung sowie die Implementierungsdetails sind in Kapitel 5 beschrieben.

4.4. Testdokumentation

4.4.1. Zielsetzung

Die Zielsetzung ist die Konzeptionierung eines Test-Report-Generator-Moduls entsprechend des in Kapitel 4.1.3 und in Kapitel 4.1.4 dargestellten Aufbaues.

4.4.2. Anforderungen

Die ermittelten Anforderungen wurden in Kapitel 4.1.2 dargelegt und diskutiert. An dieser Stelle soll auf die, für das Test-Reporting relevanten Anforderungen verwiesen werden. Die hier zu berücksichtigenden Anforderungen sind in den Tabellen 4.10, 4.11, 4.12, 4.13 und 4.14 aufgelistet. Des Weiteren sind die Anforderung 39 aus Tabelle 4.16 sowie die Anforderungen 44 und 45 aus Tabelle 4.17 zu berücksichtigen.

4.4.3. Lösungsvarianten

Eine erste Lösungsvariante ist die Verwendung eines vorhandenen Standards für die Speicherung der Testdokumentation. Wie in Kapitel 3.3.3 bereits dargestellt wurde, sind keine Standards bekannt, welche zur Realisierung der gestellten Anforderungen in dieser Arbeit verwendet werden können.

Damit besteht die Notwendigkeit ein eigenes Format für die Dokumentation der Tests zu erarbeiten. Hierbei stehen zwei mögliche Ansätze zur Auswahl.

Der erste Ansatz ist die Verwendung einer Datenbank zur Speicherung der Ergebnisse. Dieser Ansatz, wie ihn zum Beispiel EXAM – siehe Kapitel 3.2.2.2 – einsetzt, hat vor allem für die Generierung von Metriken, wie sie in Anforderung 35 und 36 gefordert sind, Vorteile, da die Daten für die Metriken über Datenbankabfragen schnell generierbar sind. Ein großer Nachteil dieses Ansatzes ist die Integration in das Versionsmanagement, welches eine zentrale Anforderung darstellt. Ebenfalls wurde diese dateibasierte Speicherung in Kapitel 4.1.3 für die Konzeptionierung und Umsetzung der Middleware festgelegt.

Daraus ergibt sich, dass ein dateibasiertes Format für die Speicherung der Testergebnisse zu entwickeln ist. Um eine möglichst reibungslose Integration in des Versionsmanagement zu ermöglichen, ist ein textbasiertes Format und kein binäres Format zu verwenden. Um eine gute Strukturierung der Daten und eine Erweiterbarkeit des Formates sicherstellen zu können, wurde die Realisierung eines Formates auf Basis von XML als bestmögliche Lösungsvariante erachtet. Außerdem ermöglicht die Verwendung von XML die direkte Erfüllung der Anforderungen 31 und 32 für die Bereitstellung eines Formates welches maschinell lesbar ist und manuell erweitert werden kann. Des Weiteren kann XML auch als Menschen lesbar betrachtet werden, sodass auch die Anforderung 33 als damit erfüllt betrachtet werden kann.

4.4.4. Lösungskonzept

Basierend auf den Betrachtungen des vorangegangenen Kapitels und den in Kapitel 4.4.2 zusammengestellten Anforderungen wurde ein XML-Schema für die Speicherung der Testergebnisse erarbeitet. In den Abbildungen 4.4 und 4.5 ist das XML-Schema grafisch dargestellt. Wie den Abbildungen zu entnehmen ist, bietet das Format die Speicherung

Abbildung 4.4.: XML-Schema für Testdokumentation Teil 1

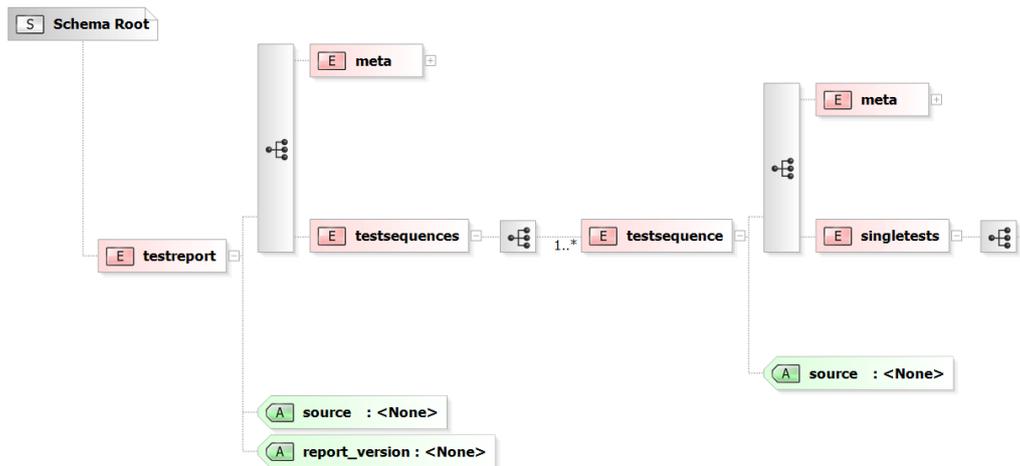
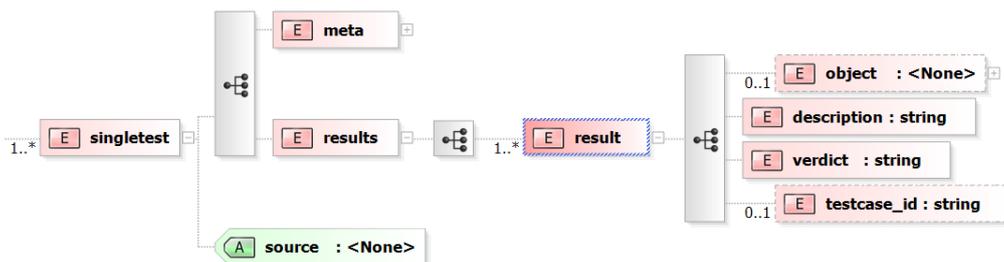


Abbildung 4.5.: XML-Schema für Testdokumentation Teil 2



der Testergebnisse in mehreren Gliederungsebenen (*testreport*, *testsequence* und *singletest*). Auf jeder Ebene besteht die Möglichkeit Meta-Daten, wie zum Beispiel Version, Ersteller, Änderungsdatum und Laufzeit) zuzuordnen. Das eigentliche Testergebnis wird im *result* gespeichert. Dies enthält neben der Beschreibung (*description*) und Bewertung (*verdict*) auch eine oder mehrere Anforderungs-IDs (*testcase_id*), für die Verknüpfung des Tests mit den Anforderungen in Requirementsmanagement-System sowie eine oder

mehrere Datenelemente (*object*) zum direkten Einbetten von Bildern oder Messdaten. Die konkrete Umsetzung sowie die Implementierungsdetails sind in Kapitel 5 beschrieben.

4.5. Integration in den Entwicklungsprozess

4.5.1. Zielsetzung

Die Integration in den Entwicklungsprozess ist keine Aufgabe, die einem einzelnen Modul der Middleware zuzuordnen ist. Die Integration ist dabei durch die Anbindung an das Versionsmanagement und das Requirementsmanagement gekennzeichnet. Dies soll hier nochmals getrennt betrachtet werden.

4.5.2. Anforderungen

Die hier zu betrachtenden Anforderungen sind, Anforderung 40 in Tabelle 4.16 sowie die Anforderungen 42 bis 45 in Tabelle 4.17. In den vorangegangenen Kapiteln wurden diese Anforderungen schon den einzelnen Modulen der Middleware zugeordnet. Sie sollen hier nochmals getrennt genauer betrachtet werden.

4.5.3. Lösungskonzept

Das Lösungskonzept ist hier in zwei Abschnitte zu trennen. Es werden die Anforderungen an die Anbindung des Versionsmanagement und des Requirementsmanagement jeweils getrennt betrachtet.

4.5.3.1. Versionsmanagement

Wie in Kapitel 4.1 schon dargelegt, ist die Einbindung der Tests in das Versionsmanagement für die Wiederholbarkeit von Tests für verschiedene Softwareversionen notwendig. Dabei soll es möglich sein, das selbe Versionsmanagement wie die Softwareentwicklung verwenden zu können. Da Versionsmanagement-Systeme für die Verwaltung von Quelltextdaten in Form von Textdateien entwickelt wurden, ist auch der Einsatz von Textdateien im Rahmen dieser Arbeit sinnvoll.

Es werden daher auf Textdateien basierende Formate für die Speicherung und den Austausch von Informationen zwischen den Modulen der Middleware verwendet. Die Speicherung der Testfälle erfolgt in Form von TTCN-3 Quelltext in Textdateien. Diese Dateien dienen als Schnittstelle zwischen dem Test-Beschreibungs-Modul und dem Test-Ausführungs-Modul. Im Übrigen beinhalten diese Dateien auch die Ausführungsreihen-

folge der Testfälle. Mit der Ablage dieser Dateien in einem Versionsmanagement-System ist die Wiederholbarkeit der Testfälle sichergestellt.

Die Speicherung der Testergebnisse erfolgt in einem definierten XML-Format, welches ebenfalls Textdateien verwendet. Dieses Format dient als Schnittstelle zwischen dem Test-Ausführungs-Modul und dem Test-Report-Generator-Modul. Es beinhaltet alle Informationen der Testausführung inklusive Bildern und Messdaten sowie die Verknüpfung zum Requirementsmanagement. Mit Versionierung dieser Datei im Versionsmanagement-System ist es jederzeit möglich den Testablauf nachzuvollziehen und verschiedenste Test-Reports und Metriken zu generieren.

Mit der Umsetzung des beschriebenen Konzeptes ist eine transparente Integration in Versionsmanagement-Systeme möglich. Ebenso können Funktionen, wie der Vergleich verschiedener Versionen der Testfälle bzw. der Test-Reports direkt aus dem Versionsmanagement-System verwendet werden. Außerdem ist die durchgehende Nachverfolgbarkeit sichergestellt.

4.5.3.2. Requirementsmanagement

Wie in Kapitel 4.1 dargestellt, ist die Einbindung der Tests in das Requirementsmanagement für die automatisierte Nachverfolgbarkeit von den Anforderungen bis zu den Testergebnissen notwendig.

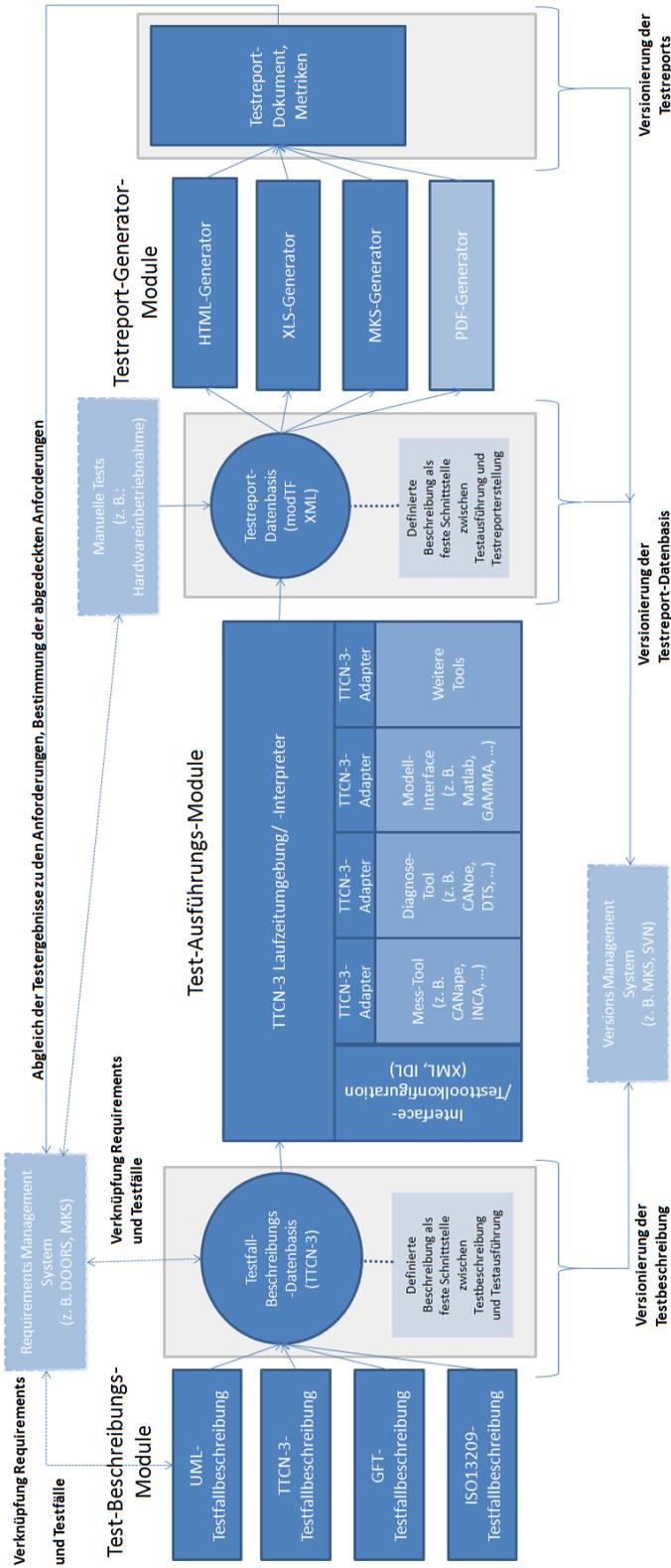
Für die Konzeptionierung dieser Anbindung wurden im Rahmen der Arbeit [97] die Requirementsmanagement-Systeme MKS Integrity, IBM Rational DOORS und HP Quality Center auf ihre Schnittstellen untersucht. Es konnte dabei gezeigt werden, dass sich ein allgemeingültiges Interface für den Austausch der Beschreibungen und Bewertungen der Anforderungen mit dem Test realisieren lassen. Die Schnittstelle besteht aus den vier Funktionen *Connect()*, *Disconnect()*, *GetDescription()* und *SetVerdict()*. Die Funktionen *Connect()* und *Disconnect()* dienen dem Verbindungsaufbau und -abbau mit den jeweiligen Requirementsmanagement-Systeme. Die Funktion *GetDescription()* liefert die Beschreibung zu einer Anforderung auf Basis der zugehörigen ID. Die Funktion *SetVerdict()* ermöglicht das Ablegen der Bewertung zu einer Anforderung auf Basis der zugehörigen ID. Es wurden für die genannten Systeme beispielhafte Umsetzungen erarbeitet. Diese Umsetzungen sowie die erarbeiteten Erkenntnisse kommen für die Middleware zur Anwendung.

Im Rahmen der genannten Arbeit konnte auch gezeigt werden, dass einzig die ID der jeweiligen Anforderungen notwendig ist, um die geforderte Nachverfolgbarkeit über alle Teile des Entwicklungs- und Testprozesses sicher zustellen. Bei der Konzeptionierung und Umsetzung der Middleware wird dies, durch die durchgehende Speicherung und Weitergabe der Anforderungs-ID umgesetzt. In der ersten Umsetzung wird die Anforderungs-ID dabei noch manuell in die Testfälle übertragen.

4.6. Gesamtkonzept

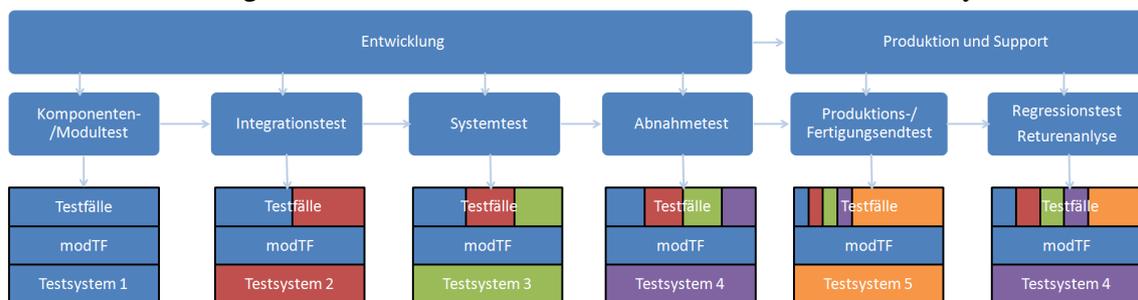
Aus den in den vorangegangenen Abschnitten betrachteten Konzepten für die einzelnen Module ergibt sich eine detailliertere Darstellung des Gesamtsystems, basierend auf der Abbildung 4.2 in Kapitel 4.1.4. Die erweiterte Darstellung ist in Abbildung 4.6 zu sehen. Anhand dieser Darstellung wird das Zusammenspiel der einzelnen Module in diesem Abschnitt erläutert.

Abbildung 4.6.: Struktur der modularen Middleware



Der Ausgangspunkt für die Tests ist die Testfallerstellung, welche mittels des Test-Beschreibungs-Moduls erfolgt. Hierbei wird die Verknüpfung der Testfälle mit den Anforderungen über die Anforderungs-ID realisiert. Die Anforderungs-ID wird als Teil der Testfälle in TTCN-3 gespeichert. Durch das GFT-Format von TTCN-3 ist eine parallele Bearbeitung der Testfälle in der grafischen Darstellung und in der textuellen Darstellung als Quelltext möglich. Weiterhin ist die Möglichkeit vorgesehen, auch weitere grafische Darstellungen, wie zum Beispiel UML oder ISO 13209, auf diesem Wege zu unterstützen. Alle Eingaben werden als TTCN-3 Dateien gespeichert. Diese bilden die Schnittstelle zum Test-Ausführungs-Modul und werden im Versionsmanagement-System versioniert. Die Ausführung der TTCN-3 Testfälle erfolgt durch die TTCN-3 Laufzeitumgebung bzw. den TTCN-3 Interpreter, welcher die Basis für das Test-Ausführungs-Modul bildet. Über die TRI-Schnittstelle von TTCN-3 erfolgt die Anbindung an das Testsystem und das DUT. Dies erfolgt über Platform Adapter (PA) und SUT Adapter (SA). Über diese Schnittstellen ist die Anbindung von Testtools, wie zum Beispiel Vector CANape, Vector CANoe, Softing DTS oder das Umgebungsmodell, möglich. Über die TCI-Schnittstelle wird dabei die Testausführung gesteuert und die Testergebnisse über das Logging-Interface gespeichert. Die Speicherung der Testergebnisse erfolgt dabei in Form der erarbeiteten XML Datei. Diese Datei enthält alle Informationen der Testausführung inklusive der Anforderungs-ID. Die Datei stellt damit die Schnittstelle zum Test-Report-Generator-Modul zur Verfügung. Es dient auch der Integration manueller Testergebnisse und dient gleichzeitig zur Ablage im Versionsmanagement-System.

Abbildung 4.7.: Einsatz der Middleware über den Produktlebenszyklus



Das Test-Report-Generator-Modul ermöglicht die Erstellung verschiedener Test-Reports auf Basis der XML Datei. Es ist dabei vorgesehen menschenlesbare Test-Reports in Formaten, wie zum Beispiel HTML oder XLS, zu erzeugen. Weiterhin realisiert dieses Modul auch den automatisierten Abgleich der Testergebnisse mit dem Requirementsmanagement-System auf Basis der Anforderungs-ID. Im Übrigen ermöglicht es dieses Modul verschiedene Metriken auf Basis der Testergebnisse zu erzeugen. Diese Metriken und Testberichte sind ebenfalls für die Ablage in das Versionsmanagement-System vorgesehen.

Das hier erarbeitete und dargestellt Gesamtkonzept ermöglicht die Erstellung und Ausführung von Tests mit der durchgehenden Anbindung an das Versionsmanagement- und

Requirementsmanagement-System. Dabei ist der Einsatz auf den verschiedenen Teststufen, vom Modultest bis zum Fertigungsendtest, möglich. Durch die Abstraktion zwischen Testbeschreibung und Testsystem mit dem Ansatz der Middleware, ist eine weitreichende Wiederverwendbarkeit von Testfällen realisierbar. In Abbildung 4.7 ist die Wiederverwendbarkeit der Testfälle sowie der mögliche Einsatz verschiedener Testsysteme über den Produktlebenszyklus, wie er in Kapitel 2.2.2 beschrieben wurde, dargestellt. Durch die unterschiedlichen Farben ist einerseits herausgestellt, dass in den verschiedenen Phasen des Produktlebenszyklus (z. B. Modultest, Integrationstest, ...) verschiedene Testsysteme zum Einsatz kommen. Andererseits ist dargestellt, dass durch die Verwendung der Middleware die Testfälle über die verschiedenen Phasen weiter verwendet werden können. Dies muss aber nicht bedeuten, dass jeweils alle Testfälle wiederverwendet werden. Am Beispiel des Produktionstests und des Regressionstests ist zu erkennen, dass die farblich gekennzeichneten Felder der Testfälle sich von der Größe deutlich unterscheiden. Dies soll verdeutlichen, dass die Auswahl der Testfälle beim Produktionstest eine andere sein kann wie beim Regressionstest. Der Ansatz der Middleware erlaubt eine hohe Flexibilität im Einsatz der Testfälle und Testsysteme.

5. Realisierung der Middleware

Es wird hier die Umsetzung des in Kapitel 4 erarbeiteten Konzeptes der modularen Middleware in Form des Testautomatisierungs-Frameworks modTF in ein real einsetzbares System dargestellt. Dabei wird in diesem Kapitel zuerst das Konzept in eine System- und Softwarearchitektur überführt. Außerdem wird ein Beispiel-Testfall eingeführt, an welchem die Eigenschaften des einzelnen Module des modTF dargestellt werden. Anschließend wird die Implementierung dargestellt.

5.1. Systemdesign

Die Systemdesign des modTF leitet sich direkt aus dem in Kapitel 4.6 dargestellten Gesamtkonzept ab. Die im Konzept erarbeiteten Module und deren Lösungskonzepte werden hier in ein erstes Design für die Implementierung umgesetzt [98, 99].

Die Testbeschreibung in Form des Test-Beschreibungs-Moduls ist als Frontend und Schnittstelle zum Testentwickler zu betrachten. Aus den in Kapitel 4.2.2 zugeordneten Anforderungen ergibt sich, dass dieses Modul als integrierte Entwicklungsumgebung (IDE) für TTCN-3 zu entwickeln ist. Es muss die Entwicklung und Speicherung der TTCN-3 Testfälle ermöglichen. Weiterhin muss es als Frontend die Realisierung der Debug-Funktionen unterstützen. Die konkrete Umsetzung dieses Moduls ist in Kapitel 5.3 dargestellt.

Die Testausführung in Form des Test-Ausführungs-Moduls ist als TTCN-3 Interpreter zu realisieren. Dieser Interpreter muss TTCN-3 Dateien entgegennehmen und diese ausführen. Ein grafisches Frontend ist dafür nicht notwendig. Es sind weiterhin Implementierungen für die Schnittstellen TRI und TCI zu realisieren. Dabei ist zu beachten, dass die Anbindung von Testsystem und Testtools über die TRI-Schnittstelle jeder Zeit modular erweiterbar sein muss, ohne das der Interpreter angepasst werden soll. Außerdem muss der Interpreter Funktionen zum Debugging der TTCN-3 Tests zur Verfügung stellen. Die Ergebnisse der Testausführung müssen über die Logging-Schnittstelle von TTCN-3 in modTF XML Dateien gespeichert werden. Die konkrete Umsetzung dieses Moduls ist in Kapitel 5.4 dargestellt.

Das Test-Reporting in Form des Test-Report-Generator-Moduls ist als Bibliothek von Funktionen zum Lesen und Auswerten des modTF XML, sowie darauf aufsetzende Generatoren für die verschiedenen Formate für Test-Reports zu realisieren. Ein grafisches Frontend ist dafür nicht notwendig. Die zu realisierende Bibliothek muss, neben der strukturierten Ausgabe der Testergebnisse und Meta-Daten, ebenfalls Funktionen für die

statistische Auswertung der Testergebnisse zur Verfügung stellen, da nur so einheitliche Statistiken über verschiedene Test-Report Formate sicher zu stellen sind. Die Generatoren für die unterschiedlichen Zielformate stellen dabei nur eine Konvertierung der durch die Bibliothek zur Verfügung gestellten Daten in das Zielformat bereit. Die konkrete Umsetzung dieses Moduls ist in Kapitel 5.5 dargestellt.

In den folgenden Abschnitten werden die einzelnen Module sowie deren Implementierung einzeln betrachtet. Neben der Implementierung werden auch die umgesetzten Anforderungen sowie die Anwendung der einzelnen Module beschrieben.

5.2. Beispiel-TestszENARIO

Zur Verdeutlichung der Eigenschaften und Leistungsfähigkeit der einzelnen Module des modTF sowie zur besseren Darstellung der Zusammenhänge, wird ein durchgehendes Beispiel verwendet. Hierfür dient der Test des Kommunikations-Stacks der AUTOSAR Basissoftware für FlexRay nach AUTOSAR Version 4.0. Für das Verständnis des Testszenarios, soll an dieser Stelle eine kurze Einführung in den Aufbau der AUTOSAR Software sowie in die FlexRay-Kommunikation gegeben werden.

AUTOSAR (AUTomotive Open System ARchitecture) [16] ist ein Standard für eine Software-Plattform für Automotive Steuergeräte, welcher einerseits die Module der Software eines solchen Automotive Steuergerätes und deren Schnittstellen beschreibt. Andererseits ist auch der Entwicklungsprozess mittels standardisierter Konfigurationsdateien im AUTOSAR-XML-Format (ArXML) beschreiben. In Abbildung 5.1 ist eine Übersicht der Schichten der AUTOSAR Software zu sehen [100]. Diese Unterteilt sich in:

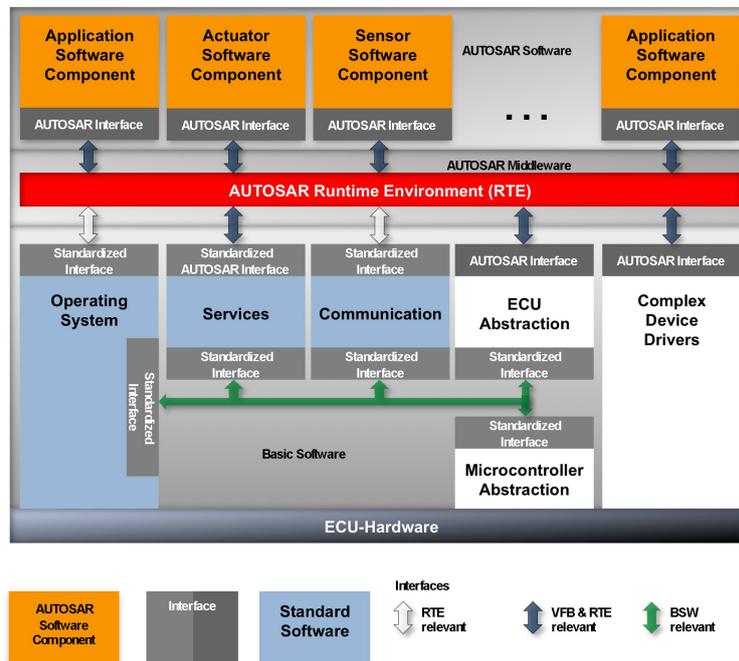
- Basis Software (engl. Basic Software – BSW)
- AUTOSAR Runtime Environment (RTE)
- AUTOSAR Software

Die Basis Software besteht aus dem Betriebssystem (engl. Operating System) und verschiedenen Basis-Software-Komponenten. Diese unterteilen sich in den *Microcontroller Abstraction Layer*, welcher die Basis-Software-Komponenten für die Ansteuerung der Hardware enthält. Oberhalb dieser Schicht ist der *ECU Abstraction Layer* zu finden, welcher die Verbindung zwischen den Hardware-abhängigen Komponenten des *Microcontroller Abstraction Layer* und dem *Service Layers* herstellt. Der *Service Layer* beinhaltet alle Hardware-unabhängigen Basis-Software-Komponenten, wie zum Beispiel Kommunikations-Handler, Fehlerspeicher oder Diagnose Kommunikation.

Das AUTOSAR Runtime Environment (RTE) dient der Abstraktion der Steuergerätespezifischen Eigenschaften. Für die AUTOSAR Software stellt die RTE als Middleware eine einheitliche Schnittstelle zur Verfügung. Ziel dieser Schicht ist die einfache Austauschbarkeit der Komponenten der AUTOSAR Software.

Die AUTOSAR Software selbst besteht aus einzelnen Komponenten, welche als *Software-*

Abbildung 5.1.: AUTOSAR Übersicht [100]



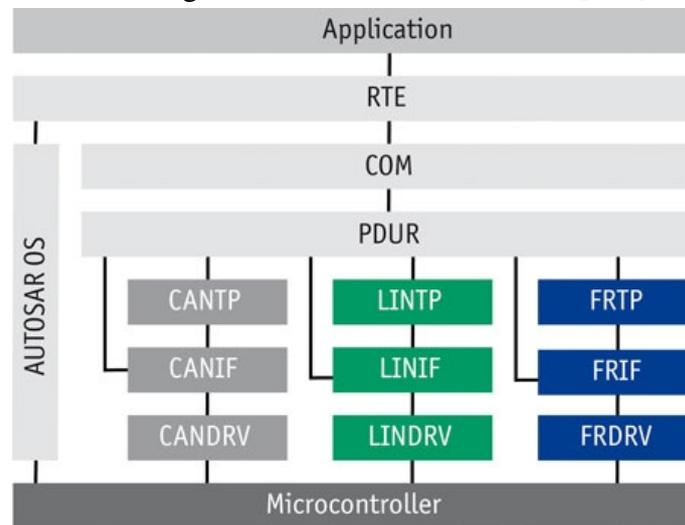
Komponente (SW-C) bezeichnet werden. Die SW-C stellt dabei eine abgeschlossene Funktion dar, welche nur über das Interface der RTE kommuniziert. Ein solche Komponente lässt sich daher leicht auf verschiedenen Steuergeräten integrieren.

Weiterführende Informationen sind dem AUTOSAR Standard unter [16] zu entnehmen.

FlexRay ist ein serielles, deterministisches Bussystem, welches ausschließlich im Automobilbereich eingesetzt wird. Es verwendet das TDMA (engl. Time Division Multiple Access) [3, S.447] Verfahren für die zeitliche Steuerung der Kommunikation. Es werden dabei Datenrahmen (engl. Frames [102, p. 49]), bestehend aus Header, bis zu 254 Byte Nutzdaten und CRC, verwendet. Die Datenrate beträgt 10 Mbit/s. Eine ausführliche Beschreibung der FlexRay-Kommunikation ist in [13] zu finden.

Im Folgenden wird der Kommunikations-Stack von AUTOSAR für die FlexRay-Kommunikation näher betrachtet. In Abbildung 5.2 ist die Verbindung der einzelnen BSWs dargestellt [101]. Beginnend bei der Hardware des Mikrocontrollers folgt als erstes der *FlexRay Driver* (FRDRV), welcher als Teil des *Microcontroller Abstraction Layers* die Ansteuerung des FlexRay-Controllers übernimmt. Es folgt die *FlexRay Interface* (FRIF) Komponente, welche als Teil des *ECU Abstraction Layers* eine einheitliche Schnittstelle zu den übergeordneten Schichten zur Verfügung stellt. Die nächste Komponente mit der Bezeichnung *FlexRay Transport Layer* (FRTP) ist optional. Diese dient der Segmentierung und des Zusammenfügens von *Protocol Data Unit* (PDU) [102, p. 63], wenn diese nicht in einen FlexRay-Datenrahmen passen. Sollte dieser Fall nicht unter-

Abbildung 5.2.: AUTOSAR Com Stack [101]



stützt werden, kann diese Komponente entfallen. Bis zu diesem Punkt sind alle Komponenten speziell für den FlexRay-Bus vorgesehen. Die nun folgenden Komponenten sind in der *Service Layer* anzusiedeln und unabhängig von einem speziellen Bussystem.

Die Komponente *PDU Router* (PDUR) dient dem statischen Routing der PDUs auf die verschiedenen Bussysteme. Das Routing wird zur Entwicklungszeit festgelegt und ist daher statisch. Es erfolgt keine dynamische Zuordnung zur Laufzeit des Systems. Die nun folgende *Communication*-Komponente (COM) dient als Schnittstelle zu der AUTOSAR RTE. Es stellt ein signalbasiertes Interface für die RTE zur Verfügung und übernimmt das Zusammenführen von Signalen in PDUs (Senderichtung) sowie das Auspacken von Signalen aus PDUs (Empfangsrichtung). Ein Signal wird bei AUTOSAR auch als *Data Element* [102, p. 31] bezeichnet und besteht aus einem Datenelement definierten Typs mit definierter Bedeutung der Werte des Datenelements sowie Gültigkeitsbereichen (valid und invalide Werte). Dabei übernimmt diese Komponente auch die Umrechnung zwischen der Darstellung auf der RTE (physikalischer Wert) und der Darstellung, welche über den Bus übertragen wird (Rohwert). Die letzte Komponente ist die RTE, welche die Signale den SW-Cs der AUTOSAR Software zur Verfügung stellt.

Die Spezifikation der Kommunikationsbeziehungen erfolgt formal in Form von Field Bus Exchange Format (FIBEX)-Dateien oder ArXML-Dateien.

Diese Beschreibung enthält für FlexRay die folgenden Punkte:

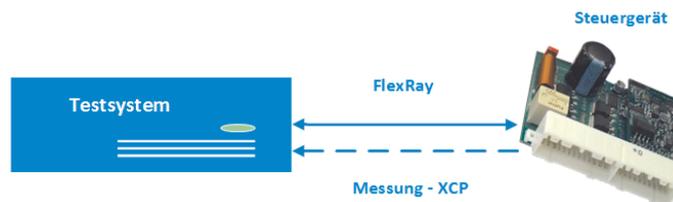
- Steuergeräte
- FlexRay-Frames mit der Zuordnung zu den Steuergeräten
- PDUs mit der Zuordnung zu FlexRay-Frames
- Signale mit der Zuordnung zu PDUs
- Umrechnungsvorschriften zwischen physikalischen Signalen (Abbildungsvorschrift an der RTE) und Rohwerten (Abbildungsvorschrift innerhalb der PDU)
- Zykluszeit (Periode im Sinne des TDMA-Verfahrens) für Frames und PDUs

Nach der Betrachtung der zu testenden Komponenten wird nun der eigentliche Testfall dargestellt. Es soll dabei beispielhaft das Signal einer PDU mit folgenden Eigenschaften betrachtet werden:

- Name: Fahrzeuggeschwindigkeit
- Datentyp des physikalischen Wertes: float (32Bit)
- Wertebereich des physikalischen Wertes: 0,0 ... 350,0
- Einheit des physikalischen Wertes: km/h
- Datentyp des Rohwertes: uint16
- Wertebereich des Rohwertes: 0 ... 35000
- Umrechnungsvorschriften: $v_{\text{phy}} = 0,01 * v_{\text{roh}}$
- Zykluszeit der PDU: 40ms

Das Signal wird durch das Testsystem gesendet und vom Steuergerät empfangen. Gegeben sei weiterhin das Testsystem, wie es in Abbildung 5.3 dargestellt ist. Die Ausführung der Tests erfolgt auf dem Testsystem, welches ebenfalls die FlexRay-Frames sendet und empfängt. Weiterhin besteht mittels Universal Measurement and Calibration Protocol / eXtended Calibration Protocol (XCP) ein lesender Zugang in den RAM-Bereich des zu testenden Steuergerätes. Es ist damit möglich in einem deterministischen Zeitintervall von 1 ms globale Variablen der Steuergeräte-Software zu lesen.

Abbildung 5.3.: AUTOSAR Com Stack –Testaufbau



Aus den vorangegangenen Ausführungen lassen sich nun drei Testfälle ableiten, welche in den folgenden Abschnitten zur Erläuterung der Eigenschaften des modTF verwendet werden.

1. Es ist der korrekte Empfang des Signals als Rohwert in der COM Komponente zu prüfen. Dabei ist der gesamte Wertebereich des Rohwertes (uint16 - [0,65535]) zu prüfen. Entsprechend der Definition des Signales ergeben sich zwei Äquivalenzklassen [3, S. 133]. Die gültige Äquivalenzklasse hat das Intervall [0,35000] und die ungültige Äquivalenzklasse hat das Intervall [35001,65535]. Unter Berücksichtigung der Grenzwertanalyse [3, S. 392] für die Äquivalenzklassen ergeben sich dabei die folgenden Signalwerte für den Test 0, 35000, 35001 und 65535. Es wird erwartet, dass der vom Testsystem gesendete Wert im Steuergerät gelesen werden kann, da die Bewertung des Gültigkeitsbereiches erst nach der Umrechnung in den physikalischen Wert innerhalb der COM Komponente statt findet.
2. Es ist sicherzustellen, dass die Zeit zwischen Empfang des Signales auf dem FlexRay und der Verfügbarkeit des Wertes auf der RTE kleiner als 10 ms ist. Dazu ist der exakte Sendezeitpunkt des FlexRay-Frames auf dem Testsystem zu bestimmen und zyklisch die RTE-Variable des Signals mittels XCP zu messen. Im Anschluss ist die Zeitdifferenz zu bestimmen.
3. Es ist die Umrechnung zwischen Rohwert und physikalischem Wert inklusive der Gültigkeitsprüfung des Wertebereiches zu prüfen. Aus der Definition des Signals im physikalischen Bereich ergeben sich 3 Äquivalenzklassen. Die gültige Äquivalenzklasse hat das Intervall [0,0, 350,0] wobei die begrenzte Genauigkeit der Zahlendarstellung für Gleitkommazahlen zu beachten ist. Die erste ungültige Äquivalenzklasse umfasst alle negativen Zahlen des Datentyps float mit dem Intervall [-3,4E+38, 0,0). Die zweite ungültige Äquivalenzklasse umfasst das Intervall (350,0, +3,4E+38]. Die erste ungültige Äquivalenzklasse lässt sich nicht sinnvoll auf den Wertebereich des Rohwertes abbilden. Die Verwendung dieser Werte sollte schon zu einer Fehlermeldung durch das Testsystem führen. Für die beiden anderen Äquivalenzklassen werden unter Berücksichtigung der Grenzwertanalyse die Signalwerte 0,0, 350,0, 350,1 und +3,4E+38 für den Test gewählt. Für die Werte der gültigen Äquivalenzklasse wird erwartet, dass der gesendete Werte mit einer Toleranz von $\pm 0,0005$ (halbe Schrittweite der Signalumrechnung) für Rundungsfehler bei der Umrechnung auf der RTE gelesen wird. Für die Werte der ungültigen Äquivalenzklasse wird erwartet, dass der letzte gültige Werte mit einer Toleranz von $\pm 0,0005$ sowie der Status mit dem Fehlerwert 2 (invalidier Wert) auf der RTE gelesen werden kann.

5.3. Testbeschreibung

Im Rahmen dieses Kapitels wird die Umsetzung des Test-Beschreibungs-Moduls im Detail betrachtet. Hierfür wurden im ersten Teil existierende Tools auf ihre Verwendbarkeit bzw. Adaptierbarkeit untersucht. Im weiteren Verlauf wird die konkrete Umsetzung für die dieses Projekt dargestellt.

5.3.1. Vorbetrachtungen

Im Rahmen der Vorbetrachtungen zur Realisierung des Test-Beschreibungs-Moduls, wurden Recherchen zu bereits existierenden Entwicklungsumgebungen für TTCN-3 Tests durchgeführt. Die verschiedenen kommerziellen Produkte, wie zum Beispiel OpenTTCN3 oder TTworkbench, beinhalten eine integrierte Entwicklungsumgebung. Da diese mit der eigentlichen Testausführung untrennbar verbunden sind, kommen diese Tools für dieses Projekt nicht in Frage.

Im Bereich der freien Entwicklungen ist das Tool LoongTesting zu nennen, welches eine integrierte Entwicklungsumgebung für TTCN-3 Testfälle mitbringt. Aber auch bei diesem Tool ist die Testerstellung und Testausführung untrennbar verbunden. Der Quelltext steht ebenfalls nicht frei zu Verfügung, so dass es auch nicht möglich ist die Entwicklungsumgebung entsprechend weiter zu entwickeln.

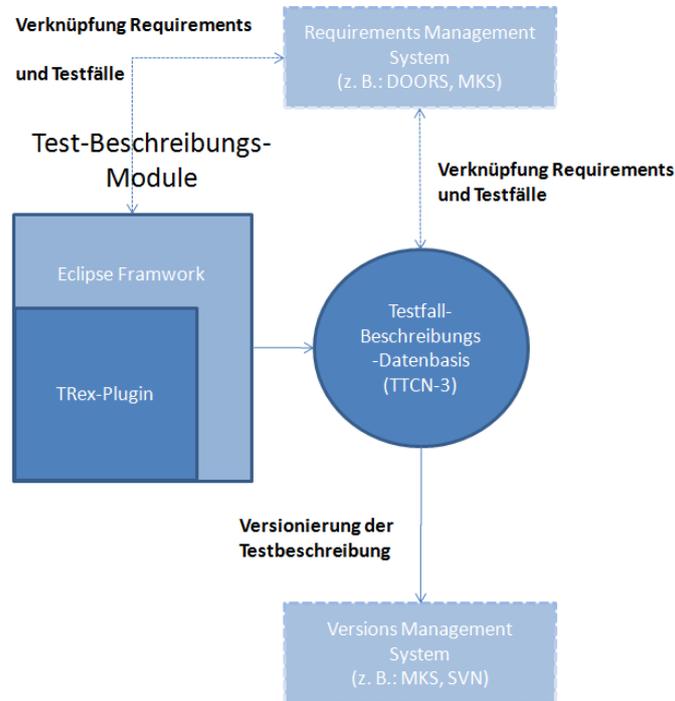
Für die Realisierung einer IDE ist der Eclipse Framework [103] der Eclipse Foundation häufig im Einsatz. Bei der Recherche zur möglichen Umsetzung des Test-Beschreibungs-Moduls als Plugin für Eclipse konnte ein bereits existierendes Plugin aufgefunden werden. Das Plugin TRex, siehe dazu auch Kapitel 3.2.4.1, stellt weitreichende Funktionen zur Entwicklung von TTCN-3 Testfällen zur Verfügung. Der Quelltext ist frei verfügbar und es findet eine stetige Weiterentwicklung statt. TRex wurde im Rahmen der Arbeit für die Erstellung von TTCN-3 Testfällen evaluiert. Die umfangreichen Funktionen zur Entwicklung, wie Syntax-Highlighting, Refactoring für TTCN-3 Code und die Erstellung von Metriken zum TTCN-3 Code, erlauben eine schnelle und effektive Entwicklung von TTCN-3 Testfällen.

5.3.2. Umsetzung des Lösungskonzeptes in die Softwarearchitektur

Auf Basis der Rechercheergebnisse zu existierenden Entwicklungsumgebungen für TTCN-3 Tests wurde der Einsatz von TRex als Grundlage für das Test-Beschreibungs-Modul des modTF ausgewählt. Die sich daraus ergebende Softwarearchitektur ist in Abbildung 5.4 zu sehen.

Es zeigte sich, dass die Entwicklung einer eigenen Entwicklungsumgebung bzw. eines eigenen Eclipse-Plugins, durch das Vorhandensein des TRex-Plugins, nicht sinnvoll er-

Abbildung 5.4.: Struktur des Test-Beschreibungs-Modul



scheint. Der zu erwartende zeitliche Aufwand steht in keinem Verhältnis zum zu erwartenden Nutzen.

Neben der Entwicklung von Testfällen in TTCN-3 ermöglicht TRex die Integration von TTCN-3 Compilern für die direkte Ausführung von Testfällen. Dies ist in [104] für OpenTTCN und Danet TTCN-3 dargestellt. Wie in [105] angegeben, ist die Integration eines eigenen Compilers bzw. Interpreters im Package *de.ugoe.cs.swe.trex.compiler* des TRex-Plugin durchzuführen.

Im Rahmen der Integration des Interpreters des modTF in TRex, ist ebenfalls die Unterstützung für das Debugging zu integrieren. Dabei soll auf die Debug-Unterstützung des Eclipse Frameworks zurückgegriffen werden [106]. TRex unterstützt bisher keine Programmierung unter Verwendung des GFT-Formates. Zu Unterstützung der grafischen Programmierung ist daher eine Erweiterung bzw. Ergänzung von TRex notwendig.

5.3.3. Implementierung der Softwarearchitektur

Es konnte gezeigt werden, dass durch die Verwendung des TRex-Plugins die Entwicklung von TTCN-3 Testfällen mit allen Funktionen einer IDE möglich ist. Zur Realisierung der in Abschnitt 5.3.2 beschriebenen Softwarearchitektur sowie zur Umsetzung der in Abschnitt 4.2.2 dargestellten Anforderungen ist nur ein geringer Implementierungsaufwand notwendig. Im Rahmen dieser Arbeit wurde jeweils die aktuelle Version des

TRex-Plugins verwendet. Dies waren jeweils die aktuellen Entwicklungsstände des Version 0.6.0.

Die Umsetzung der Anbindung an das Requirementsmanagement wird durch Anforderungs-IDs realisiert. Diese IDs lassen sich ohne Änderungen an der Entwicklungsumgebung in den Standard-Sprachumfang von TTCN-3, in Form von angepassten *log*-Aufrufen, realisieren. In Abbildung 5.5 ist die Ausgabe der Anforderungs-ID 12345 zu sehen. Diese ID wird nun für diesen Testfall – in TTCN-3 *testcase* – verwendet. Die Implementierung der weiteren Verarbeitung ist Teil des Test-Ausführungs-Moduls und wird in Kapitel 5.4 erläutert.

Abbildung 5.5.: TTCN-3 *log*-Aufrufen mit Anforderungs-ID

```
log("[[TC-ID]]12345[[_TC-ID]]");
```

Die angestrebte Integration des entwickelten TTCN-3 Interpreters in das TRex-Plugin, konnte im Rahmen dieser Arbeit nicht realisiert werden, da die Umsetzung des Interpreters sowie der Testdokumentation, wie sie in den Kapitel 5.4 und 5.5 beschrieben sind, für eine durchgehende Umsetzung des modTF eine höhere Priorität aufwiesen. Die Ausführung der TTCN-3 Testfälle ist durch Verwendung des PyDev-Plugin [107] und der dortigen Ausführung des Interpreters möglich. Der Interpreter ist so angelegt, dass das auszuführende Testskript als Kommandozeilen-Parameter übergeben werden kann oder ein Dialog zur Auswahl des TTCN-3 Testskript angezeigt wird, wie diese in Abschnitt 5.3.6 dargestellt ist.

Die vorgesehene Unterstützung des Debugging der TTCN-3 Testfälle ist im Rahmen dieser Arbeit nur teilweise realisiert worden. Unter Verwendung von PyDev kann das Debugging auf Ebene des Python-Codes des Interpreters durchgeführt werden. Durch ein weitreichendes Fehlerhandling des Interpreters mittels Exceptions im Python-Code, kann ein Fehler im TTCN-3 Testskript schnell aufgefunden werden. Durch die Realisierung der Symboltabelle des TTCN-3 Interpreters in Python können die im TTCN-3 Testskript verwendeten Variablen und deren Werte während des Debugging mittels PyDev eingesehen werden. Diese Funktionalität ersetzt nicht vollständig die direkte Debug-Unterstützung auf TTCN-3 Ebene. Die Unterstützung des GFT-Formates steht ebenfalls noch aus.

5.3.4. Anwendung auf Beispiel-Testfälle

Im Folgenden werden die in Abschnitt 5.2 beschriebenen Testfälle mittels TTCN-3 implementiert. Die aufgezeigten Implementierungen stellen eine gekürzte Form des TTCN-3-Codes dar. Aus Gründen der Übersichtlichkeit wurden die notwendigen Definitionen sowie Fehlerbehandlungen weggelassen. Da die Anforderungen formal in Form einer Fibex-Datei vorliegen, ist es möglich, die Testfälle automatisiert zu generieren. Dies soll an dieser Stelle nicht weiter betrachtet werden, da es um die Umsetzbarkeit der Testfälle für eine automatisierte Ausführung geht. Es wäre auch möglich die Testfälle manuell zu im-

plementieren.

Zur Implementierung von Testfall 1 ist es notwendig, den Wert für die Fahrzeuggeschwindigkeit als Rohwert auf dem FlexRay vom Testsystem zu senden und den Wert innerhalb des Steuergerätes mittels XCP zu lesen. Als Abbruchkriterium für den Test wird zusätzlich angenommen, dass maximal 2 Perioden zu je 40 ms des FlexRay-Frames vergehen dürfen, bis der erwartete Wert innerhalb des Steuergerätes gelesen werden kann. Dieses Abbruchkriterium kommt nur zum Tragen, wenn keine Daten per XCP vom Steuergerät gelesen werden können.

In Listing 5.1 ist die Implementierung des Testfalles beispielhaft für den Wert 0 dargestellt.

Listing 5.1: TTCN-3 Testskript für Testfall 1

```
1 timer    t_guard ;
2 log (" [[TC-ID]]1[[_TC-ID]]" );
3 flexrayPort_raw . send ( fahrzeuggeschwindigkeit ( 0 ) );
4 t_guard . start ( 2 * PERIODE );
5 alt {
6     [] xcpPort . receive ( fahrzeuggeschwindigkeit_COM_raw ( 0 ) ) {
7         setverdict ( pass ) }
8     [] xcpPort . receive ( ) { setverdict ( fail ) }
9     [] t_guard . timeout ( ) { setverdict ( fail ) }
}
```

In Zeile 3 wird die Fahrzeuggeschwindigkeit mit dem Wert 0 auf dem FlexRay gesendet. Die Ausgabe als Rohwert wird über die Verwendung des entsprechenden Ports (*flexray-Port_raw*) gesteuert. In Zeile 4 startet die Timeout-Überwachung mit einer Zeit von 80 ms. Die Genauigkeit hängt hierbei vom Testsystem ab. Dies wird in Abschnitt 5.4.3 näher erläutert. Die Zeilen 5 - 9 enthalten die Prüfung der Reaktion des Steuergerätes. Zeile 6 kommt zur Anwendung, wenn der erwartete Wert 0 per XCP gelesen werden konnte. Zeile 7 kommt zu Anwendung, wenn ein anderer Werte als 0 gelesen werden konnte. Und Zeile 8 wird ausgeführt, wenn kein Wert per XCP gelesen werden konnte und der Timeout aufgetreten ist.

Für die Implementierung des Testfalles 2 stehen die Erweiterungen *TTCN-3 Performance and Real Time Testing* [108] und *Support of interfaces with continuous signals* [109] zur Verfügung. Die Implementierung des Testfalles mittels *TTCN-3 Performance and Real Time Testing* ist in Listing 5.2 zu sehen.

Listing 5.2: TTCN-3 Testskript für Testfall 2 – Performance and Real Time Testing

```
1 log (" [[TC-ID]]2[[_TC-ID]] ");
2 flexrayPort_raw.send(fahrzeuggeschwindigkeit(0)) -> timestamp
   starttime;
3 t_guard.start(2 * PERIODE);
4 alt{
5     [] xcpPort.receive(fahrzeuggeschwindigkeit_COM_raw(0)) ->
       timestamp receivetime{
6         if ((receivetime - starttime) < 10*millisec){
7             setverdict(pass); }
8         else {
9             setverdict(fail);}
10    }
11    [] xcpPort.receive() {setverdict(fail)}
12    [] t_guard.timeout() {setverdict(fail)}
13 }
```

Im Unterschied zu Listing 5.1 ist es nun in den Zeilen 2 und 5 möglich, Zeitstempel mit dem Zeitpunkt des Sendens des FlexRay-Frames sowie mit dem Zeitpunkt des Empfanges des erwarteten Wertes über XCP, zu erhalten. Aus den Zeitstempeln lässt sich in Zeile 6 die Zeitdifferenz bestimmen. Die Genauigkeit der Zeitstempel ist vom Testsystem abhängig. Dies wird in Abschnitt 5.4.3 diskutiert. Weitergehende Anwendungsbeispiel für die Erweiterung *TTCN-3 Performance and Real Time Testing* ist in [110] zu finden.

Die Implementierung des Testfalles mittels *Support of interfaces with continuous signals* ist in Listing 5.3 zu sehen.

Listing 5.3: TTCN-3 Testskript für Testfall 2 – Support of interfaces with continuous signals

```
1 log (" [[TC-ID]]2[[_TC-ID]] ");
2 flexrayPort_raw.delta := 0.001;
3 xcpPort.delta := 0.001;
4 flexrayPort_raw.send(fahrzeuggeschwindigkeit(0));
5 t_guard.start(2 * PERIODE);
6 alt{
7     [] xcpPort.receive(fahrzeuggeschwindigkeit_COM_raw(0)){
8         xcpPortStream := xcpPort.history(0.0, now);
9         flexrayPortStream := flexrayPort_raw.history(0.0, now);
10        for(vat integer count := 0; count < lengthof(
11            xcpPortStream); count := count + 1){
12            if((flexrayPortStream[count].value != 0) and (
13                flexrayPortStream[count + 1].value == 0)){
14                sendtime := flexrayPortStream[count + 1].
15                    timestemp;}
16            if((xcpPortStream[count].value != 0) and (
17                xcpPortStream[count + 1].value == 0)){
18                receivetime := xcpPortStream[count + 1].
19                    timestemp;}
20        }
21        if ((receivetime - starttime) < 10*millisec){
22            setverdict(pass); }
23        else {
24            setverdict(fail);}
25    }
26    [] t_guard.timeout() {setverdict(fail)}
27 }
```

Der erste Unterschied zu den vorangegangenen Implementierungen ist in den Zeilen 2 und 3 zu sehen. Hier wird eine Schrittweite für die Aufzeichnung der Daten des Ports konfiguriert. Im Beispiel wird die Zeit auf 0,001 s konfiguriert. Wenn das Testsystem diese zeitliche Anforderung nicht erfüllen kann, so wird an dieser Stelle eine Exception erzeugt und es erfolgt damit ein Eintrag mit dem Verdict *error* in den Testreport. Die eigentliche Bewertung des Test erfolgt in den Zeilen 8 bis 19. Hier werden mittels des Schlüsselwortes *history* die Verläufe der Signale in Form von Listen (*Stream*) mit Paaren aus Wert (*value*) und Zeit (*timestemp*) gelesen. Aus diesen Listen werden nun die Änderungen der Werte bestimmt und die zugehörigen Zeiten ermittelt. Mittels der ermittelten Zeiten kann die Bewertung des Testfallens durchgeführt werden. Das Testsystem ist dafür verantwortlich, ein synchrone Zeitbasis zur Verfügung zu stellen. Diese Form der Implementierung benötige zwar die meisten Code-Zeilen, hat dafür den Vorteil, dass die Signalverläufe auch als grafische Darstellungen zur Visualisierung des Verhaltens des Steuergerätes verwendet werden können.

Die Implementierung des Testfalles 3 in Listing 5.4 ähnelt stark dem Testfall 1. Das Senden des Signals als physikalischer Wert erfolgt durch die Verwendung eines anderen Ports. Dieser Port realisiert über die Einbindung einer CD-Komponente von TTCN-3, wie sie

in Abbildung 4.3 dargestellt ist und in Abschnitt 5.4.1.4 erläutert wird, die Umrechnung in den Rohwert. Diese Komponente ist aus den Umrechnungsvorschriften aus der Fibex-Datei zu generieren.

Listing 5.4: TTCN-3 Testskript für Testfall 3

```

1 log (" [[TC-ID]]3[[_TC-ID]] ");
2 flexrayPort_phy . send ( fahrzeuggeschwindigkeit ( 0.0 ) );
3 t_guard . start ( 2 * PERIODE );
4 alt {
5     [] xcpPort . receive ( fahrzeuggeschwindigkeit_COM_phy () ) ->
6         value fahrzeuggeschwindigkeit {
7             if ( ( fahrzeuggeschwindigkeit > ( 0.0 - 0.0005 ) ) and (
8                 fahrzeuggeschwindigkeit < ( 0.0 + 0.0005 ) ) )
9                 setverdict ( pass ); }
10        else {
11            setverdict ( fail ); }
12    [] xcpPort . receive () { setverdict ( fail ) }
13    [] t_guard . timeout () { setverdict ( fail ) }
14 }
```

Mit der Codierung und Decodierung über die CD-Komponente stellt TTCN-3 eine sehr flexible Möglichkeit der Konvertierung von Daten zur Verfügung. Die Umrechnung kann dabei unabhängig vom speziellen Testfall angepasst werden. Damit ist eine einfache Übernahme auf ein anderes Testsystem möglich. Wie die vorangegangenen Testfälle zeigen, können je nach Testziel verschiedene Darstellungen der gleichen Schnittstelle des Testsystems verwendet werden.

5.3.5. Umgesetzte Anforderungen

In diesem Abschnitt werden die Anforderungen, welche in Abschnitt 4.2.2 der Testbeschreibung zugeordnet wurden, einzeln bezüglich ihrer Umsetzung zusammengefasst.

Anforderung 1 – Verwendung einer höheren Programmiersprache mit einfacher Syntax

Durch Verwendung von TTCN-3 kommt eine höhere Programmiersprache für die Implementierung und Ausführung der Testfälle im modTF zum Einsatz. Durch die Auslegung von TTCN-3 als Beschreibungssprache für Testfälle, ist die Syntax für diesen Anwendungsfall leicht zu erlernen.

Anforderung 2 – Verwendung einer Programmiersprache mit großem Funktionsumfang

TTCN-3 bietet mit sehr vielen in die Sprache integrierte Funktionen, wie zum Beispiel *match*, *send*, *receive*, *bit2int*, *bit2hex* oder *testcasename*, einen großen Funktionsumfang. Dieser lässt sich über Module erweitern. Ebenso stehen eine Vielzahl von Datentypen, wie zum Beispiel *universal charstring*, *bitstring*, *record* oder *union*, zur Verfügung. Für den Test ist vor allem der Datentyp *verdicttype*, mit dem Methoden *setverdict* und *getverdict*, für Bewertungen von Testfällen sehr hilfreich. Eine Übersicht der Sprachelemente ist unter [111] zu finden.

Anforderung 3 – Verwendung einer Programmiersprache mit verschiedenen Abstraktionsgraden

TTCN-3 erlaubt die Erstellung von Testfällen auf verschiedenen Abstraktionsebenen. Es ist möglich einen Test zur Prüfung eines einzelnen Signales mit den gleichen Ausdrucksmittel zu erstellen, wie ein Test eines gesamten Netzwerkprotokolls. Einerseits ermöglicht der Matching-Mechanismus für Daten auch sehr komplexe Strukturen, wie Netzwerkprotokolle einfach zu prüfen, wie dies in [112, S.199 ff] dargestellt ist. Andererseits ermöglicht TTCN-3 einen einfachen Umgang mit verteilten Systemen, wie dies in [112, S.77 ff] erläutert ist. Es ist damit, je nach Testziel, eine frei wählbare Abstraktion möglich.

Anforderung 4 – Verwendung einer Programmiersprache zur Auswertung einzelner Werte und von Signalverläufen

Die Prüfung einzelner Signale bzw. Pakete ist bei TTCN-3 mittels einer *alt*-Struktur möglich. Diese Prüfung erfolgt zyklisch, bis eine der definierten Alternativen erfüllt ist. Die Prüfung von Signalverläufen ist mittels der Erweiterung *TTCN-3 Performance and Real-time Testing* [113] umsetzbar, wie dies in [114] dargestellt ist.

Anforderung 5 – Verwendung einer Programmiersprache mit der Unterstützung einer hohen Ausführungsgeschwindigkeit

TTCN-3 erlaubt die Umsetzung als compilierte oder interpretierte Sprache. Wie im TEMEA-Projekt [115] gezeigt werden konnte, sind damit hohe Ausführungszeiten möglich. Im Rahmen dieser Arbeit ist die Ausführung von TTCN-3 als Interpreter umgesetzt worden. Die erreichte Performanz, welche im Kapitel 6 dargestellt wird, zeigte sich mit anderen Testautomatisierungslösung mindestens ebenbürtig.

Anforderung 6 – Verwendung einer Programmiersprache mit der Möglichkeit zur modellbasierten bzw. grafischen Darstellung und Änderung des Programmablaufes

Die Spezifikation des GFT-Formates erlaubt die grafische Modellierung von Testfällen. Damit erfüllt TTCN-3 die gestellt Anforderung. Im Rahmen dieser Arbeit war die Umsetzung des Standards nicht möglich. Aus diesem Grund ist die gestellte Anforderung für modTF nur in der Planung, jedoch nicht in der Umsetzung erfüllt worden.

Anforderung 7 – Verwendung von gängigen Darstellungsmethoden, wie zum Beispiel UML

Unter Anwendung der Spezifikation für das GFT-Format ist es für TTCN-3 ebenfalls möglich andere grafische Darstellungen zu realisieren. Dies ist beispielsweise auch mit dem Tool .mzT [70] der Firma sepp.med möglich. Eine verwendbare Umsetzung war im Rahmen dieser Arbeit nicht möglich.

Anforderung 8 – Parallele Verwendung von verschiedenen Darstellungsmethoden

Die Spezifikation für das GFT-Format sieht eine parallele Verwendung zu TTCN-3 in textueller Form vor. Entsprechend der fehlenden Umsetzung des GFT-Formates im Rahmen des modTF, ist diese Anforderung nur in der Planung erfüllt.

Anforderung 9 – Verwendung einer Programmiersprache mit Debug-Möglichkeit

Die Spezifikation von TTCN-3 geht auf eine Debug-Möglichkeit nicht direkt ein. Dies ist eine Anforderung an die eigentliche Umsetzung. Wie in Abschnitt 5.3.3 dargelegt wurde, ist im Rahmen dieser Arbeit die Debug-Möglichkeit über PyDev und die Python-basierte Implementierung der Interpreter verwendet worden.

Anforderung 10 – Verwendung einer Entwicklungsumgebung mit Debug-Möglichkeit

Wie in Abschnitt 5.3.2 beschrieben, ermöglicht der Debug-Framework eine Realisierung von spezifischen Debug-Funktionen für die jeweiligen Sprachen. Diese Funktionalität wurde für modTF bisher nicht umgesetzt.

Anforderung 38 – Bereitstellung von Funktionalität zur hierarchischen Gliederung der Testfälle in mehreren Ebenen

Die Gliederung von Testfällen ist in TTCN-3 mittels des *control*-Abschnittes der Testskripte und den darin enthaltenen *execute*-Aufrufen möglich. Es ist damit auch möglich dies über mehrere Ebenen hierarchisch zu gliedern.

Anforderung 42 – Bereitstellung einer bidirektionalen Schnittstelle zu Requirementsmanagement-Systemen

Für das Test-Beschreibungs-Modul stellen die IDs der Anforderungen die notwendige Schnittstelle zu den Requirementsmanagement-Systemen dar. Bei einer maschinenlesbaren Zuordnung der Testfälle zu Anforderungen, kann diese Verknüpfung automatisierte abgebildet werden.

Anforderung 43 – Bereitstellung von Funktionen zur Verknüpfung von Testfällen mit Anforderungen (z.B. Anforderungs-ID)

Die Verknüpfung der Testfälle mit Anforderungen erfolgt durch Anforderungs-IDs, welche über *log*-Aufrufe, wie in Abschnitt 5.3.3 beschrieben, an die Testausführung und die Testdokumentation weitergegeben werden.

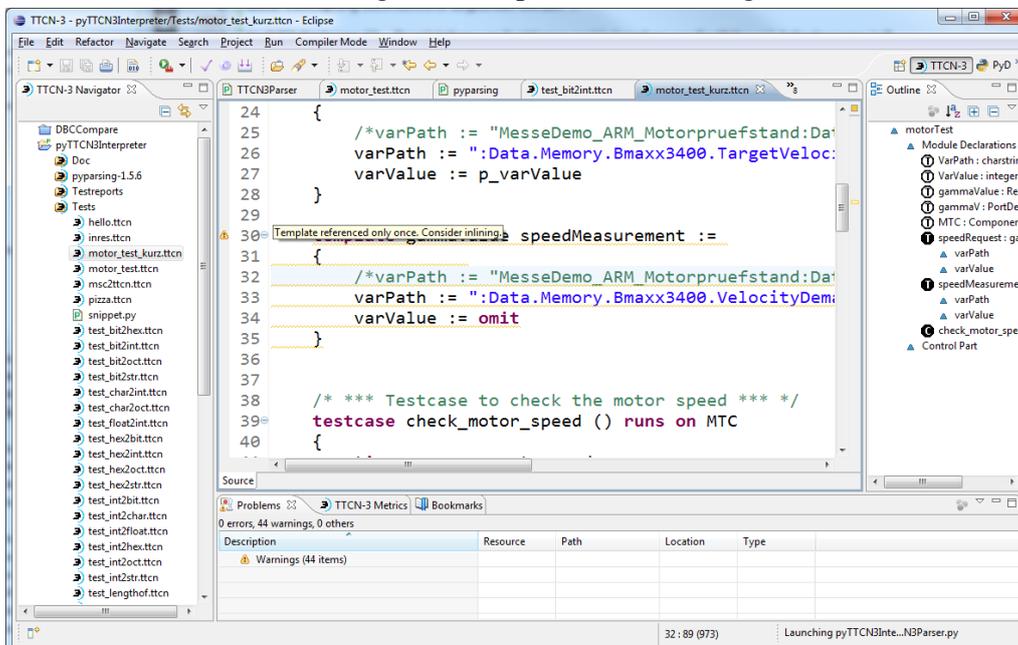
5.3.6. Anwendung des Moduls

Die Verwendung des TRex-Plugins erfordert zuerst die Installation von Eclipse. Danach ist es über die Softwareverwaltung von Eclipse möglich das TRex-Plugin [82] unter Verwendung der URL <http://www.trex.informatik.uni-goettingen.de/repository> zu installieren. Im Rahmen dieser Arbeit wurde zusätzlich das PyDev-Plugin [107] verwendet.

In Abbildung 5.6 ist die Arbeitsoberfläche mit der Ansicht des TRex-Plugins zu sehen. Auf der linken Seite ist die Projektansicht mit dem TTCN-3 Testskripten zu sehen. In der Mitte ist der Quelltext eines Testskriptes zu sehen. Dabei ist zu erkennen, dass die Code-Analyse von TRex Warnungen zu Code-Stellen ausgibt und damit die Entwicklung fehlerfreier Testskripte unterstützt. Auf der rechten Seite ist die Übersicht über alle Elemente des Testskripts zu sehen.

Eine Erläuterung zum Einstieg in TRex ist unter [116] zu finden.

Abbildung 5.6.: Eclipse mit TRex-Plugin



5.4. Testausführung

In diesem Abschnitt wird die Umsetzung des Test-Ausführungs-Moduls dargestellt. Es werden dabei verschiedene, erarbeitete Lösungsansätze präsentiert und die endgültige Umsetzung beschrieben.

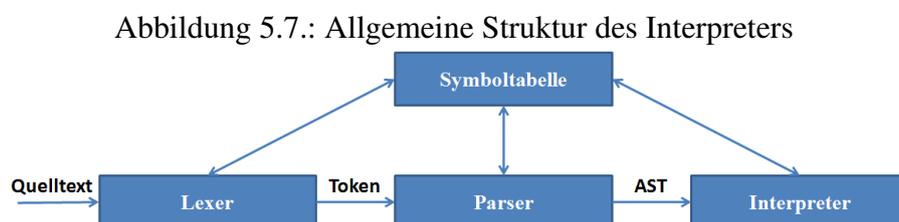
5.4.1. Umsetzung des Lösungskonzeptes in die Softwarearchitektur

5.4.1.1. Vorbetrachtungen

Auf Basis des in Abschnitt 4.3 erarbeiteten Konzeptes für die Testausführung wird nun eine Softwarearchitektur für das Test-Ausführungs-Modul erarbeitet. Das Konzept sieht die Umsetzung von TTCN-3 in Form eines Interpreters vor. Der Aufbau eines Interpreters unterteilt sich dabei in zwei Komponenten, den Parser und den Interpreter [117, S. 3ff]. Der Parser dient zum Einlesen der Zeichen des Quelltexts und der Umwandlung in einen Syntax-Baum. Der Interpreter liest die Token des Syntaxbaumes und führt diese aus. Des Weiteren ist für die Ausführung eine Symboltabelle mit allen Symbolen (z. B. Variablennamen und Funktionsnamen) notwendig. Diese Tabelle kann durch den Parser aufgebaut werden und wird während der Ausführung um die aktuellen Werte – z. B. von Variablen – erweitert.

5.4.1.2. Interpreter

Für die Implementierung der Komponenten des Interpreter gibt es verschiedene Design Pattern welche unter anderem in [117] beschrieben sind. Diese Design Pattern bilden die Softwarearchitektur des Interpreters, da diese unabhängig von der gewählten Implementierungssprache sind. Der allgemeine Aufbau eines Interpreters auf Basis der Design Pattern ist in Abbildung 5.7 dargestellt.



Der erste Schritt für die Verarbeitung des Quelltextes ist das Einlesen des Quelltextes und die Umwandlung in Token. Dies wird durch den Lexer, welcher als Teil des Parsers zu betrachten ist, realisiert. Da die manuelle Implementierung eines Lexers aufwändig und

fehleranfällig ist, wurde entschieden einen Generator zu verwenden. Dieser soll auf Basis der Grammatik-Beschreibung von TTCN-3, welche in der Backus-Naur-Form (BNF) in [56, Annex A] spezifiziert ist, einen Lexer in Python-Code erstellen. Da sich die Implementierung des Lexers im Rahmen der Umsetzung des Interpreters als sehr problematisch erwiesen hat, werden die verschiedenen Lösungsansätze in Abschnitt 5.4.2 genauer erläutert.

Für den eigentlichen Parser stehen verschiedene Lösungsmöglichkeiten zur Verfügung. Es wäre möglich den Parser auch direkt zur Ausführung des Codes, also zur Umsetzung der Interpreter-Funktionalität, zu verwenden. Dies ist aber für die Übersichtlichkeit und Wartbarkeit des Codes nicht sinnvoll. Im Rahmen dieser Arbeit soll eine klare Trennung der Teil-Module eingehalten werden, um neben der Übersichtlichkeit und Wartbarkeit, auch die Erweiterbarkeit sicher zu stellen. In Anbetracht der Komplexität der Grammatik – 565 Regeln in der BNF – konnte vom Auftreten von Mehrdeutigkeiten beim Einsatz einfacher Parser, wie zum Beispiel eines LL(1) Parsers [117, S. 38ff] ausgegangen werden. Es wurde daher für die Umsetzung in dieser Arbeit ein LL(k)-Parser [117, S. 43ff] bzw. ein Parser mit dynamischem Lookahead [117, S. 50ff] vorgesehen. Es war auch hier der Einsatz eines Generators angedacht. In Abschnitt 5.4.2 ist das Vorgehen genauer erläutert.

Als Ausgabe des Parser wird ein abstrakter Syntaxbaum – auch Abstract Syntax Tree (AST) genannt – verwendet. Für die Form des AST gibt es verschiedene Umsetzungsmöglichkeiten.

Es sind dabei die folgenden Typen zu unterscheiden:

- Parse Tree [117, S. 90ff]
- Homogeneous AST [117, S. 94ff]
- Normalized Heterogeneous AST [117, S. 96ff]
- Irregular Heterogeneous AST [117, S. 99ff]

Als bester Kompromiss zwischen Erstellungsaufwand des AST und Ausführungsgeschwindigkeit des Interpreters wurde die Umsetzung eines Normalized Heterogeneous AST vorgesehen.

Für den Interpreter gibt es ebenfalls verschiedene Implementierungsmöglichkeiten. Da eine möglichst direkte Ausführung von TTCN-3 realisiert werden sollte, um eine möglichst gute Ausführungsgeschwindigkeit zu ermöglichen, wurde die Implementierung eines High-Level Interpreters vorgesehen, da eine Generierung von Zwischen-Code (z. B. Byte-Code) als zu aufwendig erachtet wurde. Für die Ausführung des, durch den Parser erstellten AST, wurde der Ansatz eines *Tree-Based Interpreters* [117, S. 230] gewählt.

Neben den hier erläuterten Komponenten ist für einen Interpreter eine Symboltabelle notwendig, welche alle Symbole (z. B. Variablen und Funktionen) enthält. Die Symboltabelle wird dabei vom Lexer, Parser und Interpreter verwendet und jeweils um weitere Informationen ergänzt. Weiterhin kann die Symboltabelle die Namensräume (z. B. Variablen in eine Funktion) abbilden. Da TTCN-3 verschachtelte Aufrufe von Funktionen und dafür getrennte Namensräume vorsieht, ist die Umsetzung einer Symboltabelle mit Namensräumen notwendig. Dies erfolgt in Form einer *Symbol Table for Nested Scopes* [117, S. 146ff].

5.4.1.3. Schnittstellendefinition

Der Interpreter selbst bildet das TTCN-3 Executable, wie es in Abbildung 4.3 dargestellt ist. Daraus ergibt sich, dass für die Implementierung des Interpreters die beiden Interfaces TCI und TRI umzusetzen sind. Die Beschreibung dieser Interfaces ist in [59] und [58] zu finden. Diese beiden Dokumente enthalten jeweils eine allgemeine Beschreibung des Interfaces und die Umsetzung für ANSI C, C++, C# und Java. In Anbetracht des breiten Einsatzes zur Testautomatisierung und dem Vorhandensein von Bibliotheken für den Zugriff auf verschiedene Testsystem (z. B. dSPACE), sollte Python als Implementierungssprache für das Test-Ausführungs-Modul dienen. Es war daher notwendig die Schnittstellen TCI und TRI für Python zu spezifizieren. Als Grundlage dafür diente die allgemeine Schnittstellenbeschreibung sowie die Implementierungsvorgabe für Java. Da Java von allen spezifizierten Sprachen die größte Ähnlichkeit zu Python aufweist.

Das TCI stellt die Schnittstelle zum Test Management and Control (TMC) dar. Die Verknüpfung zwischen den Aufrufen von TTCN-3 und den dazugehörigen Funktionen des TCI werden in [59, Table 1 p. 24] und in [59, Table 2 p. 25] dargestellt. Im ersten Schritt war das Mapping der TTCN-3 Datentypen auf die Datentypen von Python zu definieren. Dies erfolgt in Anlehnung an [59, Table 3 p. 104] und ist in Tabelle 5.1 dargestellt.

Tabelle 5.1.: TCI – Mapping der Datentypen von TTCN-3 auf Python

TTCN-3 Typen (IDL Typen)	Python-Typen
TBoolean	bool
TFloat	float
TInteger	int
TString	str
TStringSeq	[str, ...] (Liste von str)
TChar	str (mit einem Zeichen)
TUniversalChar	[unicode, ...] (Liste von unicode)

Es wurden alle definierten Klassen und Methoden in Python deklariert. Im Rahmen der Erläuterung der Implementierung in Abschnitt 5.4.2 werden einzelne Klassen und Methoden genauer besprochen.

Das TRI stellt die Schnittstelle zum Testsystem und zum System unter Test dar. Die Verknüpfung zwischen den Aufrufen von TTCN-3 und den zugehörigen Funktionen des TRI werden in [58, Table 2 p. 16] dargestellt. Es wurden alle definierten Klassen und Methoden in Python deklariert.

5.4.1.4. Funktionsbeschreibung der Komponenten von TTCN-3

In diesem Abschnitt werden die einzelnen Funktionen der Komponenten der TTCN-3 Laufzeitumgebung beschrieben, welche über die Schnittstellen TCI und TRI angesprochen werden.

Das Test Management (TM) ist für die Ablaufsteuerung der Tests verantwortlich. Diese Komponente ist auch bei verteilten Testsystemen nur einmal vorhanden.

Das Component Handling (CH) ist für das Verwalten der Komponenten bei der parallelen Ausführung von Tests auf verschiedenen Systemen verantwortlich. Es synchronisiert die Abläufe auf den Testsystemen und verteilt die Message- und Prozedur-Aufrufe des TE an die entsprechenden, entfernten Komponenten.

Coding/Decoding (CD) dient der codieren und decodieren von Daten des DUT in die Daten bzw. Datentypen von TTCN-3. Diese Komponenten sind Testsystem bzw. DUT spezifisch und müssen gegebenenfalls angepasst werden.

Das Logging (TL) ist Schnittstelle zum Test-Reporting. Es führt alle Log-Ausgaben aus. Diese umfassen, neben den log-Aufrufen direkt im TTCN-3, ebenfalls die Ausgaben des TE während des Aufrufen der verschiedenen Funktionen des TCI und TRI.

Das TTCN-3 Executable (TE) stellt das Run Time System bzw. die Laufzeitumgebung von TTCN-3 dar. In dieser Arbeit umfasst dies den Lexer, Parser und Interpreter. Es dient der Ausführung der TTCN-3 Testfälle.

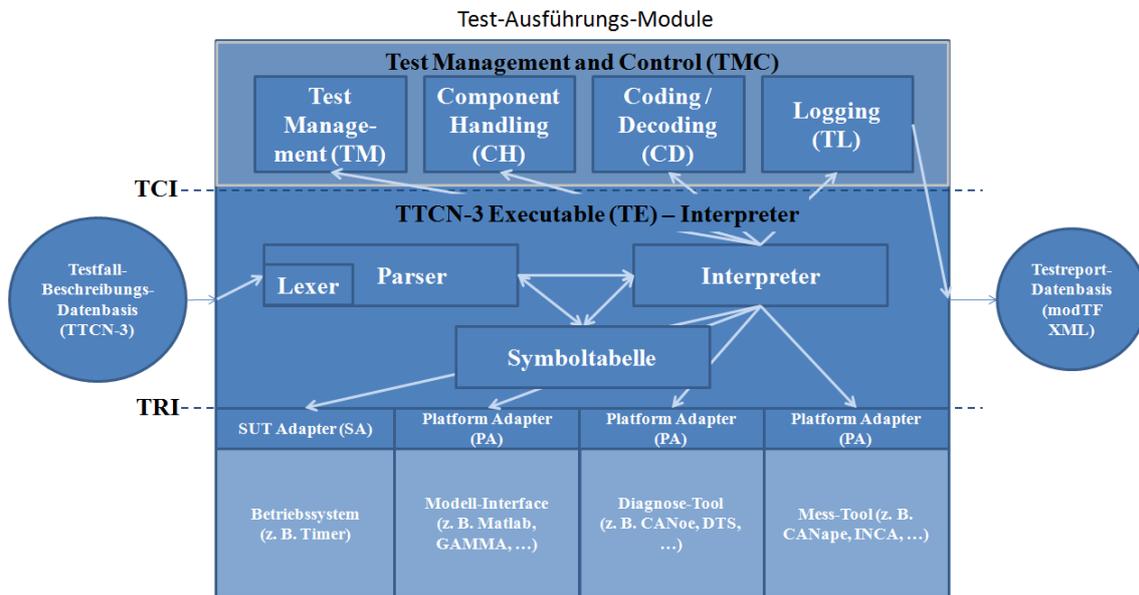
Der Platform Adapter (PA) dient der Anbindung an das Testsystem bzw. an das Betriebssystem. Es beinhaltet die Funktionen für die Zeitbestimmung und Timeout-Überwachung. Diese Komponente muss für jedes Testsystem angepasst werden.

Der SUT Adapter (SA) dient der Anbindung des SUT bzw. DUT. Dies umfasst neben der direkten Anbindung des DUT – z. B. über digitale oder analoge Signale – auch die Anbindung von Testtools. Dies können zum Beispiel Diagnose-Werkzeuge oder Messsoftware sein. Die Adapter sind spezifisch für das DUT bzw. das jeweilige Testtool zu implementieren. Dabei ist zu beachten, dass alle Funktionen re-entrant sein müssen, da diese in parallelen Prozessen aufgerufen werden können.

5.4.1.5. Gesamtarchitektur

Die in den vorangegangenen Abschnitten dargestellten Überlegungen wurden zu einer vollständigen Softwarearchitektur für das Test-Ausführungs-Modul zusammengeführt. In Abbildung 5.8 ist das Ergebnis inklusive alle Signalverbindungen dargestellt.

Abbildung 5.8.: Struktur des Test-Ausführungs-Moduls



5.4.2. Implementierung der Softwarearchitektur

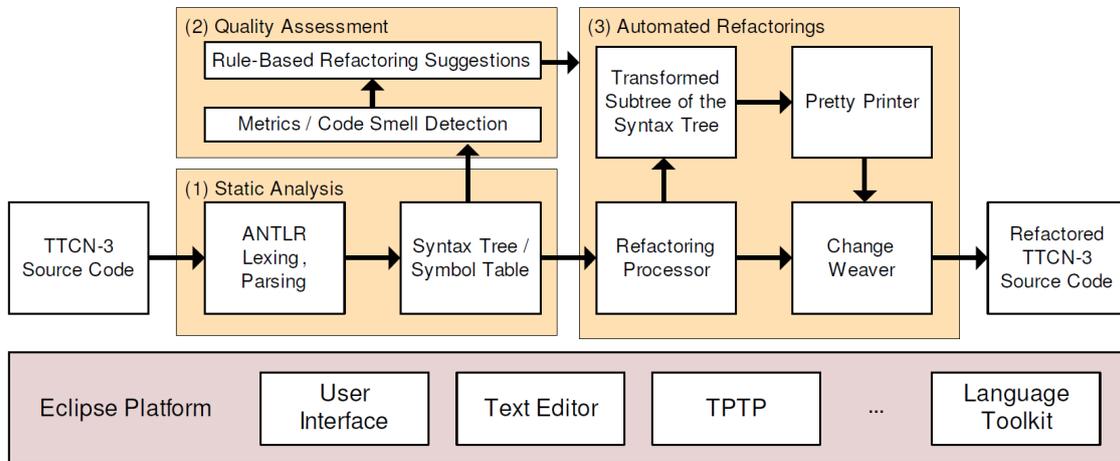
In diesem Abschnitt wird die im Rahmen dieser Arbeit durchgeführte Implementierung, der in Abschnitt 5.4.1 dargelegten Softwarearchitektur, erläutert. Da bei der Implementierung verschiedene Probleme aufgetreten sind und einzelne vielversprechende Lösungsansätze verworfen werden mussten, werden auch die fehlgeschlagenen Implementierungsansätze kurz dargestellt. Das Hauptaugenmerk liegt jedoch auf der erfolgreichen Umsetzung.

5.4.2.1. Interpreter auf Basis des TRex-Parsers

Das für die Testbeschreibung verwendete Eclipse-Plugin TRex beinhaltet, wie in Abbildung 5.9 dargestellt, eine Implementierung für einen Lexer, Parser sowie für eine Symboltabelle [118]. Der verwendete Parser-Generator ANTLR [119, 117] ermöglicht es, neben dem für das TRex-Plugin generierten Java-Code, auch Python-Code zu generieren. Als Ausgangsdaten dient eine Grammatik-Beschreibung, welche an die BNF angelehnt ist und EBNF genannt wird. Der generierte Parser arbeitet nach dem LL-Prinzip.

Bei ANTLR ist es möglich eigene Codeabschnitte in die Grammatik-Beschreibung einzubinden, um das Verhalten des generierten Parsers zu beeinflussen. Von dieser Funktion wird beim TRex-Plugin stark Gebrauch gemacht. Des Weiteren gibt es eine enge Verflechtung zwischen dem Parser und der Symboltabelle, welche ebenfalls über eingebetteten Java-Code realisiert wird. Eine weitere Untersuchung hat gezeigt, dass die vorhandenen

Abbildung 5.9.: Struktur des TRex-Plugin [118]



Beschreibungsdateien der Grammatik nur mit großem Aufwand für die Realisierung eines TTCN-3 Parser in Python nutzbar gemacht werden können. Dieser Lösungsansatz wurde nicht weiter verfolgt.

5.4.2.2. Interpreter unter Verwendung der Gold Parsing System

Zur Umsetzung des TTCN-3 Interpreters wurde eine Recherche bezüglich Parser-Generatoren gestartet. Es sollte ein Parser in der Sprache Python auf Basis der gegebenen BNF erzeugt werden können. Als vielversprechender Kandidat zeigte sich das Gold Parsing System [120]. Dieses System ermöglicht die vollständige Trennung zwischen Grammatik und dem Code. Ebenfalls stellt das Tool eine umfangreiche Debug-Möglichkeit für die grammatischen Regeln zur Verfügung. Es wird dabei ein Parser nach dem LALR-Prinzip erzeugt. Leider zeigte sich im Laufe der Implementierung, dass die gegebenen Nichteindeutigkeiten der Grammatik mit dem LALR-Parser nicht auflösbar waren, da dieser einen Lookahead von 1 verwendet. Dieser Lösungsansatz wurde nicht weiter verfolgt.

5.4.2.3. Interpreter unter Verwendung der Bibliothek PyParsing

Vorbetrachtung

Weitere Recherchen zur Umsetzung eines TTCN-3 Interpreters in Python zeigten, die Bibliothek PyParsing [121] als möglich Lösung. Die Eingabe der Grammatik erfolgt als Python-Code, welche mit Hilfe von PyParsing stark an die BNF angelehnt ist. Die beiden Listings 5.5 und 5.6 zeigen eine Gegenüberstellung für eine Grammatikregel.

Listing 5.5: BNF-Darstellung

```
ModuleId ::= Identifier [LanguageSpec]
```

Listing 5.6: PyParsing-Darstellung

```
ModuleId << pp.Group(GlobalModuleId + pp.Optional(LanguageSpec))("ModuleId")
```

Im Rahmen der Recherche konnte unter [122] ein erster Versuch der Umsetzung eines TTCN-3 Parser mit PyParsing aufgefunden werden. Die genannte Implementierung war in der Lage TTCN-3 Dateien einzulesen. Sie war nicht vollständig und basierte auf einer veralteten Version des TTCN-3 Standards, welche auch nicht dokumentiert war.

Im Rahmen dieser Arbeit wurde auf Basis der Version 4.4.1 eine neue Implementierung realisiert. Im Laufe der Umsetzung wurde die Version 4.5.1 des Standards veröffentlicht, welche ebenfalls in dieses Projekt eingeflossen ist.

Lexer und Parser

Die Implementierung des Lexers und Parsers erfolgt als Python-Package mit dem Namen *ttn3parser*. Dieses Package beinhaltet die Datei *Bnf.py*, welche die Grammatik-Beschreibung enthält. Weiterhin umfasst diese Datei die Aufrufe von Funktionen, welche während des Parsens ausgeführt werden. Diese Funktionen sind in der Datei *ParseActions.py* implementiert. Neben der Durchführung der Konsistenzprüfung, erstellen diese Funktionen auch die Symboltabelle. Dabei werden die Symbole sowie die Namensräume – Scopes genannt – angelegt. Die Scopes werden entsprechend ihrer Abhängigkeit verkettet, um das Auffinden von Symbolen während der Ausführung zu ermöglichen.

Als Ergebnis generiert der Parser einen AST, welcher durch PyParsing als Objekt des Typs *ParseResults* generiert wird. Durch die Funktionen aus *ParseActions.py* wurde das *ParseResults* um Informationen, wie eine numerische Darstellung von Konstanten und die Verknüpfung mit der Symboltabelle erweitert. Durch Auswertung des *ParseResults* ist es im Interpreter damit möglich den TTCN-3 Code auszuführen.

Symboltabelle

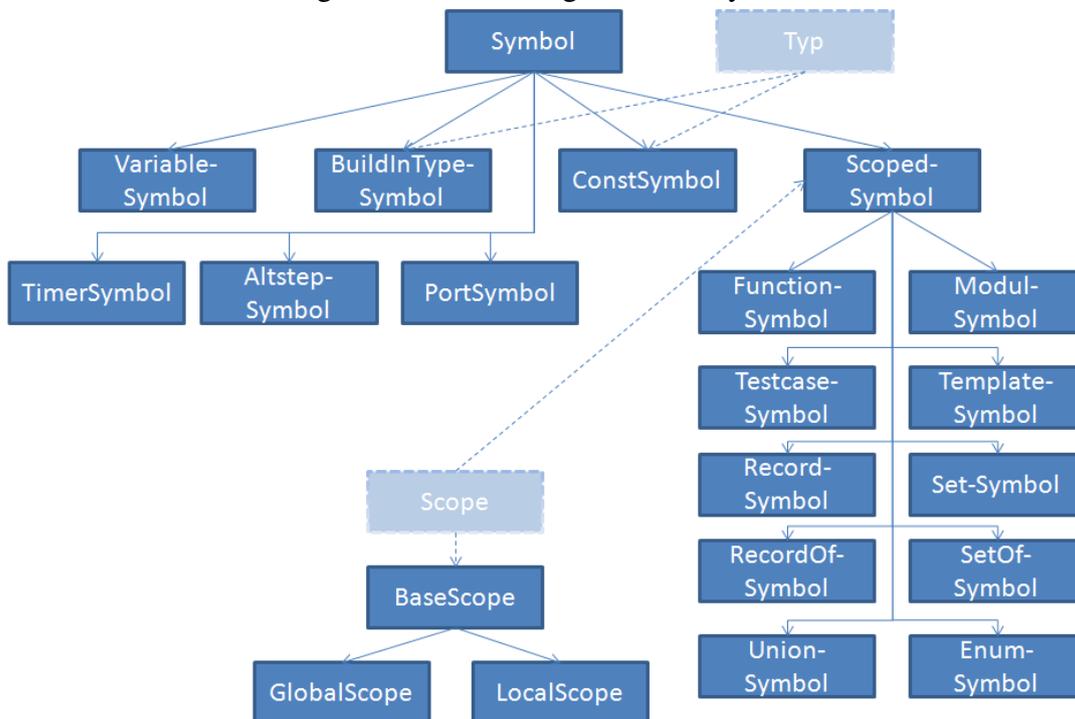
Die Symboltabelle ist als Package mit dem Namen *Symboltable* innerhalb des Packages *ttn3parser* implementiert. Die Struktur der Symboltabelle ist in Abbildung 5.10 als Klassendiagramm dargestellt. Jede Klasse ist in einer eigenen Python-Datei implementiert.

Interpreter

Der Interpreter, welcher die eigentliche Ausführung übernimmt, wurde im Package *ttn3interpreter* implementiert. Das Package umfasst die Dateien *Interpreter.py* und *InterpreterException.py*.

In der Datei *InterpreterException.py* ist eine Hierarchie von Klassen für die Exceptions

Abbildung 5.10.: Klassendiagramm der Symboltabelle



enthalten. Die Zusammenhänge der Exceptionklassen sind in Abbildung 5.11 als Klassendiagramm dargestellt.

Alle Exceptions erben hierbei von der Python-Basisklasse für Exceptions, welche dem Namen *Exception* trägt. Es sind zwei Gruppen von Exceptions zu unterscheiden. Die erste Gruppe hat die Klasse *ExecutionException* als Basisklasse. Die von ihr abgeleiteten Exceptions zeigen Laufzeitfehler bei der Ausführung des TTCN-3 Codes an. In Tabelle 5.2 sind die Klassen mit den zugehörigen Fehlerbeschreibungen aufgelistet.

Die zweite Gruppe hat die Klasse *ControlFlowException* als Basisklasse. Die von dieser Klasse abgeleiteten Exceptions dienen der Steuerung des Kontrollfluss des Interpreters. In Tabelle 5.3 sind die Klassen mit den zugehörigen Fehlerbeschreibungen aufgelistet.

In der Datei *Interpreter.py* ist die Ausführung des Codes implementiert. Durch die Verwendung eines Normalized Heterogeneous AST ist es notwendig, für jeden Knotentyp des AST eine Verarbeitungsfunktion zu implementieren. Die Verarbeitung erfolgt in Form eines Tree-Walkers. Dies bedeutet, dass für jeden Knoten im AST geprüft wird, welche Kindknoten vorhanden sind. Für jeden Kind-Knoten wird eine, dem jeweiligen Typ entsprechende, Verarbeitungsfunktion aufgerufen. Im Listing 5.7 ist dies beispielhaft für ein *module* von TTCN-3 dargestellt.

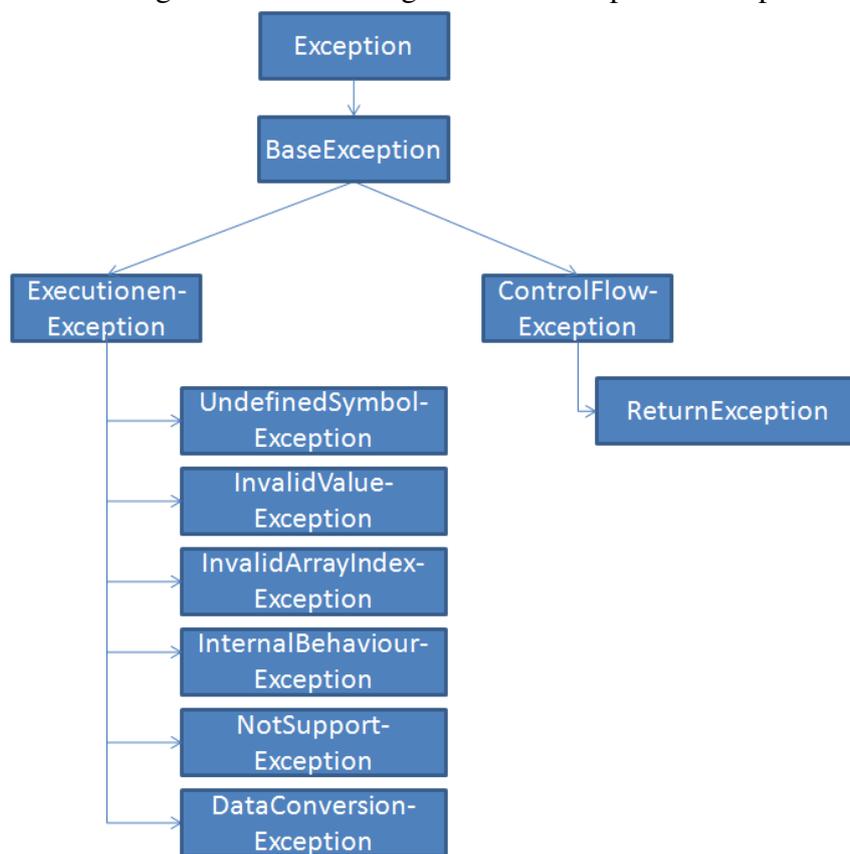
Tabelle 5.2.: TTCN-3 Interpreter – Exception für Laufzeitfehler

Exceptionklasse	Bedeutung
UndefinedSymbolException	Es wurde ein Symbol (z. B. eine Variable) referenziert, welches im aktuellen Namensraum und allen übergeordneten Namensräumen nicht existiert.
InvalidValueException	Es wurde eine Zuweisung von inkompatiblen Datentypen durchgeführt.
InvalidArrayIndexException	Es wurde auf einen Array-Index zugegriffen, der nicht existiert.
InternalBehaviourException	Bei der Ausführung ist ein Fehler, wie zum Beispiel ein fehlender AST-Knoten für einen Codeabschnitt, aufgetreten. Dies deutet auf einen Fehler im TTCN-3 Code hin.
NotSupportException	Es wurde eine noch nicht unterstützte Instruktion ausgeführt. Dies ist notwendig, da der aktuelle Interpreter nicht 100 % aller Sprachmittel von TTCN-3 unterstützt.
DataConversionException	Eine Datenkonvertierung konnte nicht ausgeführt werden.

Tabelle 5.3.: TTCN-3 Interpreter – Exception für Flusskontroller

Exceptionklasse	Bedeutung
ReturnException	Rückkehr aus einem Funktionsaufruf oder einem Testcase. Der Rückgabewert ist Teil der Exception

Abbildung 5.11.: Klassendiagramm der Interpreter-Exceptions



Listing 5.7: TTCN-3 Interpreter – Funktion *module*

```

1  def module(self, pyparsing_result):
2      '''
3      handle the module of TTCN3
4      '''
5      # get scope
6      scope = pyparsing_result.get('current_scope', None
7      )
8      if scope != None:
9          self._current_scope = scope
10     for f in pyparsing_result:
11         if isinstance(f, pyparsing.ParseResults):
12             key = f.getName()
13             value = f
14         else:
15             # print f
16             continue
17         if (key == 'ModuleDefinitionsPart'):
18             # TTCN3 module definition part
19             self.module_definition_part(value)
20         elif (key == 'ModuleControlPart'):
21             # TTCN3 module control part -> start of
22             # execution
23             self.module_control_part(value)
24         elif (key == 'WithStatement'):
25             # with statement for module
26             self.with_statement(value)
27         elif (key == 'current_scope'):
28             self._current_scope = value
29         else:
30             # default part
31             # do nothing
32             pass

```

Innerhalb der Funktionen des Tree-Walkers werden alle Funktionsaufrufe zu den TTCN-3 Schnittstellen TCI und TRI ausgeführt. Die Aufrufe zu dem TRI umfassen beispielsweise das Starten eines Timers über den PA. Dies ist in Listing 5.8 dargestellt.

Listing 5.8: TTCN-3 Interpreter – Aufruf der TRI-Funktion *triStartTimer*

```
1  def start_timer_statement(self, pyparsing_result):
2      '''
3      handle the start timer statement of TTCN3
4      '''
5      timer_ref = self.timer_ref(pyparsing_result['
6          TimerRef'])
7      timer_value = pyparsing_result.get('TimerValue',
8          None)
9      if timer_value != None:
10         timer_value = self.timer_value(timer_value['
11             Expression'])
12         timer_value = self.get_value_from_ttcn3_type(
13             timer_value)
14     else:
15         timer_value = 0
16
17     # start timer
18     self._tri_platform_PA.triStartTimer(timer_ref[0],
19         timer_value)
```

Die Aufrufe zu dem TCI umfassen beispielsweise die Ausgabe von Log-Meldungen in den Test-Report. Dies ist beispielhaft in Listing 5.9 dargestellt.

Weiterführende Erläuterungen zur Implementierung des TCI und des TRI sind im folgenden Abschnitt aufgeführt.

Listing 5.9: TTCN-3 Interpreter – Aufruf der TCI-Funktion *tliLog*

```

1  def log_statements(self , pyparsing_result):
2      '''
3      handle log statement of TTCN3
4      '''
5      # get scope
6      scope = pyparsing_result.get('current_scope', None
7      )
8      if scope != None:
9          self._current_scope = scope
10         #init text to # print with ''
11         text_to_print = ''
12         # for development # print log
13         log_item = pyparsing_result['LogItem']
14         free_text = log_item.get('FreeText')
15         if free_text == None:
16             # Template instance to log
17             template_instance = log_item.get('
18             TemplateInstance')
19             while(not isinstance(template_instance , str)):
20                 template_instance = template_instance[0]
21
22             # get symbol by name
23             symbol = self._current_scope.resolve(
24                 template_instance)
25             if symbol != None:
26                 # get value to name
27                 name = symbol.getName()
28                 value = symbol.getValue()
29                 text_to_print = "%s:_%s" % (name, value)
30             else:
31                 #unknown element to log
32                 raise UndefinedSymbolException()
33         else:
34             text_to_print = str(free_text)
35         # call log interface
36         self._tci_TL_provided.tliLog("", "", "", "", "",
37             text_to_print)

```

TTCN-3 Schnittstellenimplementierung – TCI und TRI

Die Implementierung der beiden Schnittstellen TCI und TRI basiert auf der Umsetzung des TTCN-3-Standards in Python. Im Package *ttn3runtimebase* wurden alle Klassen und Methoden in Python umgesetzt. Dabei sind alle Plattform- bzw. Testsystem-unabhängigen Funktionen innerhalb dieses Packages implementiert wurden. Die implementierten Python-Dateien entsprechen dabei jeweils einer Klasse des TCI bzw. TRI. Für die Plattform- bzw. Testsystem-abhängigen Funktionen wurde das Package *ttn3runtimecurrent* angelegt. Wie schon aus der Struktur der TTCN-3 Laufzeitumgebung in Abbildung 5.8 zu erkennen ist, sind nur Bestandteile des TRI hier zu finden. Es sind dabei die folgenden Klassen implementiert worden:

- *TriCommunicationSA*
- *TriCommunicationTE*
- *TriPlatformPA*
- *TriPlatformTE*

Die Klassen *TriCommunicationTE*, *TriPlatformPA* und *TriPlatformTE* sind nur von Python-Funktionen abhängig und somit auf allen Plattformen mit Python-Interpreter lauffähig. In der Klasse *TriCommunicationSA* hingegen ist die Anbindung an das DUT enthalten. Um eine Neuimplementierung der Klasse für jedes DUT zu vermeiden, wurde die Schnittstelle zum DUT in ein eigenes Package Namens *ttn3ports* ausgelagert. Dieses Package stellt damit die Ports zum DUT zur Verfügung, welche direkt aus TTCN-3 heraus aufgerufen werden können. Über den *map*-Aufruf können die Ports angesprochen werden, ohne das Änderungen am Interpreter notwendig sind. Die Unterstützung neuer DUTs ist damit über neue Elemente im Package *ttn3ports* realisierbar. Die dynamische Einbindung der Ports ist in Listing 5.10 am Beispiel der Funktion *triMap* dargestellt.

Listing 5.10: TTCN-3 Interpreter – Dynamische Einbindung von Port in *triMap*

```

1 def triMap(self, compPortId, tsiPortId):
2     return_value = TriStatusbase(TriStatusbase.
3         TRI_ERROR)
4     try:
5         # try to import the defined by compPortId,
6         # tsiPortId
7         if isinstance(compPortId, TriComponentId):
8             # get name
9             comp_name = compPortId.getComponentName()
10        elif isinstance(compPortId, (str, unicode)):
11            comp_name = str(compPortId)
12        else:
13            return return_value
14        # try to import the defined by compPortId,
15        # tsiPortId
16        if isinstance(tsiPortId, TriPortId):
17            # get name
18            port_name = tsiPortId.getPortName()
19        elif isinstance(tsiPortId, (str, unicode)):
20            port_name = str(tsiPortId)
21        else:
22            return return_value
23        # dynamic import of the port library
24        pkg_module_string = "ttn3ports." + comp_name
25        submodule_string = port_name
26        my_module = __import__(pkg_module_string,
27            fromlist=[submodule_string])
28        my_class = getattr(my_module, submodule_string)
29        # call specific port implementation
30        my_instance = my_class(compPortId, tsiPortId,
31            self._triCommunicationTE)
32        # create instance of the port
33        # structure self.ports[comp_name][port_name]
34        if not self.ports.has_key(comp_name):
35            self.ports[comp_name] = {}
36        self.ports[comp_name][port_name] = my_instance
37        return_value.setStatus(TriStatusbase.TRI_OK)

```

Das Package *ttn3ports* beinhaltet die Datei *portsBase.py*, welche als Schnittstellendefinition für die Implementierung eines Ports dient. Dies ist in Listing 5.11 dargestellt. Im Rahmen der Erprobung des modTF, wurde unter anderem eine Anbindung an Testsysteme-

me mit der Echtzeit-Middleware Gamma V, in Form der Datei *gammaV.py* im Package *ttn3ports*, realisiert. Damit ist die einfache und effektive Anbindung von verschiedenen Testsystemen möglich.

Listing 5.11: TTCN-3 Interpreter – Schnittstellendefinition eines Ports

```
1 class portsBase(object):
2     '''
3     base class for the ports
4     '''
5     def __init__(self, compPortId, tsiPortId,
6                 triCommunicationTE = None, paramList = None):
7         '''
8         Constructor with the optional parameter list
9         '''
10        pass
11
12    def __del__(self):
13        '''
14        Destructor
15        '''
16        self.unmap()
17
18    def send(self, paramList, sutAddress, sendMessage):
19        '''
20        Implementation for the send command of the
21        TriCommunicationSA
22        '''
23        pass
24
25    def unmap(self, paramList = None):
26        '''
27        Implementation for the unmap and unmapParam
28        command of the TriCommunicationSA
29        '''
30        pass
```

Als letzter wichtiger Punkt soll in diesem Abschnitt die Anbindung an das Test-Reporting genauer betrachtet werden. Die Schnittstelle zum Testerporting bildet die TTCN-3 Komponente TL, welche durch die Klasse *TciTLProvided* im Package *ttn3runtimebase* implementiert ist. Diese Klasse realisiert mit Hilfe der Hilfsklasse *WriteXmlHelper* die Generierung des Test-Reports in Form des modTF-XML-Formates, welches im Abschnitt 5.5 näher erläutert wird. Für alle aufzuzeichnenden Ereignisse, wie zum Beispiel des Starten eines Tests, bietet die Klasse *TciTLProvided* jeweils eine eigene Methode, welche durch die zugehörigen Methoden in den anderen Klassen bzw. durch den Interpreter aufgerufen werden. Als Beispiel soll hier der *log*-Aufruf von TTCN-3 dienen. Die Methode *tliLog* ist in Listing 5.12 dargestellt.

Listing 5.12: TTCN-3 Interpreter – Methode *tliLog*

```

1      def tliLog(self, am, ts, src, line, c, log):
2          '''
3              The TL presents all the information provided in
4              the parameters of this
5              operation to the user, how this is done is not
6              within the scope of the
7              present document.
8
9              Returns: None
10             '''
11             # add info entry with reason
12             self._add_result(log, "INFO")

```

Die eigentliche Verarbeitung wird durch die Hilfsmethode *_add_result* realisiert, welche in Listing 5.13 in gekürzter Form dargestellt ist. Diese Methode realisiert das eigentliche Generieren der XML-Strukturen. Dies umfasst zum Beispiel auch die Verarbeitung von Anforderungs-IDs in Form von Log-Meldungen, welche den String *[[TC-ID]]* enthalten. Es werden ebenfalls die Ausgabe von Bilder oder binären Datenobjekten über die Strings *[[IMG]]* bzw. *[[OBJ]]* unterstützt.

Listing 5.13: TTCN-3 Interpreter – Methode `_add_result`

```
1     def _add_result(self, result_text, result_verdict):
2         if isinstance(result_text, CharstringValuebase):
3             result_text = result_text.getString()
4         # get object or testcase id from text
5         if result_text.find("[[TC-ID]]") != -1:
6             # get testcase id
7             start_of_testcase = result_text.find("[[TC-ID
8                 ]]") + len("[[TC-ID]]")
9             end_of_testcase = result_text.find("[[_TC-ID]]
10                ")
11             self._testcase_id = result_text[
12                 start_of_testcase:end_of_testcase]
13         image = None
14         if result_text.find("[[IMG]]") != -1:
15             # get image
16             start_of_image = result_text.find("[[IMG]]") +
17                 len("[[IMG]]")
18             end_of_image = result_text.find("[[_IMG]]")
19             image = result_text[start_of_image:
20                 end_of_image]
21         object_element = None
22         if result_text.find("[[OBJ]]") != -1:
23             # get image
24             start_of_object = result_text.find("[[OBJ]]")
25                 + len("[[OBJ]]")
26             end_of_object = result_text.find("[[_OBJ]]")
27             object_element = result_text[start_of_object:
28                 end_of_object]
29
30         resultElement = self._writeXmlObj.addElement(self._
31             _resultsElement, "result")
32         # description anlegen
33         descriptionElement = self._writeXmlObj.addElement(
34             resultElement, "description")
35         # description befüllen
36         self._writeXmlObj.addText(descriptionElement,
37             result_text)
38         # verdict anlegen
39         verdictElement = self._writeXmlObj.addElement(
40             resultElement, "verdict")
41         # description befüllen
42         self._writeXmlObj.addText(verdictElement,
43             result_verdict)
```

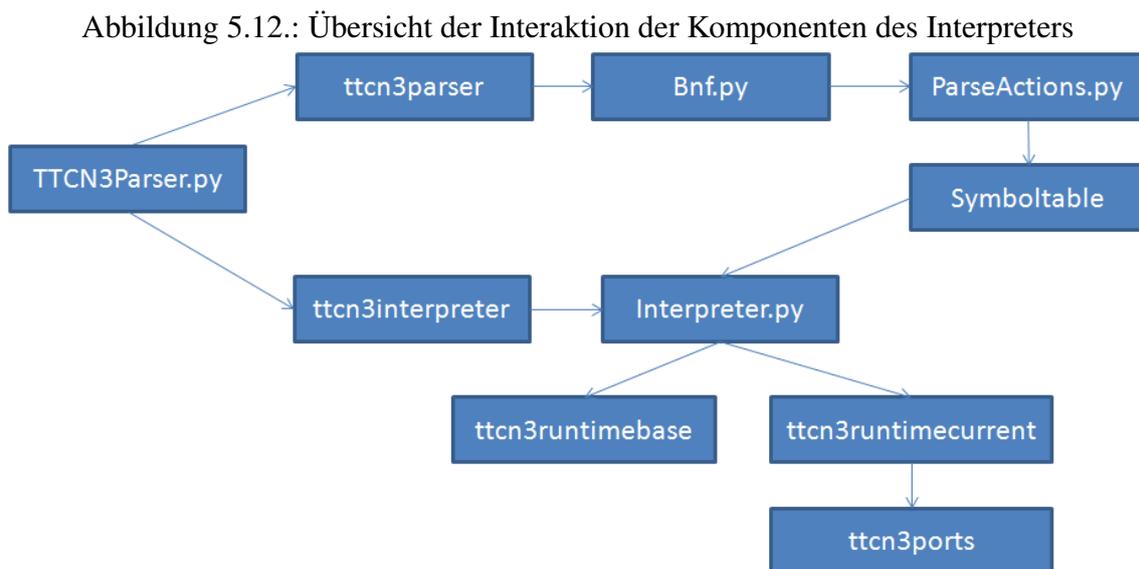
```

32     # testcase id anlegen
33     testcaseidElement = self._writeXmlObj.addElement(
34         resultElement, "testcase_id")
35     self._writeXmlObj.addText(testcaseidElement, self.
36         _testcase_id)
37     # object anlegen
38     if (image != None) or (object_element != None):
39         objectElement = self._writeXmlObj.addElement(
40             resultElement, "object")

```

Gesamtübersicht

In Abbildung 5.12 ist die Interaktion der Komponenten des TTCN-3 Interpreters des modTF dargestellt. Als Basis dient das Python-Skript *TTCN3Parser.py*. Dies stellt die Schnittstelle zum Interpreter dar und ruft die einzelnen Komponenten (Parser und Interpreter) auf. Im Rahmen dieser Arbeit konnte nicht der gesamte Sprachumfang von

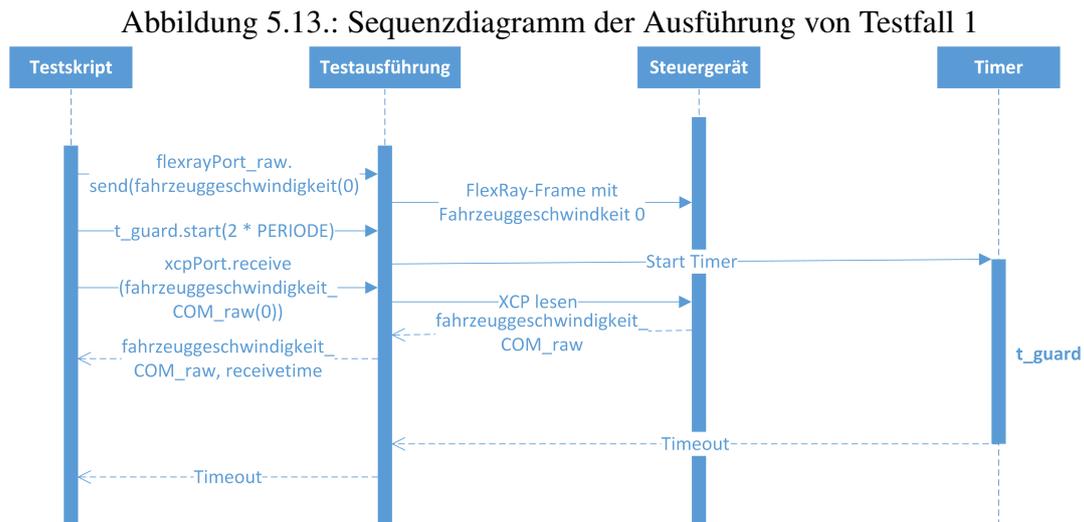


TTCN-3 im Interpreter implementiert werden. Der Parser sowie der generierte AST bilden alle Sprachelemente von TTCN-3 ab. Der Interpreter in Form des Packages *ttcn3interpreter* implementiert nicht alle möglichen Elemente des AST. Es wurden zuerst nur die notwendigen Funktionalität für die Evaluierung des modTF implementiert. Es besteht bisher keine Unterstützung, für den Test verteilter Systeme und die Kommunikation mit verteilten Testsystemen. Die Funktionen generieren eine Exception vom Typ *NotSupportException*.

5.4.3. Anwendung auf Beispiel-Testfälle

Im Folgenden werden die in Abschnitt 5.2 beschriebenen Testfälle ausgeführt. Dabei wird vor allem auf den zeitlichen Ablauf eingegangen. Es wird die Implementierung aus Abschnitt 5.3.4 betrachtet.

In Abbildung 5.13 ist die Ablauf des Testfalles 1 sowie die Interaktion mit den ver-

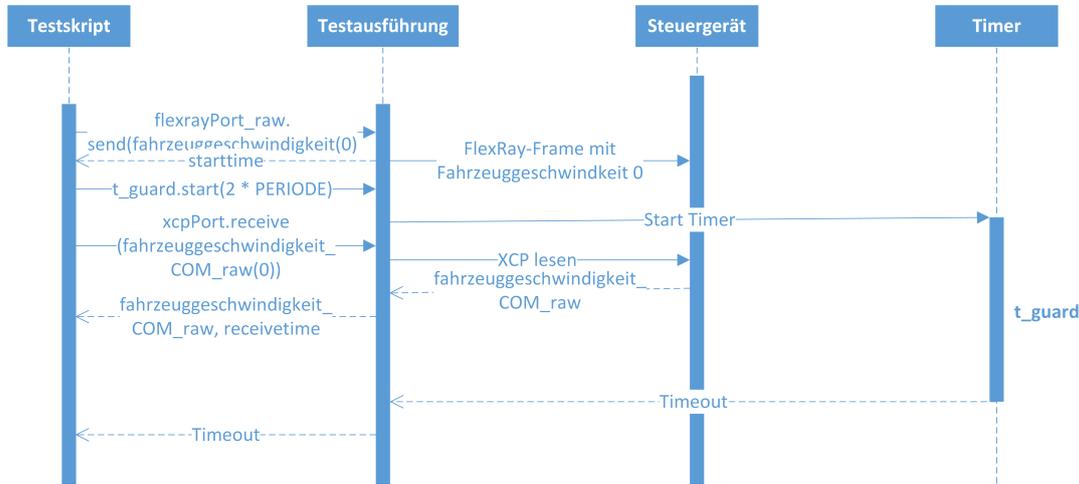


schiedenen Komponenten des Testsystems vereinfacht dargestellt. Zur besseren Übersicht wurden nur die für die Datenkommunikation wichtigen Aufrufe abgebildet. Wie der Abbildung zu entnehmen ist, arbeiten das Testsystem, bestehend aus dem Testskript und der Testausführung (TE-Komponente), sowie das Steuergerät parallel. Die Abfolge der Aufrufe ist durch das Testskript vorgegeben. Genauere zeitliche Anforderungen sind durch TTCN-3 an das Testsystem nicht gestellt. Das Zeitverhalten ist von dem jeweiligen Testsystem abhängig. Die einzige im Testfall definierte Zeit ist der `t_guard` für die Timeout-Überwachung. Hierbei stellt TTCN-3 nur die Anforderung, dass der Timer asynchron zum Testskript ausgeführt wird. Das Timeout wird als Event an das Testausführungssystem gemeldet. Die Verarbeitung erfolgt entsprechend der Aufrufe im Testskript.

Abbildung 5.14 zeigt den zeitlichen Ablauf von Testfall 2 unter Verwendung des *Performance and Real Time Testing*. Im Unterschied zu Testfall 1 werden nun Zeitstempel durch die Testausführung zurück an das Testskript gemeldet. Dabei wird die Zeitbasis durch das Testsystem zur Verfügung gestellt. Realisiert ist dies durch einen globalen Timer, welcher eine Zeit bezogen auf den Start des Testskripten liefert.

In Abbildung 5.15 ist der Ablauf für Testfall 2 unter Verwendung des *Support of interfaces with continuous signals* zu sehen. Im Unterschied zu der vorangegangenen Darstellung, werden nun hier die Werte der Ports mit einer Auflösung von 0,001 s aufgezeichnet. Für den Port `flexrayPort_raw` ist dies nicht dargestellt, da die Aufzeichnung innerhalb der Testausführung erfolgt. Für den Port `xcpPort` ist die zyklische Messung der Steuergeräte-

Abbildung 5.14.: Sequenzdiagramm der Ausführung von Testfall 2 – Performance and Real Time Testing



Variable mittels des XCP-DAQ-Modus dargestellt. Hier ist nun auch zu sehen, dass die Zeit t_{delta} definiert ist und durch die Testausführung sicher gestellt werden muss.

Testfall 3 stellt in Ergänzung zu Testfall 1 die Verwendung der CD-Komponente dar, wie dies in der Abbildung 5.16 zu sehen ist. Wie dem Sequenzdiagramm zu entnehmen ist, wird der physikalische Werte mittels der CD-Komponente in den Rohwert umgerechnet, bevor der Wert auf dem FlexRay-Bus gesendet wird.

Bei der Umsetzung des modTF wurden verschiedene Möglichkeiten von Timern und zur Steuerung des Zeitverhaltens vorgesehen. Im ersten und einfachsten Fall wird der Timer, für zum Beispiel die Timeout-Überwachung, mittels eines Times des Betriebssystems unter Verwendung des Python-Bibliothek *time*. Die Ausführung erfolgt als eigener Thread für jeden gestarteten Timer. Die Genauigkeit dieser Implementierung ist von verwendeten Betriebssystem und der Systemlast abhängig. Die Praxis ist damit eine zeitliche Auflösung von 100 ms möglich.

Für eine exaktere Messung der Zeit und eine genauere zeitliche Auflösung ist es notwendig, die Verarbeitung des Timers auf ein Testsystem mit deterministischer Ausführung zu überführen. Bei der Umsetzung des modTF wurde dies beispielhaft für die angebundene Middleware Gamma V vorgesehen. In diesem Fall wird die Verarbeitung in einem Raster von 1 ms ausgeführt. Die Ausführung wird durch die Middleware überwacht. Verletzungen der Ausführungszeit führen zu einem Abbruch des Tests. Weiterhin wurde die Umsetzung der Erweiterungen *Performance and Real Time Testing* und *Support of interfaces with continuous signals* mit Gamma V vorgesehen, jedoch nicht vollständig umgesetzt.

Abbildung 5.15.: Sequenzdiagramm der Ausführung von Testfall 2 – Support of interfaces with continuous signals

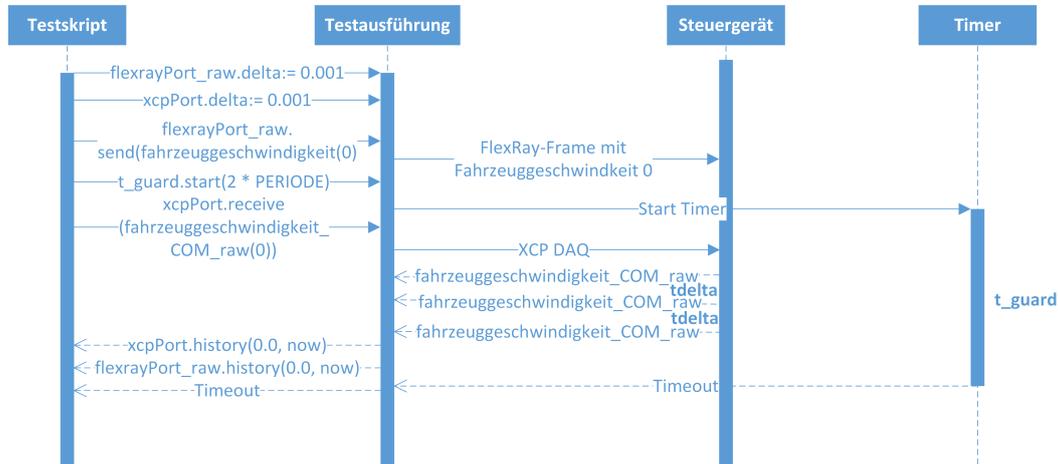
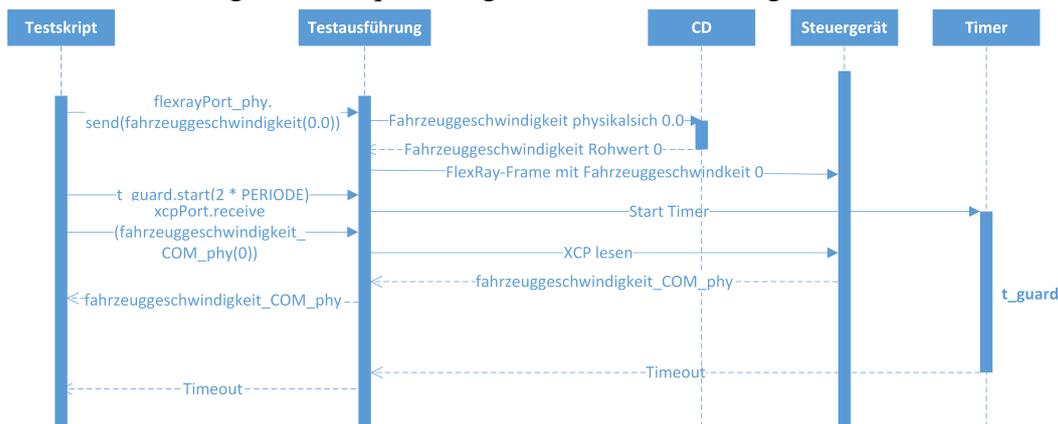


Abbildung 5.16.: Sequenzdiagramm der Ausführung von Testfall 3



5.4.4. Umgesetzte Anforderungen

In diesem Abschnitt werden die Anforderungen, welche in Abschnitt 4.3.2 der Testausführung zugeordnet wurden, einzeln bezüglich ihrer Umsetzung diskutiert.

Anforderung 11 – Verwendung einer Programmiersprache mit umfangreichen Exception-Handling

TTCN-3 sowie Python bieten beide ein Exception-Handling an. Bei der Testausführung durch den TTCN-3 Interpreter werden Fehler innerhalb der Testausführung über Python-Exceptions erkannt und behandelt. Weiterhin bietet TTCN-3 selbst Exceptions an. Diese werden intern ebenfalls auf Python-Exceptions abgebildet. Eine Behandlung auf Ebene des TTCN-3 Codes ist umgesetzt.

Anforderung 12 – Verwendung einer Programmiersprache zur einfache Umsetzung von Testfällen

TTCN-3 biete umfangreiche Mechanismen, um Fehler in den Testfällen zu vermeiden. Dabei sei die Timeout-Überwachung genannt, die für alle Prüfungen einfach verwendbar ist. Des Weiteren verhindert die strikte Prüfung der Daten und Datentypen, Fehler bzw. fehlerhafte Bewertungen in den Testfällen.

Anforderung 13 – Behandlung von Fehlern und Exceptions im Testautomatisierungssystem

Wie schon bei Anforderung 11 ausgeführt, werden Fehler innerhalb der Middleware erkannt und behandelt. Der realisierte TTCN-3 Interpreter erkennt und behandelt die größtmögliche Zahl an Fehlern während der Ausführung.

Anforderung 14 – Bereitstellung von Funktion(en) um das Testsystem in den Default-Zustand zu versetzen

Der Aufruf *unmap*, welcher für jeden Port zu einem DUT bzw. zum Testsystem zu implementieren ist, ermöglicht es am Ende eines jeden Testfalls das System zurück in den Default-Zustand zu versetzen.

Anforderung 15 – Automatische Herstellung des Default-Zustandes nach Abbruch eines Testfalls (z.B. durch unbehandelte Exception)

Der *unmap*-Aufruf wird im Fehlerfall auch durch den TTCN-3 Interpreter ausgeführt. Damit ist sichergestellt, dass das Testsystem in einen definierten Zustand versetzt wird.

Anforderung 16 – Bibliotheken sind unabhängig vom Testautomatisierungssystems lauffähig

Die umgesetzten Bibliotheken, wie zum Beispiel die Port-Implementierung für Gamma V bilden einzelne Python-Skripte bzw. Packages, welche auch getrennt verwendet werden können. Einzig ein Python-Interpreter wird benötigt.

Anforderung 17 – Testfälle sind unabhängig vom Testmanagement des Testautomatisierungssystems lauffähig

Mit dem implementierten TTCN-3 Interpreter ist es möglich die TTCN-3 Testfälle einzeln auszuführen. Das Testmanagement in TTCN-3 besteht ebenfalls aus TTCN-3 Dateien, welche die *testcases* in einer festgelegten Reihenfolge aufrufen. Damit sind TTCN-3 Skripte beliebig ausführbar.

Anforderung 18 – Ein Testfall muss ein eigenständig ausführbares Programm darstellen

Ein TTCN-3 Testskript kann mit Hilfe des TTCN-3 Interpreter getrennt ausgeführt werden.

Anforderung 19 – Bereitstellung von Funktionen zur Bewertung einzelner Testschritte

Mit der Klasse *VerdictValue* bzw. dem Datentyp *verdict* ist in jedem Fall eine definierte Bewertung eines Testfalls bzw. eines Testschrittes gegeben.

Anforderung 20 – Bereitstellung von Funktionen zur Ausgabe der Bewertung einzelner Testschritte in den Test-Report

Mittels der Methode *setVerdict* ist es möglich Bewertungen inklusive textueller Beschreibung in den Test-Report auszugeben. Des Weiteren können über den *log*-Aufruf weitere Informationen in den Test-Report ausgegeben werden.

Anforderung 21 – Erfassung der Laufzeit einzelner Testfälle durch das Testautomatisierungssystem

Die Erfassung der Laufzeit einzelner Testfälle erfolgt direkt durch den TTCN-3 Interpreter bzw. durch die Klasse *TciTLPprovided*, welche in den Methoden *tliTcStarted* und *tliTcStop* die Start- und Endzeit für jeden Testcase erfasst und in den Testbericht ausgibt.

Anforderung 22 – Weitergabe der Laufzeit an den Test-Report

Die Weitergabe der Laufzeit erfolgt durch die Ausgabe der erfassten Zeit in die modTF-XML-Datei für den Test-Report.

Anforderung 23 – Bereitstellung von Funktionen für die Weitergabe von Informationen von der Testausführung zum Test-Reporting

Die Weitergabe der Informationen erfolgt über die Klasse *TciTLPprovided* und die modTF-XML-Datei für den Test-Report.

Anforderung 24 – Berücksichtigung von Informationen für Testbewertung, Testbeschreibungen, Messdaten und Meta-Daten (z. B. Anforderungs-IDs)

Die Klasse *TciTLPprovided* sowie die beschriebenen Möglichkeiten zur Weitergabe von Bildern und binären Daten über die *log*-Anweisung an den Test-Report ermöglichen eine weitreichende Weitergabe von Daten der Testausführung an den Test-Report.

Anforderung 37 – Bereitstellung von Funktionalität zum Ausführen der Testfälle in beliebiger Reihenfolge

Die Ausführungsreihenfolge der Testfälle wird über die Aufrufreihenfolge der Testcases innerhalb von TTCN-3 Skripten geregelt. Die Reihenfolge ist beliebig anpassbar.

Anforderung 46 – Bereitstellung einheitlicher Schnittstellen für Testsysteme

Durch das TRI ist eine einheitliche Schnittstelle zum Testsystem gegeben. Mit der Umsetzung des Packages *ttn3ports* ist die Erweiterung für neue Testsystem ohne Änderung am Interpreter möglich.

Anforderung 47 – Bereitstellung abstrahierter Funktionen für den Testsystemzugriff auf Testfallebene

Der Zugriff auf das Testsystem erfolgt über die Aufrufe *map*, *send*, *receive* und *unmap* im TTCN-3 Code. Diese Aufrufe sind für alle Testsysteme gleich. Eine Unterscheidung wird erst über den Namen im *map*-Aufruf getroffen. Über den gegebenen Portnamen werden dynamisch die notwendigen Bibliotheken (*ttn3ports*) geladen und verwendet.

5.4.5. Anwendung des Moduls

Die Anwendung des Test-Ausführungs-Modules erfolgt über den Aufruf des Python-Skriptes *TTCN3Parser.py*. Dieses Skript erlaubt es über Kommandozeilen-Parameter das gewünschte Verhalten einzustellen. Die möglichen Kommandozeilenooptionen werden bei der Übergabe von *-h* beim Aufruf ausgegeben und sind in Listing 5.14 dargestellt.

Listing 5.14: TTCN-3 Interpreter – Kommandozeilenooptionen

```
Usage: TTCN3Parser.py [options] <files(s) to execute>
```

Options:

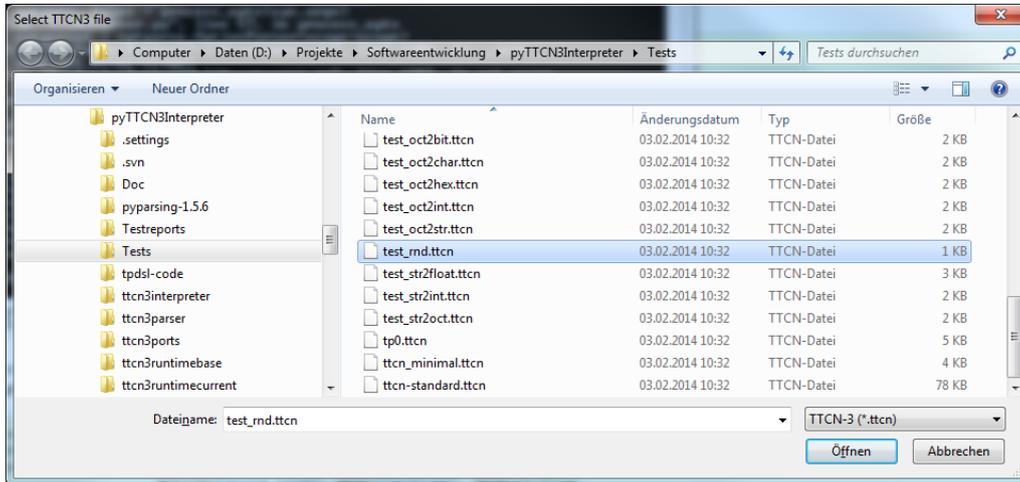
```
-h, --help          show this help message and exit
-d, --debug         debug the parser, developers only
-f, --file_dialog   select file to execute by dialog
-k, --keywords      print list of TTCN3 keywords
-t, --print-tree    print abstract syntax tree
-v, --version       show version of program
```

Für die typische Verwendung aus Eclipse bzw. TRex heraus wird das auszuführende TTCN-3 Skript dem Aufruf mit übergeben.

Um den Interpreter einzeln einzusetzen, ist es möglich über den Kommandozeilenparameter *-f* einen Datei-Dialog aufzurufen, über den das auszuführende Skript ausgewählt werden kann, wie dies in Abbildung 5.17 zu sehen ist. Mit Klick auf den Button *Öffnen* wird das gewählte TTCN-3 Skript ausgeführt. Damit wird automatisch im Verzeichnis

Test-Reports eine modTF-XML-Datei mit dem Namen des TTCN-3 Skriptes abgelegt, welche alle Ausgaben in den Test-Report enthält.

Abbildung 5.17.: Datei-Dialog bei Aufruf des Interpreters



5.5. Testdokumentation

Auf Basis des, in Abschnitt 4.4 erarbeiteten Konzeptes für die Testdokumentation, wird nun eine Softwarearchitektur für das Test-Report-Generator-Modul erarbeitet. Das Hauptaugenmerk liegt dabei auf dem modTF-XML-Format, welches die Schnittstelle zwischen dem Test-Ausführungs-Modul und dem Test-Report-Generator-Modul bildet.

5.5.1. Umsetzung des Lösungskonzeptes in die Softwarearchitektur

Das Lösungskonzept für das Test-Report-Generator-Modul besteht dabei aus drei Komponenten, welche zusammenwirken müssen und die im Folgenden aufgelistete sind:

- Die *TL*-Komponente, welche die modTF-XML-Datei erstellt
- Die modTF-XML-Datei-Struktur, welche die Daten speichert
- Das Test-Report-Generator-Modul, welches die modTF-XML-Dateien verarbeitet

Die *TL*-Komponente ist in Abschnitt 5.4.2 beschrieben.

Die modTF-XML-Datei-Struktur ist in Form des XML-Schemas in Abschnitt 4.4.4 beschrieben und soll in diesem Abschnitt nur aus Sicht der Implementierung nochmals betrachtet werden.

Das Test-Report-Generator-Modul soll einen möglichst flexiblen Umgang mit dem modTF-XML-Dateien ermöglichen. Dazu ist es notwendig eine modulare Struktur zu wählen. Als Kernkomponente ist ein XML-Parser zu erstellen, der die modTF-XML-Dateien einlesen und intern im Speicher zur Verfügung stellen kann. Dabei sollte diese Komponente ebenfalls Metriken für die Bewertungen ermöglichen. Basierend auf diesem Parser können nun verschiedene Generatoren für Test-Reports geschaffen werden, welche die Datenstruktur im Speicher in das jeweilige Dateiformat umwandeln. Die Umsetzung soll weiterhin in Python stattfinden.

5.5.2. Implementierung der Softwarearchitektur

Das Test-Report-Generator-Modul besteht aus einer Sammlung von Python-Package. Die Basis bildet dabei das Package *isxml*, dessen Elemente in Tabelle 5.4 aufgelistet sind.

Auf Basis dieses Packages sind verschiedene Test-Report-Generatoren in Form eigener Python-Skripte implementiert worden. Im Folgenden soll hier auf die Erzeugung eines HTML-Test-Reports und auf die Anbindung des Requirementsmanagement-System MKS eingegangen werden.

Der HTML-Test-Report-Generator ist in Form des Python-Skriptes *html_report_generator.py* implementiert und beinhaltet die Klasse *HtmlReportGenerator*,

Tabelle 5.4.: Package *isyxml*

Dateiname	Beschreibung
isyparser.py	Implementierung des XML-Parsers auf Basis eines Tree-Walkers
isystatistics.py	Implementierung der Statistikfunktion zur Erstellung von Metriken
parser_utils.py	Hilfsfunktionen für den XML-Parser

welche den eigentlichen Generator implementiert. Der eigentliche Generator-Prozess wird durch Aufruf der Methode *generateReport* realisiert. In Listing 5.15 ist der erste Teil einer modTF-XML-Datei zu sehen. Der daraus generierte Teil des HTML-Test-Reports ist in Abbildung 5.18 zu sehen. Dabei ist zu beachten, dass die angezeigten Übersichten – *Overview* – durch die Statistikfunktionen erzeugt wurden und kein direkter Bestandteil der XML-Datei sind. Bei der Gegenüberstellung von Listing 5.16 und Abbildung 5.19 ist hingegen ein 1:1-Mapping der Informationen aus der XML-Datei in den Test-Report zu erkennen.

Abbildung 5.18.: Ausschnitt aus einer HTML-Test-Report Teil1

Test Report: test_bit2hex.ttcn

description

Created on: 2014-02-06T16:20:24
Created by: author
Department: department
Customer: customer
Project: ProTecT
SW-Version: softwareversion
HW-Version: hardwareversion

Overview**Testcase Overview**

Testcase-ID	Source Tests	PASSED	FAILED	PASSED [%]
None	test_bit2hex_1 test_bit2hex_2 test_bit2hex_3	3	0	100.0 %
Σ 1 TCs	Overall	3	0	100.0 %

Sequence Overview

Test Sequence	PASSED	FAILED	PASSED [%]	Runtime [hh:mm:ss]
title of testsequence	3	0	100.0 %	n/a
Overall	3	0	100.0 %	n/a

Single Test Overview

title of testsequence

Single Test	PASSED	FAILED	PASSED [%]	Runtime [hh:mm:ss]
test_bit2hex_1	1	0	100.0 %	n/a
test_bit2hex_2	1	0	100.0 %	n/a
test_bit2hex_3	1	0	100.0 %	n/a
Overall	3	0	100.0 %	n/a

Overall Result

	PASSED	FAILED	PASSED [%]	Runtime [hh:mm:ss]
Overall	3	0	100.0 %	n/a

Listing 5.15: Ausschnitt aus einer modTF-XML-Datei Teil1

```

<?xml version="1.0" encoding="UTF-8"?>
<iTest:testreport report_version="0.6" source="
  xmlns:iTest="http://www.isyst.de/iTest.xsd" xmlns:xsi="
  http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.isyst.de/iTest.xsd_iTest
  .xsd">
  <meta>
    <title>test_bit2hex.ttcn</title>
    <description>description</description>
    <author>author</author>
    <date>2014-2-6T16:20:12.000Z</date>
    <version>version</version>
  
```

```

<last_revision>last_revision</last_revision>
<hardwareversion>hardwareversion</hardwareversion>
<softwareversion>softwareversion</softwareversion>
<customer>customer</customer>
<project>ProTecT</project>
<department>department</department>
</meta>

```

Abbildung 5.19.: Ausschnitt aus einer HTML-Test-Report Teil2

Sequence: title of testsequence

test squence

Last Revision: last_revision
Version: version
Revision Date: 2014-2-6T16:20:12.000Z
Author: author

[↑ back to overview](#)

Test: test_bit2hex_1

Test Source File: test_bit2hex_1
Tested on: 2014-2-6T16:20:12.000Z
Last Revision:
Version: Kein
Revision Date: 2014-2-6T16:20:12.000Z
Author:
Testcase-IDs: None

[↑ back to overview](#)

Test Steps:

#	No.		TC-ID	Verdict
1	1	"Test bit2hex 1"	None	INFO
2	2	result: 5	None	INFO
3	3	correct value	None	PASSED
4	4		None	INFO

Single Test Result

	PASSED	FAILED	PASSED [%]
test_bit2hex_1	1	0	100.0 %

[↑ back to overview](#)

Listing 5.16: Ausschnitt aus einer modTF-XML-Datei Teil2

```

<singletest source="test_bit2hex_1">
  <meta>
    <title>test_bit2hex_1</title>
    <description></description>
    <author></author>
    <date>2014-2-6T16:20:12.000Z</date>
    <version>Kein</version>
    <last_revision></last_revision>
    <starttime>2014-2-6T16:20:12.000Z</starttime>
    <endtime>2014-2-6T16:20:12.000Z</endtime>

```

```

</meta>
<results>
  <result>
    <description>&quot;Test bit2hex 1&quot;</
      description>
    <verdict>INFO</ verdict>
    <testcase_id>None</ testcase_id>
  </result>
  <result>
    <description>result: 5</ description>
    <verdict>INFO</ verdict>
    <testcase_id>None</ testcase_id>
  </result>
  <result>
    <description>correct value</ description>
    <verdict>PASSED</ verdict>
    <testcase_id>None</ testcase_id>
  </result>
  <result>
    <description></ description>
    <verdict>INFO</ verdict>
    <testcase_id>None</ testcase_id>
  </result>
</ results>
</ singletest>

```

Eine weitere Implementierung eines Test-Report-Generators stellt die Anbindung an das Requirementsmanagement-System MKS dar und ist als Python-Skript realisiert. Dieser Generator ist durch die Klasse *MksReportGenerator* implementiert, welcher ebenfalls die Methode *generateReport* zur Reporterzeugung zur Verfügung stellt. Für die Erzeugung der notwendigen Daten – die Zuordnung der Bewertungen zu den Anforderungs-IDs – kommen die Statistikfunktionen zum Einsatz. Das Ansprechen von MKS erfolgt über das Package *isymks*, welches in der Datei *mks_connect_utils.py* die in Abschnitt 4.5.3.2 dargestellte Schnittstelle implementiert. In Listing 5.17 ist die Methode *setVerdict* beispielhaft dargestellt. Für die kompakte Darstellung wurde die Fehlerbehandlung entfernt. Die direkte Kommunikation mit dem MKS System erfolgt über Aufrufe auf der Kommandozeile. Dies ist in der Methode *runMksCmd* gekapselt, um eine einheitliche Fehlerbehandlung realisieren zu können.

Listing 5.17: Ausschnitt der Methode *setVerdict* der MKS-Anbindung

```
1 def setVerdict(self, testcase_id, mks_user_name,
2     testcase_value):
3     # Kommando ausfuehren und Ergebnis holen
4     cmd_result_string = self.runMksCmd("C:\\", mks_cmd)
5
6     # auf Fehler pruefen
7     if cmd_result_string.find("Updated_item") != -1:
8         # alles ok
9         return_list[0] = self.ERROR_NO_ERROR
10        return_list[1] = cmd_result_string
11    else:
12        errorPassword = cmd_result_string.find("must_use_
13            the_option_--password")
14        if errorPassword != -1:
15            # kein Login
16            return_list[0] = self.ERROR_NO_LOGIN
17            return_list[1] = "Error_-_No_Login"
18        else:
19            # Fehler
20            return_list[0] = self.ERROR_NO_UPDATE_TESTCASE
21            return_list[1] = cmd_result_string
22    return return_list
```

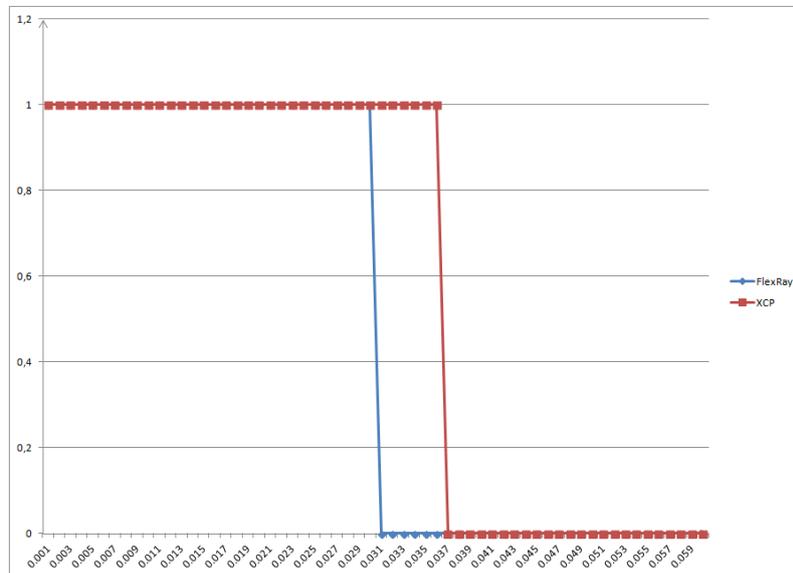
Mit den erstellten Packages ist eine flexible Basis für die Implementierung verschiedener Test-Report-Generatoren geschaffen worden. Weiterhin konnte auch die durchgängige Anbindung an das Requirementsmanagement dargestellt werden.

5.5.3. Anwendung auf Beispiel-Testfälle

Die in den Abschnitten 5.3.4 und 5.4.3 implementieren und ausgeführten Testfälle, führen in der dargestellten Form zu genau zwei Ausgaben im Testreport. Dieses sind die Testcase-ID und die Bewertung des Tests. Wegen der Übersichtlichkeit wurden in den Beispielen keine weiteren Ausgaben gemacht.

Die Implementierung von Testfall 2, unter Verwendung der Erweiterung *Support of interfaces with continuous signals*, ermöglicht es die Signalverläufe mit Hilfe des modTF-Testreportings grafisch darzustellen, wie dies in Abbildung 5.20 zu sehen ist. Die direkte Einbettung dieser grafischen Darstellung als Bild in den Testreport, wie dies in Abbildung 5.21 dargestellt ist, ist ebenfalls möglich.

Abbildung 5.20.: Darstellung der Signalverläufe auf FlexRay und XCP



5.5.4. Umgesetzte Anforderungen

In diesem Abschnitt werden die Anforderungen, welche in Abschnitt 4.4.2 der Testdokumentation zugeordnet wurden, einzeln bezüglich ihrer Umsetzung diskutiert.

Anforderung 25 – Bereitstellung verschiedener Gliederungsebenen im Test-Report

Das modTF-XML-Format stellt mit den Elementen *testsequences*, *singletests* und *results* eine dreistufige Gliederung der Test-Reports zur Verfügung. Diese Gliederung wird beispielsweise in dem HTML-Test-Report abgebildet.

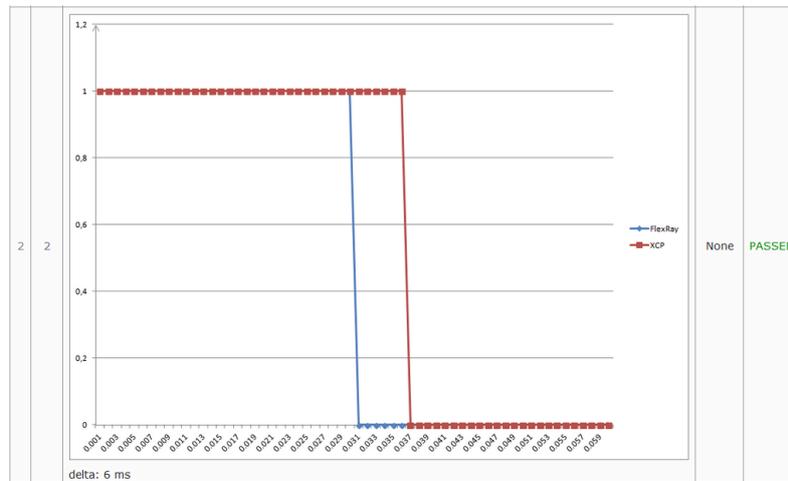
Anforderung 26 – Bereitstellung von Beschreibungen auf den verschiedenen Gliederungsebenen im Test-Report

Auf allen Gliederungsebenen können Beschreibungen in Form von Meta-Dateien angegeben werden. Auf der Ebene der *result*-Elemente ist die Verwendung von *description*-Elementen möglich, welche ein Testergebnis genauer beschreiben.

Anforderung 27 – Bereitstellung von Bewertungen zu den Beschreibungen auf allen Gliederungsebenen

Eine Zuordnung der Bewertung zu Beschreibungen erfolgt in Form der *result*-Elemente, welche die Bewertung (*verdict*) und die Beschreibung (*description*) enthalten.

Abbildung 5.21.: Testreport mit Darstellung der Signalverläufe auf FlexRay und XCP



Anforderung 28 – Bereitstellung eines Bewertungsschemas mit Priorisierung (z. B. Error, Failed, Passed und Info)

Die Priorisierung der Bewertungen erfolgt an zwei Stellen. Als erstes stellt TTCN-3 mit dem *VerdictValue* als eigener Datentyp bereits eine Priorisierung [112, S. 49] zur Verfügung, welche innerhalb der Klasse *VerdictValue* im TTCN-3 Interpreter implementiert ist.

Die Statistikfunktionen des Test-Report-Generator-Moduls implementieren eine entsprechende Priorisierung, da dies für den Abgleich mit dem Requirementsmanagement notwendig ist. Weiterhin soll das Test-Report-Generator-Modul auch eigenständig einsetzbar sein, so dass nicht davon ausgegangen werden kann, dass eine Priorisierung der Bewertungen schon bei der Erstellung der XML-Datei stattgefunden hat.

Anforderung 29 – Bereitstellung einer erweiterbaren Schnittstelle für Meta-Daten

Auf allen Gliederungsebenen stehen Meta-Daten in Form der *meta*-Elemente zur Verfügung. Diese können beliebig erweitert werden. Der XML-Parser des Test-Report-Generator-Moduls, bildet alle gefundenen Meta-Daten mit ihrem Bezeichner in die Datenstruktur für die weitere Verarbeitung ab. Einzig der jeweilige Test-Report-Generator muss die erweiterten Meta-Daten auswerten.

Anforderung 30 – Speicherung der Meta-Daten im Testbericht

Die Meta-Daten sind in der modTF-XML-Datei gespeichert. Je nach Anforderung können die Daten zum Beispiel in den HTML-Test-Report übernommen werden, wie dies in Abschnitt 5.5.2 veranschaulicht ist.

Anforderung 31 – Bereitstellung eines Speicherformates für das Test-Reporting, welches maschinell verarbeitbar ist

Mit dem entwickelten und umgesetzten modTF-XML-Format ist ein maschinell verarbeitbares Speicherformat für Test-Reports realisiert worden.

Anforderung 32 – Bereitstellung eines Speicherformates für das Test-Reporting, welches manuell ergänzbar ist

Das modTF-XML-Format ist ebenfalls manuell mit Hilfe eines Texteditors oder eines XML-Editors ergänzbar.

Anforderung 33 – Bereitstellung einer Umwandlung des maschinell verarbeitbaren Test-Report in Menschen lesbarer Form

Mit der Generierung von HTML-Test-Reports, wie dies im Abschnitt 5.5.2 erläutert wurde, ist die Umwandlung des modTF-XML-Format in eine Menschen lesbare Form verdeutlicht worden.

Anforderung 34 – Unterstützung verschiedener Formate, wie zum Beispiel PDF, HTML und XLS, für den Test-Report

Im Rahmen dieses Projektes konnte eine Basis für Test-Report-Generatoren geschaffen werden. Mit der dargestellten Implementierung eines HTML-Report-Generators konnten die Möglichkeiten verdeutlicht werden. Eine Umsetzung nach XLS oder PDF konnte innerhalb dieser Arbeit nicht realisiert werden.

Anforderung 35 – Bereitstellung von Funktionen zur Erstellung von Metriken

Mit Hilfe der bereitgestellten Statistikfunktionen können verschiedene Metriken erzeugt werden. Eine Auswahl ist in Abbildung 5.18 zu sehen.

Anforderung 36 – Bereitstellung von Funktionen zur Konfiguration der Metriken

Die bereitgestellten Statistikfunktionen stellen Daten für verschiedene Metriken zur Verfügung. Es ist auf Basis der Daten des XML-Parsers auch möglich weitere Metriken zu erzeugen. Die Ausgabe dieser Metriken in Test-Reports kann beliebig angepasst werden.

Anforderung 39 – Speicherung der Ausführungsreihenfolge der Testfälle

Mit der Speicherung der modTF-XML-Dateien und der darin enthaltenen Gliederung ist auch die Ausführungsreihenfolge der einzelnen Tests gespeichert.

Anforderung 44 – Bereitstellung von Funktionen zum Abgleich von Testergebnissen mit Anforderungen

Mit der Implementierung des MKS-Test-Report-Generators konnte der Abgleich der Testergebnisse mit den Anforderungen im Requirementsmanagement-System gezeigt werden.

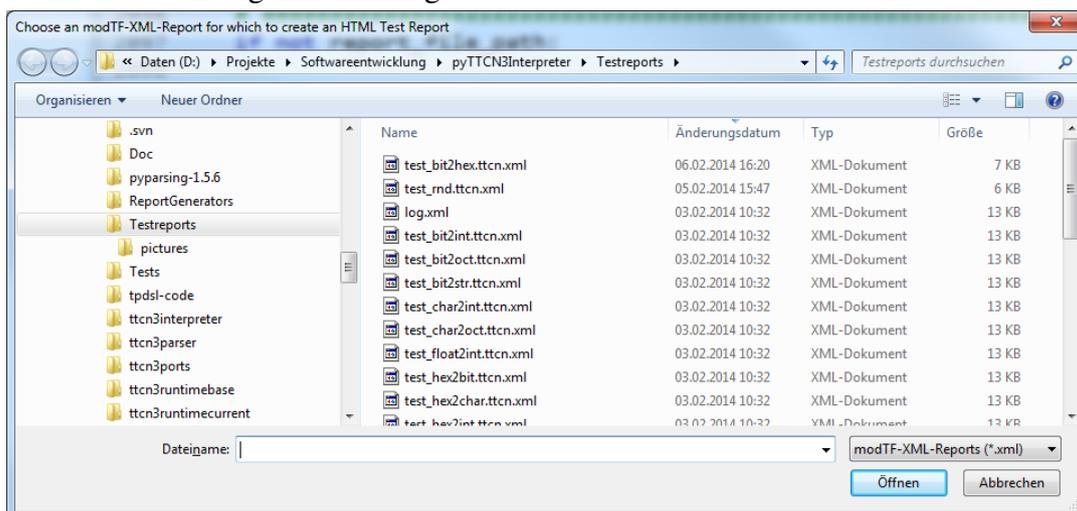
Anforderung 45 – Bereitstellung von Funktionen zum Abgleich von Beschreibungen (z. B. Testfallbeschreibung)

Die Bibliothek zur MKS-Anbindung stellt die Methode *getDescription* zur Verfügung, welche die Beschreibung des MKS-Elements zurück liefert. Dies wird bisher für die Generierung des Test-Reports im Rahmen des modTF nicht verwendet.

5.5.5. Anwendung des Moduls

Der Aufruf des jeweiligen Test-Report-Generators erfolgt über die Ausführung des zugehörigen Python-Skripts. Diesem Aufruf kann direkt die modTF-XML-Datei übergeben werden, welche für die Generierung des Test-Reports verwendet werden soll. Wird dem Aufruf kein Parameter übergeben, so wird ein Datei-Dialog für die Auswahl der modTF-XML-Datei angezeigt, wie er in Abbildung 5.22 zu sehen ist.

Abbildung 5.22.: Dialog für die Auswahl der modTF-XML-Datei



5.6. Integration in den Entwicklungsprozess

Auf Basis des in Abschnitt 4.5 erarbeiteten Konzeptes für die Integration des modTF in den Entwicklungsprozess, wird dies nun in der Softwarearchitektur umgesetzt.

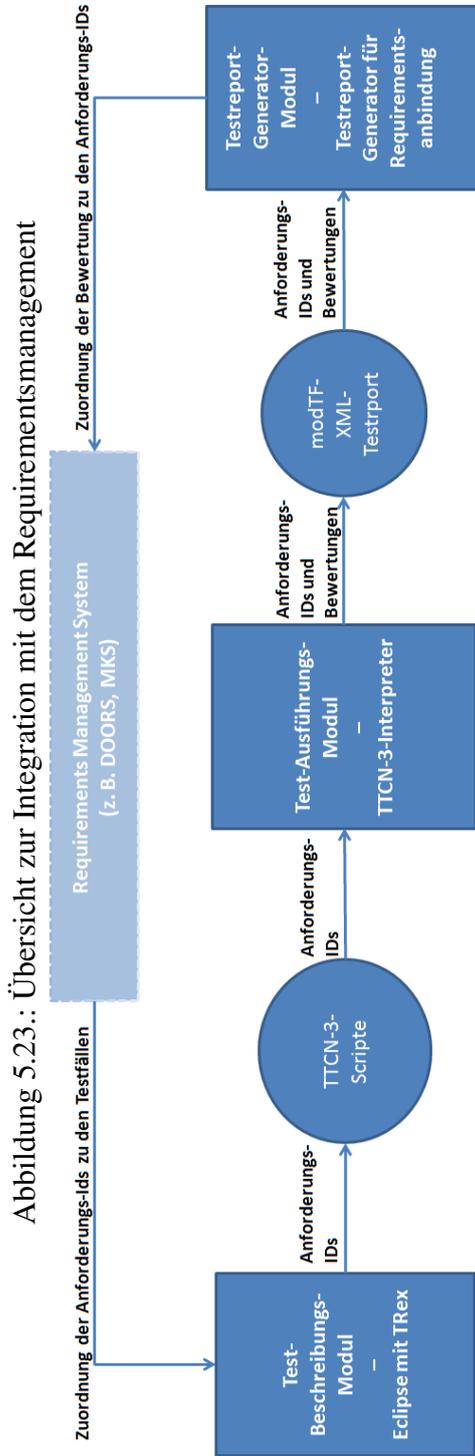
5.6.1. Umsetzung des Lösungskonzeptes

Wie in Kapitel 4.5.3 bereits erläutert, besteht die Integration des modTF in den Entwicklungsprozess aus zwei Teilen: die Integration mit dem Versionsmanagement und die Integration mit dem Requirementsmanagement.

Die Umsetzung dieser Integration mit dem Requirementsmanagement ist bereits in den Kapiteln 5.3 und 5.5 erläutert wurden. An dieser Stelle soll das Zusammenspiel über die einzelnen Module des modTF dargestellt werden. In Abbildung 5.23 ist der Informationsfluss zwischen Requirementsmanagement und dem modTF-System dargestellt. Die Verknüpfung der Anforderungen mit den Testfällen erfolgt über die eindeutige ID der jeweiligen Anforderung. Dies muss meist manuell erfolgen. Alternativ wäre auch eine automatische Generierung von leeren Testfällen auf Basis der Anforderungsstruktur im Requirementsmanagement-System möglich. Die Anforderungs-IDs werden in den TTCN-3 Skripten gespeichert und bei Testausführung zusammen mit den Bewertungen der Testfälle in den modTF-XML-Test-Report gespeichert. Mit Hilfe der Funktionen und Packages des Test-Report-Generator-Moduls, wurde es möglich, diese Informationen automatisiert an das Requirementsmanagement-System weiter zugeben und weiterhin auch zusätzliche Beschreibungen in den Test-Report zu integrieren.

Die Umsetzung der Integration mit dem Versionsmanagement ist entsprechend des Konzeptes direkt realisiert worden. Mit der Verwendung von TTCN-3 für die Testbeschreibung ist eine direkte Versionierung der TTCN-3 Skripte möglich. Es setzt, wie die meisten anderen Programmiersprachen, ein textbasiertes Dateiformat ein. Damit ist mit den Standard-Funktionen eines Versionsmanagement-System die Versionierung sowie der Vergleich verschiedener Dateiversionen möglich. Es sind alle Funktionen realisierbar, wie sie auch für die Entwicklung des Codes für die eigentliche Software zum Einsatz kommen.

Die Anbindung des Test-Reports in Form des modTF-XML-Test-Reports an das Versionsmanagement ist ebenfalls ohne zusätzlichen Aufwand möglich. Die textbasierten XML-Dateien lassen sich ebenfalls mit den Standard-Funktionen in einem Versionsmanagement-System verwalten.



5.6.2. Umgesetzte Anforderungen

In diesem Abschnitt werden die Anforderungen, welche in Abschnitt 4.5 der Integration in den Entwicklungsprozess zugeordnet wurden, einzeln bezüglich ihrer Umsetzung diskutiert.

Anforderung 40 – Bereitstellung einer Schnittstelle zum Versionsmanagement-System

Die Schnittstelle zum Versionsmanagement stellen die beiden Dateiformate TTCN-3 und modTF-XML dar. Diese lassen sich ohne weiteren Integrationsaufwand in jedem Versionsmanagement-System verwalten.

Anforderung 42 – Bereitstellung einer bidirektionalen Schnittstelle zu Requirementsmanagement-Systemen

Mit der konzeptionierten und für MKS umgesetzten Schnittstelle zu Requirementsmanagement-Systemen, wie sie in Kapitel 4.5.3.2 dargestellt ist, ist einerseits die Verknüpfung der Anforderungen über Anforderungs-IDs mit den Testfällen möglich und andererseits können die Testergebnisse automatisiert zurück gespielt werden. Eine Erweiterung des Testberichts mit der Beschreibung der Anforderungen ist ebenso möglich.

Anforderung 43 – Bereitstellung von Funktionen zur Verknüpfung von Testfällen mit Anforderungen (z.B. Anforderungs-ID)

Die Verknüpfung der Anforderungen muss noch manuell erfolgen. Alternativ wäre auch eine automatische Generierung von leeren Testfällen auf Basis der Anforderungsstruktur im Requirementsmanagement-System möglich.

Anforderung 44 – Bereitstellung von Funktionen zum Abgleich von Testergebnissen mit Anforderungen

Mit der Methode *SetVerdict()* kann das Ergebnis eines Testfalls automatisiert der Anforderung zugeordnet werden.

Anforderung 45 – Bereitstellung von Funktionen zum Abgleich von Beschreibungen (z. B. Testfallbeschreibung)

Mit der Methode *GetDescription()* ist es möglich die Beschreibung zu einer Anforderungs-ID aus dem Requirementsmanagement-System auszulesen und zum Beispiel in den modTF-XML-Test-Report zu integrieren.

6. Validierung und Evaluierung

In diesem Kapitel werden die Funktionen und die Anwendungsmöglichkeiten des modularen Testautomatisierungs-Frameworks (modTF) anhand von praktischen Beispielen dargestellt. Dabei wird der Einsatz einzelner Module sowie des gesamten modTF als Testautomatisierungslösung gezeigt.

6.1. Validierung des Interpreters

Für die Validierung des Interpreters konnte in der späteren Entwicklungsphase – nachdem der Parser und der Interpreter zur Ausführung von TTCN-3 Testskripten in der Lage waren – der TTCN-3 Interpreter selbst eingesetzt werden. Die Validierung der Built-in-Funktionen von TTCN-3 wurde mit Hilfe von TTCN-3 Testskripten selbst durchgeführt. In Kapitel 5.5.2 ist der Test der Software-Funktion *bin2hex* in Ausschnitten dargestellt. Das zugehörige TTCN-3 Testskript ist in Listing 6.1 auszugsweise zu sehen. Wie dort zu sehen ist, können mittels TTCN-3 sehr kompakte Tests definiert werden. Mit dem dargestellten Schema wurden alle Built-in-Funktionen entsprechend der Vorgaben des Standards für jede einzelne Funktion validiert.

Listing 6.1: TTCN-3 Testskript für die Funktion *bin2hex*

```
1 module helloTTCN3 {
2     testcase test_bit2hex_1 () runs on MITC {
3         var hexstring result;
4         log("Test_bit2hex_1");
5         result := bit2hex('0101'B);
6         log(result);
7         if(result == '5'H) {
8             setverdict(pass, "correct_value")
9         }
10        else {
11            setverdict(fail, "incorrect_value")
12        }
13    } control {
14        execute(test_bit2hex_1());
15    }
16 }
```

6.2. Hardware In The Loop-Anwendung

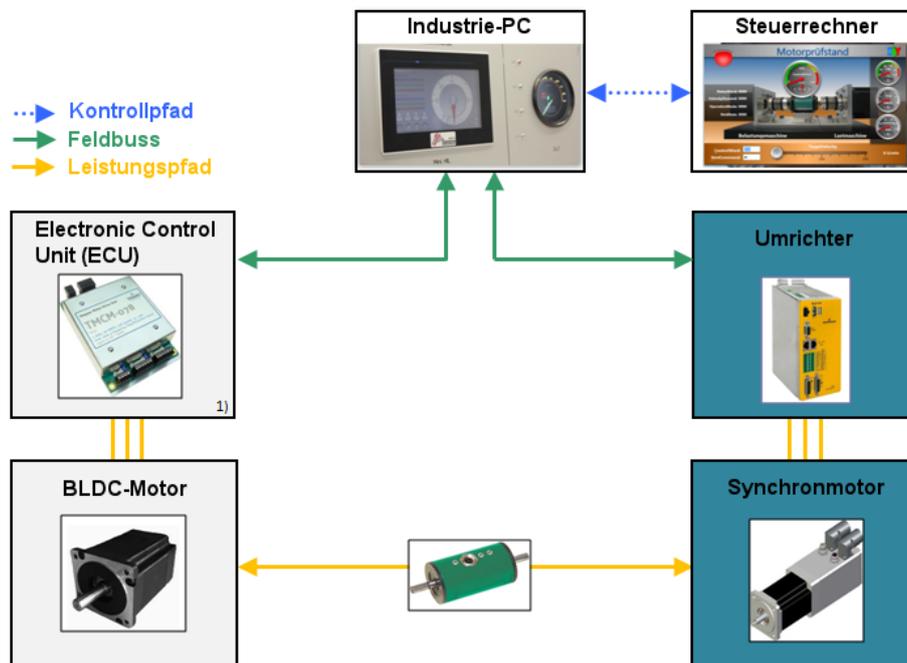
In diesem Abschnitt wird die Verwendung des modularen Testautomatisierungs-Frameworks (modTF) als komplette Testautomatisierungslösung für den Software-Test einer ECU mittels eines Hardware In The Loop-Systemes erläutert.

6.2.1. Gamma V-basierte Hardware In The Loop-Systeme

6.2.1.1. Der Systemaufbau

Entsprechend der allgemeinen Darstellung eines HIL-Systems in Abbildung 2.18, besteht das hier dargestellte System aus einem Steuerrechner mit Windows 7 als Betriebssystem, sowie einem Echtzeitsystem in Form eines Industrie-PC vom Typ Blue Power [123] mit ARM11 Prozessor der Firma TQ Systems mit dem Betriebssystem Linux. Der Systemaufbau ist in Abbildung 6.1 zu sehen. Das DUT bildet dabei die ECU mit dem angeschlossenen Elektromotor (BLDC-Motor).

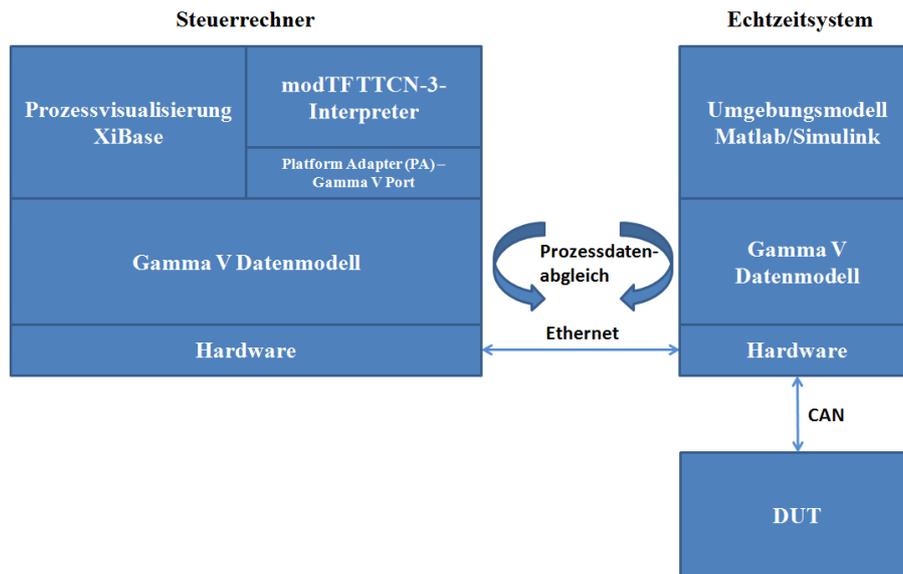
Abbildung 6.1.: Systemaufbau HIL-Demosystem



Die echtzeitfähige Ausführung des Umgebungsmodells sowie die Anbindung der eigentlichen IOs erfolgt über die Echtzeit-Middleware Gamma V [124] der Firma RST. Eine genauere Beschreibung der Funktionen der Middleware ist unter [125] zu finden. Neben dem Gamma V-Service auf dem Echtzeitsystem wird auch ein Gamma V-Service auf dem

Steuerrechner ausgeführt. Über die Ethernet-Verbindung zwischen den beiden Systemen werden die Daten des Datenmodells der Middleware ausgetauscht. Die sich ergebende Softwarestruktur ist in Abbildung 6.2 dargestellt. Auf dem Steuerrechner wird neben dem modTF-System auch die Software XiBase9 der Firma XiSys für die Prozessvisualisierung ausgeführt.

Abbildung 6.2.: Softwarestruktur HIL-Demosystem



6.2.1.2. Anbindung der Echtzeit-Middleware Gamma V

Für die Anbindung von Gamma V war die Implementierung eines Ports in Form eines Python-Skripts im Package *ttn3ports* notwendig. Es wurde dabei auf die Python-Schnittstelle der Middleware zurück gegriffen. In Listing 6.2 ist beispielhaft die Implementierung der Methode *send* aufgezeigt. Diese greift über die Klasse *gaapi* auf den Gamma V-Service zu. In den Zeilen 11 und 12 wird die Formatierung des Namens der zu schreibenden Variablen in Gamma V-Datenmodell angepasst. In Zeile 13 wird über die Klasse *gaapi* die Variable mittels des Namens geöffnet und in der folgenden Zeile wird der übergebene Wert in die Variable geschrieben. Abschließend wird diese geschlossen und der Rückgabewert der Funktion auf *TRI_OK* gesetzt. Damit ist das Schreiben einer Variablen innerhalb der Middleware Gamma V aus TTCN-3 heraus abgeschlossen.

Listing 6.2: Gamma V Port – *send*

```
1  def send(self, paramList, sutAddress, sendMessage):
2      '''
3      Implementation for the send command of the
4      TriCommunicationSA
5      '''
6      return_value = TriStatusbase(TriStatusbase.
7      TRI_ERROR)
8      if not self._gamma_sevice_attached:
9          return return_value
10
11     # set PV
12     sutAddress = sutAddress.strip("\")
13     sutAddress = sutAddress.strip("\'")
14     pv = gaapi.PV(sutAddress)
15     pv.value = sendMessage
16     pv = None
17
18     return_value.setStatus(TriStatusbase.TRI_OK)
19
20     return return_value
```

6.2.1.3. Die Testumsetzung und Testausführung

Auf Basis der Anbindung für Gamma V, konnten nun Testfälle in TTCN-3 unter Verwendung von Eclipse und TRex auf dem Steuerrechner manuell erstellt werden. In Listing 6.3 ist ein entsprechendes Testskript in gekürzter Form dargestellt. Im ersten Teil des Skriptes wird mit der Definition des Ports auf *gammaV*, die Verknüpfung zur Port-Implementierung des Interpreters hergestellt. Im Folgenden werden zwei Variablen vom Typ *gammaValue* angelegt, welche Zieldrehzahl (*speedRequest*) und die Istdrehzahl (*speedMeasurement*) des DUT abbilden. Innerhalb des eigentlichen Testcase wird mit *map* die Verbindung zum Gamma V-Service hergestellt. Danach wird die Zieldrehzahl eingestellt und die erreichte Istdrehzahl geprüft. Am Ende wird die Verbindung zum Gamma V-Service mit *unmap* beendet.

Listing 6.3: TTCN-3 Testskript – Motortest

```

1 module motorTest {
2   type component MTC { port gammaV testsystemPort; }
3   template gammaValue speedRequest (VarValue p_varValue
4     ) := {
5     varPath := ":Data.Memory.Bmaxx3400.TargetVelocity"
6     ,
7     varValue := p_varValue }
8   template gammaValue speedMeasurement := {
9     varPath := ":Data.Memory.Bmaxx3400.VelocityDemand"
10    ,
11    varValue := omit }
12 testcase check_motor_speed () runs on MTC {
13   log (" [[TC-ID]]12345[[_TC-ID]]");
14   map(self:testsystemPort , system:gammaV);
15   /* *** Set target speed 1000rpm*** */
16   testsystemPort.send(speedRequest(target_speed));
17   t_guard.start(0.1);
18   alt {
19     [] testsystemPort.receive(speedMeasurement) ->
20     value receive_value {
21       t_guard.stop;
22       if ((receive_value.varValue >
23         target_speed_low) and (
24         receive_value.varValue <
25         target_speed_high)) {
26         setverdict(pass, "Target_speed_
27         reached"); }
28       else {
29         setverdict(fail, "Target_speed_out
30         of_range"); } };
31     [] testsystemPort.receive {
32       t_guard.stop;
33       setverdict(fail, "Invalid_response");
34     };
35     [] t_guard.timeout { setverdict(fail, "Timeout
36     "); };
37   }
38   /* *** Set target speed 0rpm*** */
39   testsystemPort.send(speedRequest(0))
40   unmap(MTC:testsystemPort , system:gammaVPort);
41   stop; }
42 }

```

Mittels des modTF kann dieser Test auf dem Steuerrechner ausgeführt werden. Bei der Testdurchführung wird der modTF-XML-Test-Report erstellt, welcher danach zur Generierung eines HTML-Test-Reports verwendet werden kann. In Abbildung 6.3 ist ein Ausschnitt aus dem generierten HTML-Test-Report zu sehen. Dabei sind die ausgeführten Abschnitte des in Listing 6.3 dargestellten Testskriptes klar wieder zu erkennen.

Abbildung 6.3.: HTML-Test-Report – Motortest

Test: check_motor_speed

Test Source File: check_motor_speed
Tested on: 2013-6-26T11:7:25.000Z
Last Revision:
Version: Kein
Revision Date: 2013-6-26T11:7:25.000Z
Author:
Testcase-IDs: 12345

[↑ back to overview](#)

Test Steps:

#	No.		TC-ID	Verdict
1	1	"[[TC-ID]]12345[[_TC-ID]]"	12345	INFO
2	2	receive_value: Template: varPath:":Data.Memory.Bmaxx3400.VelocityDemand" varValue:998	12345	INFO
3	3	"Target speed reached"	12345	PASSED
4	4	"pass"	12345	INFO
5	5	End of the testcase	12345	INFO

Single Test Result

	PASSED	FAILED	PASSED [%]
check_motor_speed	1	0	100.0 %

[↑ back to overview](#)

6.2.1.4. Ergebnisse

In den vorangegangenen Abschnitten konnte die durchgehende Testautomatisierung mit dem modTF-System für ein HIL-System gezeigt werden. Mit der Anbindung des Echtzeit-Middleware Gamma V konnte die modulare Erweiterbarkeit des modTF gezeigt und erprobt werden. Weiterführende Beschreibungen zur Anwendung des modTF sind den Arbeiten [126] und [127] zu finden.

6.3. Anwendung der Einzelmodule

In diesem Abschnitt wird die Verwendung der einzelnen Module des modularen Testautomatisierungs-Frameworks (modTF) auf getrennten Systemen für den Test gezeigt. Damit soll die Funktionsweise und reale Umsetzung des modularen Ansatzes verdeutlicht werden.

6.3.1. Verteilte Testerstellung und Testausführung

Das in Abschnitt 6.2.1.1 dargestellte Testsystem kann auch ohne Steuerrechner eingesetzt werden. Die betriebssystemunabhängige Implementierung des modTF ermöglicht die Ausführung der TTCN-3 Testskripte direkt auf dem Linux des Echtzeitsystems. Dabei kann der gleiche Port zur Anbindung des Gamma V eingesetzt werden, wie auf dem Steuerrechner unter Windows. Die Ergebnisse der Ausführung sind dabei identisch.

Mit dieser Anwendung konnte gezeigt werden, dass auch der getrennte Einsatz der Module des modTF möglich ist. Die Testerstellung mittels TRex wurde hier auf einen Windows-PC durchgeführt. Die TTCN-3 Skripte wurden mittels Netzwerk auf dem Echtzeitrechner übertragen. Dort wurde die Testausführung mit Hilfe des Test-Ausführungs-Moduls von modTF durchgeführt.

6.3.2. Anbindung der Testdokumentation an kommerzielle Testautomatisierungssysteme

In diesem Abschnitt wird die getrennte Verwendung des Test-Reporting-Moduls sowie die Anbindung an das Requirementsmanagement-System MKS in einem realen Projekt dargestellt. Als Testautomatisierung kam AutomationDesk von dSPACE zum Einsatz, welches vor allem im Bereich des Test-Reports Schwächen zeigte. Die Tests wurden mit einem HIL-System der Firma iSyst auf Basis eines dSPACE-Echtzeitsystems durchgeführt. Das DUT war ein Automobil-Steuergerät aus dem Bereich elektronische Fahrwerksregelung.

6.3.2.1. AutomationDesk Integration

Für die Anbindung von AutomationDesk war es notwendig die generierten Test-Reports des AutomationDesk einzulesen und in ein modTF-XML umzuwandeln. Mittels einer in Python geschriebenen Applikation wurde die Möglichkeit geschaffen, die Projektdateien des AutomationDesk einzulesen, daraus die Pfade und Dateinamen der Testberichte auszulesen und diese in ein modTF-XML zu überführen. Als Basis dienten dabei nicht die HTML-Test-Reports des AutomationDesk. AutomationDesk generiert für jeden Testbericht auch eine XML-Darstellung, welche bei dieser Anwendung eingelesen wurden. Dabei ist zu beachten, dass die XML-Struktur des AutomationDesk die weitere Verarbeitung erschwert hat. Es wurden die Testbeschreibungen, Bewertungen und Informationen zu Anzeige, wie zum Beispiel Schriftgröße und Schriftfarbe, zusammen in einem Tag abgelegt.

Mit Hilfe der geschaffenen Software war es möglich eine modTF-XML-Datei aus den Test-Reports des AutomationDesk zu generieren. Die Abbildung 6.4 zeigt die Oberfläche für die Konvertierung der Test-Reports.

Im Anschluss konnten alle Komponenten des Test-Report-Generator-Moduls verwendet

Abbildung 6.4.: AutomationDesk-Anbindung GUI

The screenshot shows a Windows-style application window titled "iTest AutomationDesk XML-Erzeugung". The window contains a "File" menu bar and a main form area. The form has several labeled input fields: "Source" (D:\HIL_Projekte\Testprojekt\AutomationDesk), "Title" (Titel des Testreports), "Description" (Beschreibung des Testreports), "Author" (Autor des Testreports), "Version" (Version des Testreports), "Last_revision" (Letzte Änderungen), "Hardwareversion" (Version der getesteten Hardware), "Softwareversion" (Version der getesteten Software), "Customer" (Name des Kunden), "Project" (Name des Projektes), and "Department" (Name der Abteilung). Below these fields are four buttons: "Add project", "Start creation", "Save Meta", and "Exit". There are also two checkboxes: "Report with MKS-TestCase Description" and "Disconnect from MKS-Server after Reportgeneration". At the bottom of the form, there are two more input fields: "MKS Username" (containing the text "X-Trenkel-Isyst") and "MKS Password (only needed if not connected to MKS)". At the very bottom of the window, there is a section labeled "Statusinformationen" with a scrollable area.

werden, ohne das weitere Anpassungen nötig waren.

6.3.2.2. Anbindung von Anforderungsmanagementsystemen

Im Rahmen der AutomationDesk-Anbindung wurde auch die Anbindung an das Requirementsmanagement-System MKS erprobt. Bei der im vorangegangenen Abschnitt beschriebenen Konvertierung des Test-Reports, wurden auch die Anforderungs-IDs aus den Testbeschreibungen extrahiert und entsprechend im modTF-XML abgelegt. Damit war es, mit Hilfe des Test-Report-Generator-Moduls, möglich die Testergebnisse automatisiert in das Requirementsmanagement-System abzugleichen.

In diesem Projekt wurden ca. 1100 Anforderungen getestet. In der Testumsetzung wurden ca. 340 Testfälle in AutomationDesk manuell implementiert, welche ca. 18600 Bewertungen ausgaben. Zu Beginn des Projektes wurden die Ergebnisse manuell in das MKS System eingepflegt, um eine Bestimmung der Anforderungsabdeckung durch den Test zu realisieren. Dies dauerte im Schnitt 40 h pro Softwarerelease. Mit dem automatisierten Abgleich der Ergebnisse konnte diese Arbeitszeit auf weniger als 1 h reduziert werden.

Die Bestimmung der Abdeckung konnte direkt in MKS erfolgen. Dabei konnte gezeigt werden, dass die manuelle Übertragung der Ergebnisse sehr fehleranfällig war. Die automatisierte Auswertung zeigte, dass mehr als 50 % der Anforderungen ungetestet waren. Die manuelle Auswertung hatte eine deutlich bessere Testabdeckung ergeben. Im Laufe des Projektes konnten die Lücken im Test geschlossen werden. Durch den automatisierten Abgleich der Ergebnisse konnten die Fortschritte schnell und einfach überwacht werden. Es konnte damit eine signifikante Verbesserung der Testabdeckung und damit der Produktqualität erzielt werden.

7. Vergleich zu bestehenden Testautomatisierungssystemen

Die Eigenschaften des modTF werden unter verschiedenen Gesichtspunkten im Folgenden mit anderen Testautomatisierungslösungen – EXAM, iTestStudio, TA2/TA3, AutomationDesk und ECU-Test – verglichen werden. Dabei sollen die erkannten Stärken und Schwächen des in dieser Arbeit vorgeschlagenen modularen Ansatzes verdeutlicht werden.

7.1. Vergleich des Integrationsaufwandes

In diesem Abschnitt wird der Vergleich des ermittelten Aufwandes für die Integration der Testautomatisierung in einen bestehenden Testprozess mit bestehenden Testsystemen erörtert. Dabei wird davon ausgegangen, dass bisher keine Testautomatisierung eingesetzt wird. Das Testsystem bietet eine Automatisierungsschnittstelle. Dabei besteht der hauptsächlich Aufwand in der Anbindung des vorhandenen Testsystems.

Bei modTF ist in diesem Fall ein Port, welcher die Anbindung des Testsystemes übernimmt, zu erstellen. Die Anbindung der Echtzeit-Middleware Gamma V umfasst ca. 160 Zeilen Python-Code – inklusive Kommentaren.

Bei den Systemen EXAM, iTestStudio und TA2/TA3 ist der Implementierungsaufwand mit dem des modTF vergleichbar. Es stehen Schnittstellenvorgaben zur Verfügung, welche für das neue Testsystem umzusetzen sind. Dabei ist festzustellen, dass der Umfang des zu erstellenden Codes im Schnitt größer ist als bei modTF.

Bei dem System AutomationDesk ist der Aufwand deutlich höher. Es ist zwar die Programmierung eigener Elemente mit Python möglich, aber die fehlende Vorgabe von Schnittstellen erschwert die Integration neuer Systeme.

Bei ECU-Test ist der Aufwand am größten, da für jedes neue Bibliotheks-Element, wie es für die Anbindung eines Testsystem notwendig ist, ein System aus zwei Python-Klassen zu programmieren ist. Die eine Klasse stellt die Oberfläche innerhalb der ECU-Test-Oberfläche zur Verfügung. Die zweite Klasse realisiert die eigentliche Anbindung des Testsystems. Alleine die Realisierung eines dieser Bibliotheks-Elemente übersteigt den Umfang von 160 Zeilen-Python-Code um ein Vielfaches.

7.2. Vergleich des Portierungsaufwandes bei dem Wechsel zwischen Testsystemen

In diesem Abschnitt wird der Vergleich des Aufwandes für die Portierung vorhandener Testfälle auf ein neues Testsystem betrachtet. Es kommen dabei die Erfahrungen aus der Anbindung der Echtzeit-Middleware Gamma V zum Tragen.

7.2.1. Vergleich des Initialaufwandes

Der Initialaufwand für das Aufsetzen der Testumgebung entspricht der im Kapitel 7.1 beschriebenen Anbindung des Testsystems. Alle weiteren ermittelten Aufwände beziehen sich auf die einzelnen Testfälle.

7.2.2. Vergleich des Implementierungsaufwandes der Testfälle

Der Hauptaufwand bei dem Umstieg von einem Testsystem zu einem Anderen liegt in der Portierung der Testskripte. Je besser die Trennung zwischen den Testskripten und der Testsystem-Anbindung ist, desto geringer ist der Aufwand.

Für die Umstellung auf ein neues Testsystem ist bei TTCN-3 der Port für die Testsystem-Anbindung im jeweiligen Testskript anzupassen. Wenn der Port in einer TTCN-3 Datei global definiert ist, so ist die Anpassung nur einmal notwendig. Je nach Art des Testsystems ist auch die Anpassung der Variablenzugriffe (z. B. Name der Variablen im Gamma V-Datenmodell oder Pfad im dSPACE-Modell) anzupassen.

Bei den Systemen EXAM und iTestStudio gibt es definierte Schnittstellen für das Ansprechen eines Testsystems. Bei diesen beiden System ist der Aufwand für die Portierung etwas geringer einzuschätzen als bei modTF. Die Instanziierung der Testsystem-Anbindung erfolgt an einer zentralen Stelle und muss daher nur einmalig geändert werden. Die Verknüpfung von Variablennamen innerhalb der Testskripte mit den Variablen des Testsystems, werden in getrennten Datenstrukturen realisiert. Daher ist eine Anpassung der eigentlichen Testskripte überflüssig. Nachteilig ist bei diesen Systemen, dass die definierte Schnittstelle zum Testsystem an die Schnittstelle der Systeme von dSPACE angelehnt ist. Dies führt zu einem erhöhten Umfang des Codes für die Anbindung anderer Testsysteme. Bei TA2/TA3 ist ein höherer Aufwand als bei modTF zu erkennen, da die Variablenzugriffe zum Testsystem in jedem Testskript verteilt auftreten. Es müssen damit alle Zugriffe auf einzelne Variablen des Testsystems angepasst werden.

Bei AutomationDesk und ECU-Tests ist der Aufwand am Größten. Da es keine einheitlichen Schnittstellen für Testsysteme gibt, sind neue Blöcke (Bibliotheks-Elemente) für den Variablenzugriff auf das Testsystem zu erstellen. Damit ist in jedem Testskript, jeder Block für den Zugriff auf das Testsystem auszutauschen.

7.3. Vergleich des Implementierungsaufwandes beim Einsatz verschiedener Testbeschreibungsmethoden

In diesem Abschnitt werden die Ergebnisse des Vergleichs des Aufwands bei der Implementierung von Testfällen auf verschiedenen Testautomatisierungssystemen – TTCN-3 (modTF), Python (iTestStudio), UML (EXAM) – dargestellt.

7.3.1. Vergleich des Initialaufwandes

Beim initialen Aufwand unterscheiden sich die genannten Systeme nur wenig. Es ist bei allen drei Systemen notwendig, die Konfiguration des Testsystems – z. B. Pfad zum Datenmodell – anzupassen. Weiterhin sind die Variablen des Testsystems auf Variablen innerhalb der Testautomatisierung zu mappen. Einzig durch die grafische Oberfläche und den häufigen Wechsel zwischen Eingabeelementen, ist das Mapping der Variablen unter EXAM mit einem höheren zeitlichen Aufwand verbunden.

7.3.2. Vergleich des Implementierungsaufwandes der Testfälle

Bei der Implementierung von Testfällen sind modTF allen anderen System überlegen. Einzig iTestStudio ermöglicht eine ähnlich effektive Testumsetzung wie modTF. Bei der Überwachung von Timeouts (Ausbleibende Reaktion des zu testenden Steuergerätes) und bei der Prüfung mehrerer verschiedener Möglichkeiten für eine Bewertung zeigt modTF bzw. TTCN-3 seine Vorteile. Außerdem bietet die Verwendung eines eigenen Datentyps für Bewertungen und die strenge Typ-Prüfung von TTCN-3 eine geringere Fehleranfälligkeit bei der Erstellung von Testfällen. Weiterhin ermöglicht TTCN-3 mit der GFT auch eine grafische Darstellung der Testfälle.

EXAM erfordert einen deutlich höheren Aufwand bei der Testerstellung, da die grafische Zusammenstellung der Testschritte zeitaufwändig ist. Des Weiteren werden komplexere Testfälle schnell unübersichtlich. Die grafische Erstellung von Testfällen erleichtert den Einstieg für unerfahrene Personen im Bereich Test nicht. Der gebotene Funktionsumfang von EXAM erfordert einen hohen Einarbeitsaufwand für die Testerstellung und eine einfachere Wartbarkeit ist ebenfalls nicht gegeben.

8. Zusammenfassung und Ausblick

8.1. Zusammenfassung

Mit dem entwickelten Konzept der modularen, portierbaren Middleware sowie der Realisierung als modTF konnte gezeigt werden, dass ein modularer Ansatz für die Automatisierung von Tests umsetzbar ist und bisher bekannte Lösungen bezüglich des Aufwandes und erreichbarer Ergebnisse übertrifft.

Mit dem Einsatz der standardisierten Sprache TTCN-3 für die Realisierung der Tests als zentrales Speicherformat für die Testfälle, konnte ein einfacher Weg für den Austausch von Testfällen zwischen verschiedenen Testabteilungen vom Zulieferer und vom OEM geschaffen werden. Einerseits können die Testfälle durch die standardisierte Sprache in verschiedenen Laufzeitumgebungen ausgeführt werden. Andererseits bietet der Framework modTF mit der flexiblen Anbindung verschiedener Testsysteme die Möglichkeit, die Testfälle mit der Laufzeitumgebung in Form des Test-Ausführungs-Moduls zwischen den Testabteilungen auszutauschen.

Der realisierte modulare TTCN-3 Interpreter des Test-Ausführungs-Moduls ermöglicht mit der standardisierten Schnittstelle zu Testsystemen die schnelle und einfache Anbindung neuer Testsysteme. Dies erlaubt eine problemlose Erweiterung von Testsystemen. Zugleich wird die einfache Portierung von Testfällen zwischen verschiedenen Testsystemen sichergestellt, ohne dass weitreichende Änderungen an der Implementierung der Testfälle notwendig sind.

Durch die leichte Portierbarkeit und die einfache Einbindung neuer Testsysteme ist auch die Lauffähigkeit der Testfälle über die Lebensdauer des zu testenden Produktes sichergestellt. Es ist damit nicht mehr zwingend notwendig, dass spezielle Testsysteme, welches während der Entwicklungsphasen eingesetzt wurde, auch über die gesamte Lebensdauer des Produktes zur Verfügung stehen.

Mit Hilfe der standardisierten Anbindung von Testsystemen und der mittels TTCN-3 möglichen Umsetzung von Testfällen auf verschiedenen Abstraktionsebenen ist die Realisierung von Testfällen, vom Modultest bis zum Fertigungsendtest, mit dem hier vorgestellt Ansatz des modTF möglich. Somit ist es möglich, das in der Entwicklung gewonnene Know-how aus dem Testbereich bis hin zum Fertigungsendtest wieder zu verwenden, und die Aufwände für die Portierung bzw. Neuimplementierung der Testfälle auf verschiedenen Testautomatisierungslösungen für die unterschiedlichen Teststufen entfallen. Mit der breiteren Verwendung von Testfällen und deren Wiederverwendung dieser Testfälle auf verschiedenen Teststufen sowie über verschiedene Testprojekte hinweg, kann

eine deutliche Verbesserung der Qualität der Testfälle erreicht werden. Die Wiederverwendung dieser Testfälle trägt ebenfalls zur Steigerung auf den verschiedenen Teststufen bei. Weiterhin führt der Einsatz des TRex-Plugin für das Test-Beschreibungs-Modul des modTF zu einer Erhöhung der Codequalität der Testfälle. Die weiterreichende, statische Codeanalyse und die Refactoring-Funktionen unterstützen ebenso die Entwicklung qualitativ hochwertigen Codes für die Testfälle.

Mit dem Entwurf und der Umsetzung des modTF-XML-Formates für die Speicherung der Testergebnisse und der zugehörigen Meta-Daten, konnte ein flexibles und erweiterbares Format für Test-Reports geschaffen werden. Es erlaubt eine maschinelle Verarbeitung und bleibt dabei für den Nutzer lesbar. Die Integration von Meta-Daten sowie die Integrierbarkeit von Bildern und Messreihen bzw. von anderen binären und digitalen Daten vereinfacht die Verwaltung der Testergebnisse stark. Mit Hilfe des geschaffenen Test-Report-Generator-Moduls und den darauf aufbauenden Test-Reportgeneratoren, können verschiedene Sichten auf die Testergebnisse – z. B. im HTML-Format – einfach erzeugt werden.

Durch die Verwendung von textbasierten Formaten in Form von TTCN-3 für die Testfälle und dem modTF-XML-Format für die Testergebnisse, ist eine direkte Integration des Testprozesses in vorhandene Versionsmanagement-Systeme möglich. Es können damit die für die Softwareentwicklung von eingebetteten Systemen verwendeten Versionsmanagement-Systeme nahtlos verwendet werden. Auch der direkte Vergleich verschiedener Versionsstände ist ohne zusätzliche Software möglich, welche bei binären Speicherformaten notwendig wäre. Mit der durchgehenden Berücksichtigung von Anforderungs-IDs, beginnend bei der Testfallerstellung über die Testausführung bis hin zu den Testergebnissen, ist es möglich, die durchgehende Nachverfolgbarkeit in fortschrittlichen Entwicklungsprozessen sicher zu stellen. Im Rahmen der Arbeit konnte der automatisierte Abgleich von Testergebnissen am Beispiel des Requirementsmanagement-System MKS gezeigt werden. Im praktischen Einsatz zeigte sich durch die durchgehende Nachverfolgbarkeit neben der Zeitersparnis ebenfalls eine signifikante Steigerung der Produktqualität durch eine schnellere Erkennung nicht oder fehlerhaft getesteter Anforderungen während der Entwicklung. Auf Basis der erarbeiteten Schnittstellen ist auch eine Anbindung anderer Requirementsmanagement-System möglich.

Zusammenfassend ist festzustellen, dass mit Realisierung der modularen, portierbaren Middleware als modTF ein modulares System zur Testautomatisierung geschaffen wurde, welches viele der aktuellen Probleme im Bereich des automatisierten Software-Tests löst. Durch den modularen Ansatz ist das System flexibler einsetzbar und einfacher in bestehende Entwicklungsprozesse zu integrieren als dies bisher verfügbare Lösungen zulassen. Mit dem modTF ist ein nachhaltiger Einsatz von Tests für eingebettete Systeme möglich.

8.2. Ausblick

Die vorliegende Dissertation stellt die Konzeptionierung und Realisierung eines modularen Testautomatisierungs-Framework vor. Im Rahmen der praktischen Umsetzung konnten aus zeitlichen Gründen nicht alle im Konzept dargestellten Aspekte vollständig umgesetzt werden.

Dies beginnt bei der grafischen Modellierung von Testfällen. Die hier dargestellte Umsetzung unterstützt das grafische Formate GFT für die Erstellung von Testfällen in TTCN-3 nicht. Die Unterstützung des GFT sowie die daraus ableitbare Unterstützung weiterer grafischer Modellierungsmöglichkeiten – wie zum Beispiel UML – sind in einer Weiterführung der Arbeit zu realisieren.

Der umgesetzte TTCN-3 Interpreter unterstützt nicht den vollständigen Sprachumfang. Die Unterstützung verteilter Testsysteme sowie die Integration der Erweiterung für Real Time Testing ist für den breiteren Einsatz des modTF notwendig. Weiterhin ist die Umsetzung eines übergeordneten Testmanagements in Form des TM-Moduls zu realisieren. Zur besseren Unterstützung der Testentwicklung ist eine Debug-Funktionalität auf Ebene des TTCN-3 Codes zu realisieren. Dabei kann auf die konzeptionellen Betrachtungen aus dieser Arbeit aufgebaut werden.

Für die praktische Nutzung des modTF ist die Unterstützung von weiteren Testsystemen zu realisieren. Basierend auf der geschaffenen Schnittstelle sind Ports für Testsysteme – wie zum Beispiel dSPACE oder National Instruments – zu implementieren und zu erproben.

Eine weiterführende Erprobung in realen Projekten erscheint sinnvoll, um die Möglichkeiten und Grenzen der hier vorstellten Lösung noch besser bewerten zu können. Damit verbunden ist eine Erweiterung der Anbindung an verschiedene Requirementsmanagement-Systeme sinnvoll, um die Universalität der geschaffenen Schnittstellen zu überprüfen.

Abschließend erscheint die Untersuchung zu einer weiteren Automatisierung im Bereich des Software-Tests als sinnvoll. Die stark zunehmende Komplexität der Systeme erfordert neue Ansätze für den Test. Die in dieser Arbeit geschaffene automatisierte Anbindung an die Anforderung ist dabei nur als erster Schritt zu sehen. Mit der Formalisierung der Anforderungen wäre eine automatisierte Generierung für Testfälle möglich. Mit formaler Spezifikation, wie sie zum Beispiel durch das Tool SpecScribe [128] unterstützt wird, wäre eine durchgehende Automatisierung der Entwicklung und des Tests von eingebetteten Systemen möglich.

Anhang A.

Eigene Veröffentlichungen

begutachtete Veröffentlichungen

1. Trenkel, K.: *Development of a universal Ethernet interface for different microcontrollers*, Tagung "ECUMICT 2006", KaHo St Lieven, Gent, S. 61 - 69, ISBN 9-08082-552-2
2. Trenkel, K.; Heinkel, U.: *Absicherung der Buskommunikation – Bussystemübergreifende Wiederverwendbarkeit von Testfällen*, 4. GMM-Fachtagung "AmE 2013 Automotive meets Electronics", 19.- 20. Februar 2013, Dortmund, S. 89 - 94, ISBN 978-3-8007-3485-6
3. Trenkel, K.; Heinkel, U.; Spittler, F.; Tremmel, J.: *Modular Test Framework – From Component Test to Production Test*, Fachtagung "ETS2013 (IEEE European Test Symposium)", 27.-31. Mai 2013, Avignon, France, ISBN 978-1-4673-6375-4
4. Trenkel, K.; Spittler, F.; Rauch, H.; Heinkel, U.: *modTF - ein modulares Framework zur Testautomatisierung*, Tagungsband Embedded Software Engineering Kongress 2013, 02.-06. Dezember 2013, Sindelfingen, Deutschland, S. 358 - 367, ISBN 978-8343-2408-5
5. Trenkel, K.: *Echtzeitfähige Sensorsimulation für Entwicklung und Test*, 5. GMM-Fachtagung "AmE 2014 Automotive meets Electronics", 18.-19. Februar 2014, Dortmund, S. 147 - 150, ISBN 978-3-8007-3580-8
6. Trenkel, K.; Heinkel, U.: *An Advanced Modular and Portable Test Automation Framework for Practical Use*, Embedded World Conference, 25.-27. Februar 2014, Nürnberg, Session 8 Beitrag 3, ISBN 978-3-645-50131-6

nicht begutachtete Veröffentlichungen

1. Trenkel, K.: *Verknüpfung der Testautomatisierung mit dem Requirements Management für Automotive SPICE*. Tagung "5. Neu-Ulmer Test-Engineering-Day 2010", 09 Juni 2010, Ulm
2. Trenkel, K.: *Testautomatisierung für Eingebettete Systeme in Zeiten agiler Entwicklungsprozesse*. Tagung "BICCnet - Innovation Forum Embedded Systems 2011", 8. April 2011, Konferenzzentrum München
3. Trenkel, K.: *Modulare Testsysteme – Integration von E4Y Komponenten zu einem HIL-System*. Tagung "E4Y TechDay", 13. September 2012, München
4. Trenkel, K.; Rauch, H.: *Modular und skalierbar: Automatisierung des Tests von eingebetteten Systemen*. Automation Vally Nord-Bayern, elektrotechnik 94/2012, Vogel Business Media, Würzburg, 23 November 2012, S. 13 - 15, ISSN 1619-9405
5. Trenkel, K.: *HIL-Test – Software-Test auf realer Hardware*. Fachvortrag "ASQF – FG Automatisierung Franken", 18. Oktober 2013, Bubenreuth
6. Trenkel, K.; Heinkel, U.; Spitteller, F.: *Further Development of Test Automation – Combination of Practical Experience and Academic Research*, Tagung "Belgium Testing Days 2013", 28. Februar 2013, Brüssel, Belgium

Anhang B.

Literaturverzeichnis

- [1] K. Trenkel, "Development of a universal Ethernet interface for different micro-controllers: KaHo St Lieven, Gent, ISBN 9-08082-552-2," in *Tagung "ECUMICT 2006"*, pp. 61–69.
- [2] U. Vigenschow, *Testen von Software und Embedded Systems: Professionelles Vorgehen mit modellbasierten und objektorientierten Ansätzen*, 2nd ed. Heidelberg: dpunkt-Verl., 2010.
- [3] C. Haubelt and J. Teich, *Digitale Hardware / Software-Systeme: Spezifikation und Verifikation*, ser. eXamen.press. Berlin u.a: Springer, 2010.
- [4] K. Borgeest, *Elektronik in der Fahrzeugtechnik: Hardware, Software, Systeme und Projektmanagement*. Springer Vieweg, 2013.
- [5] Peter Clarke, "Audi selects Tegra processor for infotainment and dashboard," 17.01.2012. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1260984
- [6] Robert Bosch GmbH, "PSI5 Peripheral Sensor Interface 5." [Online]. Available: <http://psi5.org/>
- [7] SAE International, "J 2716 - SENT - Single Edge Nibble Transmission for Automotive Applications," 27.01.2010. [Online]. Available: http://standards.sae.org/j2716_201001/
- [8] K. Wüst, *Mikroprozessortechnik: Grundlagen Architekturen Schaltungstechnik und Betrieb von Mikroprozessoren und Mikrocontrollern*, 4th ed., ser. SpringerLink : Bücher. Wiesbaden: Vieweg+Teubner, 2011.
- [9] International Organization for Standardization, "ISO 9141: Road vehicles - diagnostic systems - requirements for interchange of digital information," Genève, 1994. [Online]. Available: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=16738
- [10] —, "ISO 17987 - Local Interconnect Network (LIN)," Genève, 2013.
- [11] —, "ISO 11898-1 – Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling: Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling," Geneva, 2003.

- [12] W. Zimmermann, *Bussysteme in der Fahrzeugtechnik: Protokolle und Standards*, 2nd ed., ser. SpringerLink : Bücher. Wiesbaden: Vieweg+Teubner Verlag, 2007.
- [13] W. Zimmermann and C. Schmidgall, *Bussysteme in der Fahrzeugtechnik: Protokolle, Standards und Softwarearchitektur ; mit 103 Tabellen*, 4th ed. Wiesbaden: Vieweg + Teubner, 2011.
- [14] MOST Cooperation, “Real Interconnectivity - Networked Intelligence,” Karlsruhe. [Online]. Available: <http://www.mostcooperation.com/home/index.html>
- [15] H.-W. Schaal, “Ethernet und IP im Kraftfahrzeug: Neue Anforderungen an das Entwicklungswerkzeug durch den Ethernet- und IP-Einsatz,” *Elektronik automotive*, vol. 2012, no. 04.2012, pp. 38–41. [Online]. Available: http://www.vector.com/portal/medien/cmc/press/PON/Ethernet_IP_ElektronikAutomotive_201204_PressArticle_DE.pdf
- [16] AUTOSAR development cooperation, “Home: Welcome to the AUTOSAR development partnership,” 2013. [Online]. Available: <http://www.autosar.org/index.php?p=0&up=0&uup=0&uuup=0>
- [17] F. Reimann, M. Glaß, J. Teich, and Ulrich Abelein, “Szenarienbasierte Integration von Diagnosefunktionalität in E/E Architekturen,” in *GMM-Fachbericht – Automotive meets Electronics (AmE)*, 2013, pp. 15–20.
- [18] VDI/VDE Innovation + Technik GmbH, “SEIS - Sicherheit in Eingebetteten IP-basierten Systemen.” [Online]. Available: <http://strategiekreis-elektromobilitaet.de/public/projekte/seis>
- [19] F. Reimann, M. Glaß, and J. Teich, “Migration Strategies for Ethernet-based E/E Architectures,” in *Proceedings of the Embedded World Conference*, 2013, p. 7.
- [20] H. Eisele, “CAN with Flexible Data Rate – Eigenschaften, Standardisierung und Einsatz im Automobil,” in *AmE 2013*, VDE GMM, Ed. VDE VERLAG GMBH, 2013, pp. –.
- [21] Peter Hank, “Ethernet and extended existing network protocols enable tomorrow’s mobility,” in *AmE 2013*, VDE GMM, Ed. VDE VERLAG GMBH, 2013, pp. 61–62.
- [22] IEEE, “IEEE standard glossary of software engineering terminology,” New York, 31.12.1990. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=159342&filter%3DAND%28p_Publication_Number%3A2238%29
- [23] H.-G. Baum, A. G. Coenenberg, and T. Günther, *Strategisches Controlling*, 4th ed. Stuttgart: Schäffer-Poeschel, 2007. [Online]. Available: http://deposit.ddb.de/cgi-bin/dokserv?id=2803424&prov=M&dok_var=1&dok_ext=htm
- [24] Markus Rentschler, “Design für Testability: Effizienzsteigerung für den gesamten Produktlebenszyklus,” Berlin, 27.09.2012.

- [25] A. Spillner and T. Linz, *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester ; Foundation Level nach ISTQB-Standard*, 3rd ed. Heidelberg: dpunkt-Verl, 2007.
- [26] A. Spillner, *Praxiswissen Softwaretest - Testmanagement: Aus- und Weiterbildung zum Certified Tester ; Advanced Level nach ISTQB-Standard*, 2nd ed. Heidelberg: dpunkt-Verl, 2008.
- [27] H. Balzert, *Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*, ser. Lehrbücher der Informatik. Heidelberg: Spektrum Akad. Verl, 1998, vol. 2.
- [28] Jens Coldewey, "Leichte Prozesse Motivation und Ueberblick," 2001. [Online]. Available: <http://www.coldewey.com/publikationen/conferences/GI/Glashuetten.Proc.7.3.2001.pdf>
- [29] J. Humble and D. Farley, *Continuous delivery: [reliable software releases through build, test, and deployment automation]*, ser. A Martin Fowler signature book. Upper Saddle River and NJ: Addison-Wesley, 2011.
- [30] K. Beck, *Test-driven development: By example*, 14th ed., ser. A Kent Beck signature book. Boston and Mass: Addison-Wesley, 2009.
- [31] K. Schwaber, *Agiles Projektmanagement mit Scrum*, 1st ed. s.l: Microsoft Deutschland GmbH, 2007. [Online]. Available: http://ebooks.ciando.com/book/index.cfm/bok_id/29410
- [32] Verband der Automobilindustrie e.V., "Automotive SPICE," Berlin, 2012. [Online]. Available: <http://www.automotivespice.com/>
- [33] International Organization for Standardization, "ISO/IEC 15504 SPICE," Genève, 2004.
- [34] J. Teich and C. Haubelt, *Digitale Hardware/Software-Systeme: Synthese und Optimierung*, 2nd ed., ser. eXamen.press. Berlin and Heidelberg: Springer-Verlag Berlin Heidelberg, 2007. [Online]. Available: <http://dx.doi.org/10.1007/978-3-540-46824-0>
- [35] Hitex Development Tools, "Tessy - Automated Unit / Module / Integration Testing of Embedded Software," 2012. [Online]. Available: <http://www.hitex.com/index.php?id=module-unit-test>
- [36] Christiane Brünglinghaus, "Produktentwicklung in der Automobilindustrie: Fahrzeugtechnik," 30.01.2013. [Online]. Available: <http://www.springerprofessional.de/produktentwicklung-in-der-automobilindustrie/3930746.html>
- [37] M. Reimer, R. Brück, M. Wahl, I. Birner, and S. Sturm, "Ganzheitlicher Ansatz zur Synchronisation der E/E-Applikationsentwicklung innerhalb des Netzwerkes der Automobil- und Halbleiterindustrie," Offenbach, 2013.

- [38] P. Sebastiao Correia and T. Kilic, “IP-basierte Fahrzeugdiagnose mit Fokus auf einen Werkstattserver,” Offenbach, 2013.
- [39] dSPACE GmbH, “Real-Time Hardware,” 2012. [Online]. Available: <http://www.dspace.com/en/inc/home/products/systems/functp/echtzhw.cfm>
- [40] MicroNova AG, “EXAM,” 2012. [Online]. Available: <http://www.exam-ta.de/en.html>
- [41] iSyst Intelligente Systeme GmbH, “Hardware In The Loop (HIL) – Automotive: HIL-Testumgebung für komplexe Steuergeräte und Regelungen,” 2012. [Online]. Available: http://www.isyst.de/kompetenzen/dienstleistungen/hil_automotive.php
- [42] dSPACE GmbH, “Echtzeit-Hardware,” 2012. [Online]. Available: <http://www.dspace.com/de/gmb/home/products/systems/functp/echtzhw.cfm>
- [43] National Instruments Corporation, “<http://www.ni.com/f/solutions/62/8461/de/>,” 2013. [Online]. Available: <http://www.ni.com/f/solutions/62/8461/de/>
- [44] I. The MathWorks, “xPC Target: Perform hardware-in-the-loop simulation and real-time rapid control prototyping,” 2013. [Online]. Available: www.mathworks.de/products/xpctarget/index.html
- [45] dSPACE GmbH, “DS5202 FPGA Base Board,” 2013. [Online]. Available: http://www.dspace.com/de/gmb/home/products/hw/modular_hardware_introduction/i_o_boards/ds5202.cfm?nv=n2
- [46] National Instruments Corporation, “NI HIL Simulator Reference System,” 2013. [Online]. Available: <http://sine.ni.com/nips/cds/view/p/lang/de/nid/207822>
- [47] Daniel Lohmann, “Der Scheduler von Windows,” 2013. [Online]. Available: https://www4.cs.fau.de/Lehre/WS06/V_BS/fohlen/09-Sched-Windows-2x2.pdf
- [48] dSPACE GmbH, “AutomationDesk,” 2012. [Online]. Available: <http://www.dspace.com/en/inc/home/products/sw/expsoft/automdesk.cfm>
- [49] TraceTronic, “ECU-TEST your innovation: start with high quality test automation today,” 2011. [Online]. Available: <http://www.tracetronic.de/produkte/ecu-test.html>
- [50] Kristian Trenkel and Hans Rauch, “Modular und skalierbar: Automatisierung des Tests von eingebetteten Systemen: Mini HIL-System,” in *elektrotechnik*, vol. 94/2012, pp. 13–15. [Online]. Available: <http://www.elektrotechnik.vogel.de/messtechnik-prueftechnik/articles/386792/index2.html>
- [51] C. Heinz and R. Weigel, *Automatisierung – Auf dem Weg zur effizienten Testfallerstellung für den (Hardware-)Funktionstest*, ser. AmE 2013. VDE-Verlag, 2013.
- [52] Python Software Foundation, “Python Programming Language – Official Website,” 2013. [Online]. Available: <http://www.python.org/>
- [53] International Organization for Standardization, “ISO/IEC 23270 – C# Language

- Specification,” Geneva, 2006. [Online]. Available: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=42926
- [54] —, “ISO 13209 - Open Test sequence eXchange format (OTX),” 2012.
- [55] ETSI, “WELCOME TO ETSI’S OFFICIAL TTCN-3 HOMEPAGE: Testing and Test Control Notation Version 3 (TTCN-3),” 2013. [Online]. Available: <http://www.ttcn-3.org/>
- [56] —, “TTCN-3: Core Language,” 2013. [Online]. Available: http://www.etsi.org/deliver/etsi_es/201800_201899/20187301/04.05.01_60/es_20187301v040501p.pdf
- [57] —, “TTCN-3: Operational Semantics,” 2013. [Online]. Available: http://www.etsi.org/deliver/etsi_es/201800_201899/20187304/04.04.01_60/es_20187304v040401p.pdf
- [58] —, “TTCN-3: TTCN-3 Runtime Interface,” 2013. [Online]. Available: http://www.etsi.org/deliver/etsi_es/201800_201899/20187305/04.05.01_60/es_20187305v040501p.pdf
- [59] —, “TTCN-3: TTCN-3 Control Interface,” 2013. [Online]. Available: http://www.etsi.org/deliver/etsi_es/201800_201899/20187306/04.05.01_60/es_20187306v040501p.pdf
- [60] —, “TTCN-3 Application Areas,” 2009. [Online]. Available: <http://www.ttcn-3.org/ApplicationAreas.htm>
- [61] World Wide Web Consortium, “Extensible Markup Language (XML),” 2013. [Online]. Available: <http://www.w3.org/XML/>
- [62] IEEE, *IEEE 1445-1998 standard for digital test interchange format (DTIF)*. New York: Institute of Electrical and Electronics Engineers, 1999.
- [63] —, *IEEE 1450-1999 standard for extensions to standard test interface language (STIL) for DC level specification*. New York and N.Y: Institute of Electrical and Electronics Engineers, 2003.
- [64] —, *IEEE 1671-2010 standard for Automatic Test Markup Language (ATML) for exchanging automatic test equipment and test information via XML*. New York: Institute of Electrical and Electronics Engineers, 2011.
- [65] Test Anything Protocol, “Main Page: TAP, the Test Anything Protocol,” 2007. [Online]. Available: http://www.testanything.org/wiki/index.php/Main_Page
- [66] Test Environment Toolkit, “TETworks: Home page for the Test Environment Toolkit (TET),” 2012. [Online]. Available: <http://tetworks.opengroup.org/>
- [67] ASAM e.V., “ASAM HIL V1.0.2,” 2013. [Online]. Available: [http://www.asam.net/nc/home/standards/standard-detail.html?tx_rbwmbmasamstandards_pi1\[showUid\]=1534&start=](http://www.asam.net/nc/home/standards/standard-detail.html?tx_rbwmbmasamstandards_pi1[showUid]=1534&start=)

- [68] Object Management Group, “Object Management Group,” 2013. [Online]. Available: <http://www.omg.org/>
- [69] International Organization for Standardization, “ISO/IEC 19505 – Object Management Group Unified Modeling Language (OMG UML),” 2012. [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=32624
- [70] sepp.med gmbh, “Modellbasiertes Testen (MBT) in der Praxis,” 2013. [Online]. Available: <http://www.seppmed.de/produkte/mbtmzt.html>
- [71] The MathWorks, Inc., “Simulink: Simulation und Model-BASed Design,” 2013. [Online]. Available: <http://www.mathworks.de/products/simulink/>
- [72] —, “MATLAB: Die Sprache für technische Berchnungen,” 2013. [Online]. Available: <http://www.mathworks.de/products/matlab/>
- [73] National Instruments Corporation, “Systemdesignsoftware NI LabVIEW,” 2013. [Online]. Available: <http://www.ni.com/labview/d/>
- [74] emotive GmbH & Co. KG, “OTX - Open test sequence data eXchange format,” 2011. [Online]. Available: <http://www.emotive.de/en/de/doc/car-diagnostic-systems/applications/otx>
- [75] dSPACE GmbH, “AutomationDesk,” 2012. [Online]. Available: <http://www.dspace.com/de/gmb/home/products/sw/expsoft/automdesk.cfm>
- [76] MicroNova AG, “Kundenzeitschrift Juni 2009 Sonderausgabe EXAM,” 2009. [Online]. Available: http://www.micronova.de/images/stories/Unternehmen/Kundenzeitschrift/kundenzeitschrift_2009_juni_sonderausg_exam.pdf
- [77] TraceTronic, “Testautomatisierungs-Software ECU-TEST,” 2010. [Online]. Available: <http://www.tracetronic.de/produkte/ecutest.html>
- [78] Testing Technologies IST GmbH, “TTworkbench - The Reliable Test Automation Platform,” 23.12.2013. [Online]. Available: <http://www.testingtech.com/products/ttworkbench.php>
- [79] MBtech Group GmbH & Co. KGaA, “PROVEtech:TA – Das umfassende Werkzeug zur Testautomation.” 2012. [Online]. Available: https://www.mbtech-group.com/eu-de/electronics_solutions/tools_equipment/provetechta_test_automation.html
- [80] OpenTTCN Ltd, “OpenTTCN Tester 2012 Tour,” 2014. [Online]. Available: <http://www.openttcn.com/>
- [81] Fraunhofer FIRST, “TTCN-3 TUTORIAL,” 2005. [Online]. Available: <http://ttcn-3.net/tutorial.html>
- [82] Universität Göttingen, “TRex - the TTCN-3 Refactoring and Metrics Tool.” [Online]. Available: <http://www.trex.informatik.uni-goettingen.de/>
- [83] University of Science and Technology of China, “TTCN Lab of USTC,” 2010.

- [Online]. Available: <http://ttn.ustc.edu.cn/MainPageEn.html>
- [84] Brodbit, "BTT - BroadBit Test Tool." [Online]. Available: <http://www.broadbit.com/page8/page2/page2.html>
- [85] INRIA, "T3DevKit," 18.05.2009. [Online]. Available: <http://www.irisa.fr/tipi/wiki/doku.php/t3devkit>
- [86] Ricardo Rezzano, Ariel Sabiguero, Frank Le Gall, Xiaohong Huang, Nikolay Pakulin, Xianrong Wang, Anthony Baire, "An open compiler for TTCN-3: picoTTCN-3," 2010. [Online]. Available: <http://www.ttcn-3.org/TTCN3UC2010/June10/Paper13-An%20open%20compiler%20for%20TTCN-3%20picoTTCN-3.pdf>
- [87] INRIA, "Go4IT Project," 21.07.2008. [Online]. Available: http://www.irisa.fr/tipi/wiki/doku.php/go4it_project
- [88] W. Borgert, "tthreeparser - parse TTCN-3 files: tthreeparser," 27.04.2001. [Online]. Available: <http://manpages.ubuntu.com/manpages/gutsy/man1/tthreeparser.1.html>
- [89] IEEE, "IEEE 829-2008 - IEEE Standard for Software and System Test Documentation," 2008. [Online]. Available: <http://standards.ieee.org/findstds/standard/829-2008.html>
- [90] International Organization for Standardization, "ISO/IEC/IEEE 29119:2013 – Software testing," Geneva, 2013. [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=45142
- [91] Dr. Roman Nagy, "Use-Case-basiertes Testen von AUTOSAR-Softwarekomponenten," Berlin, 2012.
- [92] Florian Prester, "Model-centric testing," in *testing experience magazine*, vol. 9 (01-2010), pp. 108–111. [Online]. Available: http://www.testingexperience.com/issues/testingexperience01_10.pdf
- [93] Rolf Hänisch, "SPEZIFIZIEREN, MODELLIEREN UND TESTEN IN DER AUTOMATISIERUNG MIT LIMBO," Berlin, 2012.
- [94] F. Z. Winfried Dulz, "MaTeLo - Statistical Usage Testing by Annotated Sequence Diagrams, Markov Chains and TTCN-3," in *Proceeding QSIC '03 Proceedings of the Third International Conference on Quality Software*, p. 336.
- [95] Jürgen Großmann, Hans-Werner Wiesbrock, "Wiederverwendbarkeit und Management von modellbasierten X-in-the-Loop Tests mit TTCN-3 Embedded," 2010. [Online]. Available: http://www.temea.org/media/download/wiederverwendbarkeit_und_management_von_modellbasierten_x-in-the-loop_tests_mit_ttcn-3_embedded.pdf
- [96] Clark Wiedmann, "A Performance Comparison between an APL Interpreter and Compiler," in *APL '83 Proceedings of the international conference on APL*, vol.

- Volume 13, pp. 211–217. [Online]. Available: <http://delivery.acm.org/10.1145/810000/801219/p211-wiedmann.pdf>
- [97] W. Staar, “Untersuchung der Schnittstelle zwischen Requirements-Management-Tools und der Testautomatisierung: Diplomarbeit,” Master’s thesis, Georg-Simon-Ohm-Hochschule, Nürnberg, 30.09.2011.
- [98] K. Trenkel and U. Heinkel, “An Advanced Modular and Portable Test Automation Framework for Practical Use: 25.-27. Februar 2014, Nürnberg, ISBN 978-3-645-50131-6,” in *Embedded World Conference*, vol. Session 8 Beitrag 3.
- [99] K. Trenkel, F. Spittler, H. Rauch, and U. Heinkel, “modTF - ein modulares Framework zur Testautomatisierung: 02.-06. Dezember 2013, Sindelfingen, ISBN 978-8343-2408-5,” in *Embedded Software Engineering Kongress 2013*, pp. 358–367.
- [100] AUTOSAR development cooperation, “Media Pictures: AUTOSAR-components-and-inte.jpg,” 2015. [Online]. Available: http://www.autosar.org/fileadmin/images/media_pictures/AUTOSAR-components-and-inte.jpg
- [101] Patrick Markl, “AUTOSAR Basic Software for CAN-MOST: Body Electronics,” 12.01.2011. [Online]. Available: http://www.automotive-eetimes.com/en/autosar-basic-software-for-can-most-gateways.html?cmp_id=71&news_id=222901315
- [102] AUTOSAR development cooperation, “AUTOSAR Glossary,” 09.12.2011. [Online]. Available: http://www.autosar.org/fileadmin/files/releases/4-0/main/auxiliary/AUTOSAR_TR_Glossary.pdf
- [103] Eclipse Foundation, “Eclipse Project,” 2014. [Online]. Available: <http://www.eclipse.org/>
- [104] Universität Göttingen, “Compiler Integration,” 2012. [Online]. Available: <https://www.trex.informatik.uni-goettingen.de/trac/wiki/CompilerIntegration>
- [105] —, “Plug-In Structure,” 2010. [Online]. Available: <http://www.trex.informatik.uni-goettingen.de/trac/wiki/PluginStructure>
- [106] The Eclipse Foundation, “Eclipse Debug Project,” 2014. [Online]. Available: <http://www.eclipse.org/eclipse/debug/>
- [107] I. Appcelerator, “What is PyDev,” 2014. [Online]. Available: <http://pydev.org/>
- [108] ETSI, “TTCN-3: Extensions: TTCN-3 Performance and Real-time Testing,” Sophia Antipolis Cedex, 06-2014. [Online]. Available: http://www.etsi.org/deliver/etsi_es/202700_202799/202782/01.02.01_60/es_202782v010201p.pdf
- [109] —, “TTCN-3 Extensions: Support of interfaces with continuous signals,” Sophia Antipolis Cedex, 04-2012.
- [110] Juergen Grossmann, Diana Serbanescu, Ina Schieferdecker, “Testing Embedded Real Time Systems with TTCN-3,” pp. 81–90. [Online]. Available:

- http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=4815340&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxppls%2Fabs_all.jsp%3Farnumber%3D4815340
- [111] Fraunhofer Fokus, “TTCN - 3 Quick Reference Card,” 2014. [Online]. Available: http://www.blukaktus.com/TTCN3QRC_viewme.pdf
- [112] C. Willcock, *An introduction to TTCN-3*. West Sussex and England and Hoboken and NJ: J. Wiley, 2005.
- [113] ETSI, “TTCN-3 Performance and Real-time Testing,” Geneva, 07-2010. [Online]. Available: http://www.etsi.org/deliver/etsi_es/202700_202799/202782/01.01.01_60/es_202782v010101p.pdf
- [114] Z. R. Dai, *An approach to model-driven testing: Functional and real-time testing with UML 2.0, U2TP and TTCN-3*. Stuttgart: Fraunhofer-IRB-Verl., 2006.
- [115] Fraunhofer Fokus, “TEMEA - Testspezifikationstechnologie und -methodik für eingebettete Echtzeitsysteme im Automobil,” 2010. [Online]. Available: <http://www.temea.org/>
- [116] Universität Göttingen, “Getting Started,” 2010. [Online]. Available: <http://www.trex.informatik.uni-goettingen.de/trac/wiki/GettingStarted>
- [117] T. Parr, *Language implementation patterns: Create your own domain-specific and general programming languages*. Raleigh and N.C: Pragmatic Bookshelf, 2010.
- [118] Universität Göttingen, “Automation of Refactoring for TTCN-3 Test Specifications,” 2012. [Online]. Available: <http://www.trex.informatik.uni-goettingen.de/svn/trex/web/wiki/pdf/trex-poster2.pdf>
- [119] Terence Parr, “ANTLR (ANother Tool for Language Recognition),” 2014. [Online]. Available: <http://www.antlr.org/>
- [120] Devin Cook, “GOLD Parsing System: Multi-Programming Language, Parser,” 2012. [Online]. Available: <http://www.goldparser.org/>
- [121] Tangient LLC, “Pyparsing Wiki Home,” 2014. [Online]. Available: <http://pyparsing.wikispaces.com/>
- [122] W. Martin Borgert, “TTCN BNF PyParsing,” 07.04.2008. [Online]. Available: <http://people.debian.org/~debacle/ttcn-bnf.html>
- [123] TQ-Group, “Industrie-PC Blue Power: Industrie-PC mit Freescale i.MX35,” 12.12.2013. [Online]. Available: <http://www.tq-group.com/produkte/produktdetail/prod/industrie-pc-blue-power/extb/Main/productdetail/>
- [124] RST Industrie Automation GmbH, “Gamma V: Gamma V - Die Echtzeit-Middleware,” 2012. [Online]. Available: <http://www.rst-automation.com/gamma-v-5>
- [125] —, “INTRODUCING THE GAMMA PLATFORM.” [Online]. Available: <http://www.rst-automation.de/downloadsneu/finish/8-gamma-v/>

53-introducing-the-gamma-platform

- [126] K. Trenkel and U. Heinkel, *Absicherung der Buskommunikation – Bussystemübergreifende Wiederverwendbarkeit von Testfällen*, ser. AmE 2013. VDE-Verlag, 2013.
- [127] K. Trenkel, U. Heinkel, F. Spiteller, and J. Tremmel, “Modular Test Framework - From Component Test to Production Test: 27.-31. Mai 2013, Avignon, France, ISBN 978-1-4673-6375-4,” in *IEEE European Test Symposium*, vol. ETS2013.
- [128] U. Pross, E. Markert, J. Langer, A. Richter, C. Drechsler, and U. Heinkel, “A Platform for Requirement Based Formal Specification: FDL’08 (Forum on Specification & Design Languages), Stuttgart, 23.-25. September 2008,” in *IEEE Catalog Number CFP0826E-USB*, IEEE, Ed., pp. 237–238.

Anhang C.

Abbildungsverzeichnis

2.1. Schematische Darstellung eines Steuergerätes	5
2.2. Bussysteme mit Gateway im Fahrzeug [12, S. 1]	9
2.3. Produktlebenszyklus nach Boston Consultin Group (BCG) [23]	11
2.4. Produktlebenszyklus nach Markus Rentschler [24]	11
2.5. Wasserfall-Modell [25, S. 19]	12
2.6. V-Modell [26, S.33]	13
2.7. W-Modell [26, S.35]	14
2.8. Doppel-Dach-Modell [3, S.14]	14
2.9. Spiralmodell [27]	16
2.10. Rapid Application Development [26, S.44]	16
2.11. Extreme Programming [26, S.41]	17
2.12. Test-Driven Development	18
2.13. Scrum	19
2.14. V-Modell in der Automobilindustrie	20
2.15. Fundamentalen Testprozesses nach ISTQB [25, S.20]	21
2.16. Testpyramide	22
2.17. Model In The Loop – Verbindung des Funktions- und Umgebungs-modelles	25
2.18. Aufbau der HIL-Testumgebung	27
3.1. Beispiel Sequenzdiagrammen von EXAM	39
3.2. Beispiel Simulink Modell	41
3.3. Beispiel AutomationDesk Testablauf	42
3.4. Aufbau des ECU-TEST Applikation Servers [77]	49
3.5. Aufbau des iTestStudio	50
4.1. Allgemeiner Funktion der Middleware	61
4.2. Modulkonzept der modularen Middleware zur Testautomatisierung	78
4.3. Struktur des Testausführungsmoduls auf TTCN-3-Basis	87
4.4. XML-Schema für Testdokumentation Teil 1	89
4.5. XML-Schema für Testdokumentation Teil 2	89
4.6. Struktur der modularen Middleware	93
4.7. Einsatz der Middleware über den Produktlebenszyklus	94

5.1.	AUTOSAR Übersicht [100]	99
5.2.	AUTOSAR Com Stack [101]	100
5.3.	AUTOSAR Com Stack –Testaufbau	101
5.4.	Struktur des Test-Beschreibungs-Modul	104
5.5.	TTCN-3 <i>log</i> -Aufrufen mit Anforderungs-ID	105
5.6.	Eclipse mit TRex-Plugin	112
5.7.	Allgemeine Struktur des Interpreters	113
5.8.	Struktur des Test-Ausführungs-Moduls	117
5.9.	Struktur des TRex-Plugin [118]	118
5.10.	Klassendiagramm der Symboltabelle	120
5.11.	Klassendiagramm der Interpreter-Exceptions	122
5.12.	Übersicht der Interaktion der Komponenten des Interpreters	131
5.13.	Sequenzdiagramm der Ausführung von Testfall 1	132
5.14.	Sequenzdiagramm der Ausführung von Testfall 2 – Performance and Real Time Testing	133
5.15.	Sequenzdiagramm der Ausführung von Testfall 2 – Support of interfaces with continuous signals	134
5.16.	Sequenzdiagramm der Ausführung von Testfall 3	134
5.17.	Datei-Dialog bei Aufruf des Interpreters	138
5.18.	Ausschnitt aus einer HTML-Test-Report Teil1	141
5.19.	Ausschnitt aus einer HTML-Test-Report Teil2	142
5.20.	Darstellung der Signalverläufe auf FlexRay und XCP	145
5.21.	Testreport mit Darstellung der Signalverläufe auf FlexRay und XCP	146
5.22.	Dialog für die Auswahl der modTF-XML-Datei	148
5.23.	Übersicht zur Integration mit dem Requirementsmanagement	150
6.1.	Systemaufbau HIL-Demosystem	154
6.2.	Softwarestruktur HIL-Demosystem	155
6.3.	HTML-Test-Report – Motortest	158
6.4.	AutomationDesk-Anbindung GUI	160

Anhang D.

Tabellenverzeichnis

4.1.	Anforderungsliste – Testerstellung und -entwicklung Teil I	66
4.2.	Anforderungsliste – Testerstellung und -entwicklung Teil II	67
4.3.	Anforderungsliste – Testerstellung und -entwicklung Teil III	67
4.4.	Anforderungsliste – Testerstellung und -entwicklung Teil IV	68
4.5.	Anforderungsliste – Testausführung Teil I	69
4.6.	Anforderungsliste – Testausführung Teil II	69
4.7.	Anforderungsliste – Testausführung Teil III	70
4.8.	Anforderungsliste – Testausführung Teil IV	70
4.9.	Anforderungsliste – Testausführung Teil V	70
4.10.	Anforderungsliste – Test-Reporting Teil I	72
4.11.	Anforderungsliste – Test-Reporting Teil II	72
4.12.	Anforderungsliste – Test-Reporting Teil III	72
4.13.	Anforderungsliste – Test-Reporting Teil IV	73
4.14.	Anforderungsliste – Test-Reporting Teil V	73
4.15.	Anforderungsliste – Testmanagement Teil I	74
4.16.	Anforderungsliste – Testmanagement Teil II	74
4.17.	Anforderungsliste – Testmanagement Teil III	75
4.18.	Anforderungsliste – Testsystemanbindung Teil I	75
4.19.	Gegenüberstellung der Lösungsmöglichkeiten für die Modulschnitt-stellen	76
4.20.	Gegenüberstellung der Testbeschreibungsmethoden – Python, ISO 13209 und TTCN-3	81
4.21.	Anforderungserfüllung der Testbeschreibungsmethoden – Python, ISO 13209 und TTCN-3	84
5.1.	TCI – Mapping der Datentypen von TTCN-3 auf Python	115
5.2.	TTCN-3 Interpreter – Exception für Laufzeitfehler	121
5.3.	TTCN-3 Interpreter – Exception für Flusskontroller	121
5.4.	Package <i>isxml</i>	140

Anhang E.

Listings

5.1.	TTCN-3 Testskript für Testfall 1	106
5.2.	TTCN-3 Testskript für Testfall 2 – Performance and Real Time Testing . .	107
5.3.	TTCN-3 Testskript für Testfall 2 – Support of interfaces with continuous signals	108
5.4.	TTCN-3 Testskript für Testfall 3	109
5.5.	BNF-Darstellung	119
5.6.	PyParsing-Darstellung	119
5.7.	TTCN-3 Interpreter – Funktion <i>module</i>	123
5.8.	TTCN-3 Interpreter – Aufruf der TRI-Funktion <i>triStartTimer</i>	124
5.9.	TTCN-3 Interpreter – Aufruf der TCI-Funktion <i>tliLog</i>	125
5.10.	TTCN-3 Interpreter – Dynamische Einbindung von Port in <i>triMap</i>	127
5.11.	TTCN-3 Interpreter – Schnittstellendefinition eines Ports	128
5.12.	TTCN-3 Interpreter – Methode <i>tliLog</i>	129
5.13.	TTCN-3 Interpreter – Methode <i>_add_result</i>	130
5.14.	TTCN-3 Interpreter – Kommandozeilenoptionen	137
5.15.	Ausschnitt aus einer modTF-XML-Datei Teil1	141
5.16.	Ausschnitt aus einer modTF-XML-Datei Teil2	142
5.17.	Ausschnitt der Methode <i>setVerdict</i> der MKS-Anbindung	144
6.1.	TTCN-3 Testskript für die Funktion <i>bin2hex</i>	153
6.2.	Gamma V Port – <i>send</i>	156
6.3.	TTCN-3 Testskript – Motortest	157

Anhang F.

Lebenslauf

Studium

09/2001 – 16.11.2005 – Studium der Elektrotechnik/Technische Informatik an der Fachhochschule Jena, Diplomthema: “Entwicklung der Hard- und Software für eine universell einsetzbare ETHERNET-Schnittstelle für verschiedene Mikrorechner”, Diplomnote: 1,3

07.03.2008 – 31.03.2010 – Promotionsstudent an der TU Chemnitz an der Fakultät Informatik

Seit 01.04.2013 – Promotionsstudent an der TU Chemnitz an der Fakultät für Elektrotechnik und Informationstechnik

Anstellung

07.11.2005 – 31.12.2007 – bei der SYSTEC electronic GmbH, Greiz als Entwicklungsingenieur im Bereich Softwareentwicklung

01.01.2008 – 30.06.2013 – Wissenschaftlicher Mitarbeiter der TU Chemnitz mit Arbeitsort iSyst Intelligente Systeme GmbH in Nürnberg im Bereich HIL-Tests sowie im Förderprojekt ProTecT Embedded Systems

Seit 01.07.2013 – Mitarbeiter der iSyst Intelligente Systeme GmbH in Nürnberg im Bereich Entwicklung, Förderprojekte und Betreuung studentischer Arbeiten