DIPLOMARBEIT

Unit-Testen in modellbasierter Softwareentwicklung

ausgeführt zur Erlangung des akademischen Grades
eines Diplom-Ingenieurs unter der Leitung von

Univ.Prof. Dipl.-Ing. Dr. Hermann Kaindl

am

Institut für Computertechnik (E384)
der Technischen Universität Wien

durch

Christoph Luckeneder
Matr.Nr. 0927141
Theodor-Körner-Gasse 30/7, 1210 Wien

Wien, am 23. September 2015 _____

Kurzfassung

Im Rahmen dieser Diplomarbeit sollte ein bestehender Prozess zur Verifikation von Software Units verbessert werden. Nach einer eingehenden Beschäftigung mit der Literatur und dem bestehenden Verifikationsprozess wurde der Fokus auf die für Unit-Testen relevanten Subprozesse des Testprozesses eingeschränkt.

Bei den betrachteten Subprozessen handelt es sich um Prozesse, welche einfache Regeln für die Erstellung der Testfälle zur Verfügung stellen. Der große Spielraum bei der Erstellung der Testfälle erschwert unter anderem die Nachvollziehbarkeit durch andere Personen und führt dazu, dass diese nur schwer wiederholbar sind. Um die Wiederholbarkeit zu verbessern, wurde bereits vor einiger Zeit ein modellbasiertes Testautomatisierungsprogramm angekauft, dessen Anwendung sich in der Praxis noch nicht durchgesetzt hat.

Basierend auf einer Analyse der betrachteten Subprozesse und der Herausarbeitung verschiedener Verbesserungsmöglichkeiten, wurde ein Konzept für einen neuen Workflow innerhalb dieser Subprozesse erarbeitet. Dieser Workflow wurde besonders mit Augenmerk auf die Beschleunigung der bisher mit hohem manuellen Aufwand verbundenen Testdurchführung entwickelt.

Da das Konzept das Testen auf der Zielhardware vorsieht, beinhaltet dieses eine Automatisierung eines Vorganges vor dem Flashen des Steuergerätes mit der Testsoftware. Dabei muss der Testrahmen, bei dem es sich um einen vorhergehenden Release des Systems handelt in dem die zu testende Unit eingebettet ist, durch eine direkte Modifizierung der HEX-Datei angepasst werden. Da die notwendigen Modifizierungen von der zu testenden Unit abhängig sind, ist es notwendig, eine darauf abgestimmte Konfigurationsdatei zu erstellen. Im Rahmen der Arbeit wurde zur Entlastung der Tester ein Tool entwickelt und prototypisch implementiert. Dieses Tool nimmt die notwendigen Einstellungen in der Konfigurationsdatei vor und koordiniert die vor dem Flashen notwendigen Modifizierungen.

Der erweiterte Workflow nutzt zudem den Funktionsumfang des Automatisierungsprogramms TPT, insbesondere die zur Verfügung gestellte Funktion zur Simulation von ASCET-Modellen. Die Anwendung der Simulation hat den entscheidenden Vorteil, dass nicht jeder Testlauf auf der Zielhardware durchgeführt werden muss. Das Warten auf die Verfügbarkeit von Zielhardware und der zusätzliche Aufwand für die Integration der zu testenden Unit in einen lauffähigen Programmstand entfällt dabei. Eine erste schnellere Verifizierung der Funktionalität ist dadurch möglich. Da die Simulation der Funktionalität keine ausreichende Verifizierung darstellt, sind abschließende Tests auf der Zielhardware anzuraten. Daher wurden im Rahmen der Arbeit plattformunabhängige Tests eingeführt, welche in der Simulation und auf den bestehenden Open-Loop Testplätzen gleichermaßen ausgeführt werden können. Ein vom ersten unabhängiges zweites Tool bietet dazu einerseits ein Rahmenwerk zur Erstellung plattformunabhängiger Testfälle und andererseits unterstützt es den Tester bei der testspezifischen Konfigurierung des TPT-Files um diese Plattformunabhängigkeit zu erzielen.

Durch den verstärkten Einsatz des Automatisierungsprogrammes TPT und der beiden Tools im erweiterten Workflow ist zu erwarten, dass die Dauer des Unit-Testvorganges, welche bei manuellen Testen erfahrungsgemäß stark schwanken kann, kürzer, besser abschätzbar und planbarer wird.

Abstract

In the course of this master thesis, the current process for the verification of software units was subject of being improved. Based on a literature research and an analysis of the current verification process, the focus was on two sub-processes for unit testing.

These sub-processes provide basic rules for the development of test cases. These rules leave the test personnel a wide scope for the test case design, so that the test cases are difficult to be replicated and repeated by other test personnel. In order to improve the repeatability, the model-based test-automation software TPT has been procured some time ago, but it is rarely used in practice in the context of these sub-processes.

Based on an analysis of these sub-processes, several possible improvements were found and a concept for a new workflow within these sub-processed was defined. The focus was on less time consumption for unit testing, in particular its previously time-consuming manual steps.

This concept includes an automation of a part of the workflow, before the test software is being flashed on the Engine Control Unit (ECU). In this step, the test-frame, which consists of an earlier software release where the unit under test is embedded, must be adapted. This modification is performed directly on the HEX-file to gain access to the inputs of the unit. Since it is specific to the given unit, a specific configuration file must be created. In the course of this work, a tool was developed and implemented to support the tester. This tool creates the necessary settings in the configuration file and coordinates the modification process.

The proposed workflow makes use of the functionality of the test-automation software TPT, especially its functions for simulation of ASCET models. Tests on the simulator have the advantage over tests on the target hardware that it is not necessary to wait for availability of this hardware and to integrate the unit into the earlier software release. Thus a first quick verification is possible. Because the verification based on tests on the simulator alone is insufficient, platform-independent tests were introduced. These tests are designed to run both on the simulator and the open-loop test setting. A second tool provides a framework for the setup of the TPT file and supports the tester during configuration of the test-dependent settings in the TPT file to achieve platform-independence.

Through the usage of the test-automation software TPT and of the two new tools in the extended workflow, a shorter and more predictable duration for the overall unit-test process is expected.

Danksagung

An dieser Stelle bedanke ich mich bei jenen Personen, die mich bei der Erstellung dieser Diplomarbeit unterstützt haben.

Ein besonderer Dank gilt meinen beiden Diplomarbeitsbetreuern Herrn Michael Dübner seitens der Robert Bosch AG und Univ.Prof. Dipl.-Ing. Dr. Hermann Kaindl seitens der TU Wien, welche die Arbeit erst möglich gemacht haben, für die Unterstützung bei der Ausarbeitung.

Ich bedanke mich bei all meinen Kollegen in der Robert Bosch AG, die mir immer mit Rat und Tat zur Seite standen. Besonderer Dank gilt dabei Herrn Martin Korinek und Herrn Michael Mandl, welche mir in unzähligen Diskussionen immer wieder neue Denkanstöße geliefert haben.

Herzlichen Dank bringe ich meinen Eltern entgegen, die mein Studium ermöglicht und mich dabei nicht nur moralisch unterstützt haben.

Ein großer Dank gebührt meiner langjährigen Freundin Kerstin, welche mir in dieser Zeit immer zur Seite stand.

Inhaltsverzeichnis

Abkürzungen

ASAM	Association for Standardization of Automation and Measuring Systems
ASCII	American Standard Code for Information Interchange
AUTOSAR	Automotive Open System Architecture
BC	Base Component
CAN	Controller Area Network
CIM	Computation Independent Model
DLL	Dynamic Link Library
ECU	Electronic Control Unit
FC	Functional Component
FCI	Functional Component Implementation Test
FCF	Functional Component Functional Test
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
MDA	Model-driven Architecture
MDE	Model-driven Engineering
MDF	Measurement Data Format
MDSD	Model-driven Software Development
PIM	Platform Independent Model
PSM	Platform Specific Model
PVER	Program Version
SUT	System Under Test
TDD	Test Driven Development
UML	Unified Modeling Language

1 Einleitung

Dieses Kapitel vermittelt die Motivation für die Arbeit und gibt anschließend einen kurzen Überblick über den Inhalt der folgenden Kapitel.

1.1 Motivation

Der stetig steigende Kostendruck und die Anforderung der Kunden, immer schneller auf Änderungswünsche reagieren zu können, haben die Softwareentwicklung in den letzten Jahren grundlegend verändert. Bislang blieb die Entwicklung von sicherheitskritischer Software zumindest weitgehend davon verschont, der Kostendruck und die Time to Market spielen allerdings eine stetig zunehmende Rolle.

Für die Entwicklung von Standard-Software existieren bereits seit einigen Jahren Prozesse, welche helfen, diese Herausforderungen zu meistern. Alle diese Prozesse haben gemeinsam, dass die einzelnen Schritte des Verifikationsprozesses stark optimiert und größtenteils automatisiert sind.

Um wichtige Schritte in Richtung eines effizienteren Testprozesses machen zu können und gleichzeitig die sicherheitskritische Integrität des Systems zu wahren, war es zunächst notwendig, die bestehenden Subprozesse bezüglich dem Unit-Testen zu analysieren und das Verbesserungspotenzial ausfindig zu machen. Um das Testen der Software Units besser und effizienter zu gestalten, setzt diese Arbeit genau dort an und versucht mit der Einführung eines neuen Workflows und der Unterstützung durch Tools einen Schritt in diese Richtung zu gehen.

1.2 Überblick

Das Kapitel Grundlagen gliedert sich in vier Teilbereiche zu den Themen Softwareentwicklung, Model-driven Software Development, ISO26262 und Verifikationsmethoden. Im Unterkapitel Softwareentwicklung wird auf allgemeine Grundlagen der Softwareentwicklung eingegangen. Im speziellen wird das V-Modell der Softwareentwicklung erläutert. Dabei liegt der Fokus auf der Erläuterung der für das Testen der Software relevanten Teilaspekte. Eine darauf folgende Beschreibung der einzelnen Testphasen soll ein einheitliches Verständnis für den späteren Verlauf der Arbeit sicherstellen. Im letzten Teil dieses Unterkapitels werden die Begriffe Verifikation und Validierung kurz erläutert und die Unterschiede aufgezeigt. Das darauf folgende Unterkapitel

Model-driven Software Development vermittelt die im Rahmen der Arbeit notwendigen Grundlagen der modellbasierten Softwareentwicklung und geht auf potentielle Vorteile gegenüber der klassischen Softwareentwicklung ein. Da die Arbeit im Umfeld der Automobilindustrie stattfindet, wird auf die diesbezüglich zentrale Norm ISO26262 eingegangen. Nach einer kurzen allgemeinen Einführung in das Thema wird eine mögliche normkonforme Umsetzung eines modellbasierten Entwicklungsprozesses aufgezeigt. Da Normen den aktuellen Stand der Technik festhalten und dieser rechtliche Relevanz hat, wird im letzten Abschnitt zur ISO26262 auf rechtliche Aspekte eingegangen. Dabei wird herausgehoben, was diese Norm so besonders gegenüber anderen Normen macht und welche rechtlichen Probleme damit verbunden sind. Der Schlussteil des Kapitels befasst sich mit Verifikationsmethoden. Zuerst wird auf verschiedene Methoden zur Verifikation der Implementierung gegenüber der Spezifikation, deren Einsatzgebiete und Vor- bzw. Nachteile, eingegangen, um anschließend zu begründen, warum sich die restliche Arbeit mit dem Thema Testen beschäftigt.

Da die Arbeit im Umfeld eines Unternehmens stattfindet bzw. das Ergebnis später dort zum Einsatz kommen soll, wird in Kapitel 3 über den Ausgangspunkt der Arbeit, auf unternehmensspezifische Grundlagen, auf denen die Arbeit aufsetzt, eingegangen. Der diesbezüglich erste Teil befasst sich mit der zu testenden Software und gibt einen Überblick über dessen inneren Aufbau ohne zu tief ins Detail zu gehen. Darauf folgend werden in Unterkapitel 3.2 die zur Anwendung kommenden Tools benannt und eine kurze Zusammenfassung der Funktionalität gegeben. Bevor in Unterkapitel 3.4 auf den Aufbau der Testplätze eingegangen werden kann, wird in Unterkapitel 3.3 erläutert wie der Testrahmen für Unittests auf der Zielhardware aussieht und welche Modifizierungen am lauffähigen Programm dafür vorgenommen werden müssen.

In Kapitel 4 über den Stand der Technik wird näher auf das Thema Testen eingegangen. Das Unterkapitel 4.1 gibt einen Einblick, welche Gründe für den Einsatz von Testautomatisierung sprechen und bei welchen Tests sie idealerweise eingesetzt werden. Darauf folgend werden in Abschnitt 4.2 nach einer kurzen theoretischen Einleitung bezüglich der Klassifizierung von Testfällen verschiedene Methoden zur Testfallerstellung vorgestellt. Das Kapitel schließt mit einer Besprechung der Möglichkeiten, wie Testen in den Designprozess integriert werden kann.

Kapitel 5 befasst sich mit den für das Unit-Testen relevanten Subprozessen des Testprozesses. Nach einer kurzen Beschreibung der Ziele und des Workflows wird auf das Verbesserungspotenzial dieser Subprozesse eingegangen.

Um die Arbeitsweise der von ASCET zur Verfügung gestellten Simulationsumgebung besser zu verstehen, wurde eine Analyse durchgeführt. Kapitel 6 beschreibt, wie die Implementierung und Ausgangsbasis für die Simulation der Funktionalität am Hostrechner aus dem Modell abgeleitet werden. Danach wird begründet, warum Tests auf der Zielhardware zusätzlich zu empfehlen sind.

Im Kapitel zur Erweiterung der bestehenden Subprozesse wird zunächst dargelegt, welche Ziele dabei verfolgt wurden. Im dazugehörigen Unterkapitel 7.2 wird der neue Workflow mit dem Einsatz der Simulation beschrieben. Das Kapitel schließt mit der Festlegung des Testrahmens und einer Erläuterung zur Anwendbarkeit dieses Testrahmens für Tests auf der Zielhardware.

Das Kapitel 8 geht auf die Umsetzung der im Rahmen der Arbeit prototypisch implementierten Tools ein. Dabei wird jeweils ein typischer Use Case aufgezeigt und veranschaulicht, welche Schnittstellen zu anderen Programmen bzw. Dateien notwendig sind. Zusätzlich wird auf mehrere mögliche Umsetzungen der Plattformunabhängigkeit eingegangen und die Auswahl begründet.

Die Arbeit schließt mit einer kurzen Zusammenfassung und bietet einen Ausblick auf mögliche Weiterentwicklungen.

2 Grundlagen

Dieses Kapitel gibt eine kurze Einführung in das Themengebiet der Softwareentwicklung (SW-Entwicklung) und Testen von Software. Da die Arbeit im Umfeld einer modellbasierten Softwareentwicklung stattfindet, werden diesbezüglich einige Grundlagen vermittelt.

2.1 Softwareentwicklung

Besonders bei hardwarenahen Systemen wie es bei eingebetteten Systemen (Embedded Systems) der Fall ist, ist das Software-Design nur als Teilbereich des übergeordneten Systemdesigns zu betrachten. In vielen Fällen werden durch den Auftraggeber eines Embedded Systems lediglich die Systemanforderungen formuliert. Ob eine Anforderung durch Hardware, Software oder einer Zusammenarbeit von Hardware und Software realisiert wird, muss danach durch den Auftragnehmer entschieden werden. Bedingt durch immer günstigere und leistungsfähigere Standard-Hardwarekomponenten ist es möglich, den Hardwareteil des Systems so auszulegen, dass die Ressourcen für spätere Erweiterungen ausreichen. Der systementwickelnden Firma wird damit in vielen Fällen ermöglicht, neue Anforderungen über die Software zu realisieren ohne die Hardware ändern zu müssen. Die geringeren Kosten für Software-Änderungen gegenüber Hardware-Änderungen zur Implementierung einer neuen Systemanforderung sprechen ebenfalls für den vermehrten Einsatz von Software in Embedded Systems.

Schnell auf Kundenwünsche reagieren zu können und dabei die notwendige Sicherheit des Gesamtsystems nicht zu vernachlässigen, stellt alle an der SW-Entwicklung beteiligten Personen vor große Herausforderungen. Um diese zu meistern, wurden im Laufe der Zeit mehrere Vorgehensmodelle entwickelt. Dieses Unterkapitel gibt eine kurze Einführung in ein klassisches Vorgehensmodell.

2.1.1 V-Modell der SW-Entwicklung

Es gibt verschiedene Vorgehensweisen der Softwareentwicklung und das in Abbildung 2.1 abgebildete V-Modell stellt somit nur eine von vielen Möglichkeiten dar. Diese Arbeit bezieht sich aus zwei Gründen auf dieses Vorgehensmodell. Zum einen wird dieses im für die Automobilindustrie führenden Standard ISO26262 [ISO11] angeführt und zum anderen in der Firma Robert Bosch AG zur Entwicklung von sicherheitskritischer Software eingesetzt. Dabei finden die einzelnen Schritte im V-Modell nicht streng aufeinander folgend statt. Wird zum Beispiel bei einer

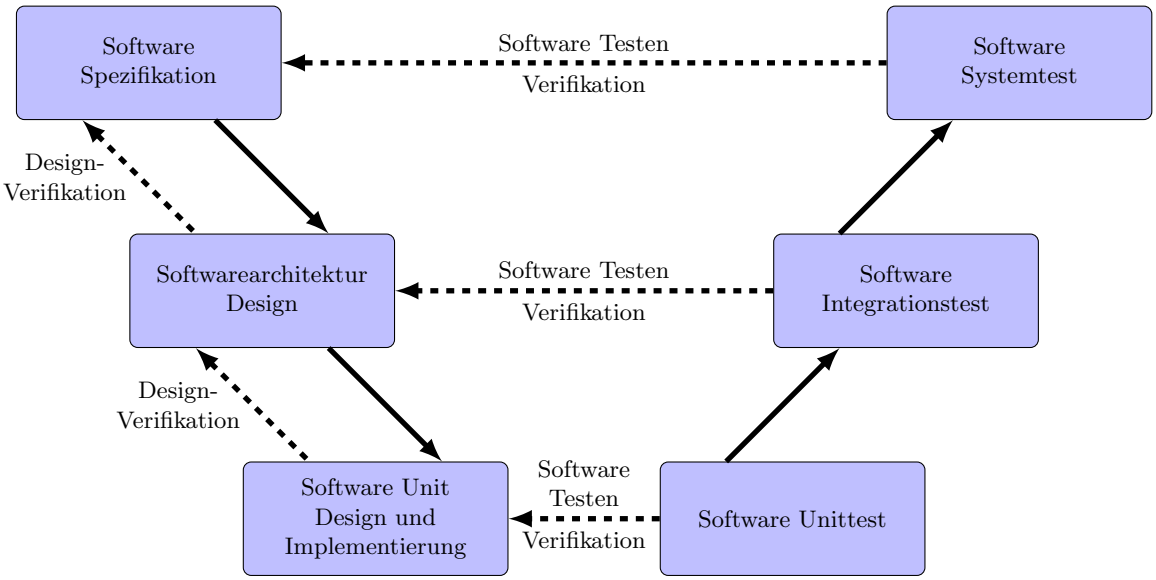

Abbildung 2.1: V-Modell, gezeichnet in Anlehnung an das Referenzmodell der Softwarentwicklung [ISO11, Figure 2].

Design-Verifikation ein Missstand aufgedeckt, ist es in vielen Fällen notwendig, den vorhergehenden Schritt im Design zu wiederholen bzw. das Ergebnis zu überarbeiten. Im Idealfall sind Änderungen nur bezüglich des vorhergehenden Schrittes notwendig.

In der Softwareentwicklung hat sich herausgestellt, dass Testentwickler Spezifikationen aus einem anderen Blickwinkel als SW-Designer betrachten. Bei entsprechend frühzeitigem Beginn der Testentwicklung können die Rückmeldungen der Testentwickler für die SW-Designer hilfreich sein. Frühestmögliche Testentwicklung ermöglicht somit nicht nur eine kürzere Produktentwicklungszeit (Time to Market), sondern bietet neben den Design-Verifikationen eine zusätzliche Möglichkeit Fehler im SW-Design frühzeitig zu entdecken. Aus diesen Gründen hat es sich bei der Entwicklung von Software weitgehend durchgesetzt, dass ein Teil der Tests parallel zum SW-Design entwickelt werden.

2.1.2 Unit-, Integrations-, System- und Akzeptanztest

Unabhängig vom Vorgehen beim Testen sind im V-Modell drei unterschiedliche Arten von Tests vorgesehen. Aufgrund der stetig steigenden Komplexität von Software kommt das Prinzip „devide and conquer“ zur Anwendung. Wenn hier von Unit-, Integrations- und Systemtest gesprochen wird, sind diese jeweils im Kontext der Softwareentwicklung zu betrachten. Einen Spezialfall stellt der Akzeptanztest dar, da er nicht zwingend Teil der Softwareentwicklung ist. Je nachdem ob es sich um ein reines Software- oder ein Hardware/Software-Projekt handelt ist der Akzeptanz-Test auf der Ebene der Software oder darüber anzusiedeln. Aus diesem Grund wurde der Akzeptanztest nicht in diese Version des V-Modells einbezogen.

Unittest

Unit wird der kleinste Teil der Softwarearchitektur genannt, für den eine gesonderte Spezifikation vorhanden ist und welcher für sich getestet werden kann [HL14]. Dabei spielt es keine

Rolle, wie die Unit durch den Entwickler implementiert wird. Eine Unit kann somit die von ihr geforderte Funktionalität durch die Verwendung von mehreren Funktionen, bzw. Klassen bei objektorientierten Programmiersprachen, und deren Interaktionen erzielen. In [Vig10] und [Som12, Kap. 2.2.3] werden Unittests als alle vom Entwickler durchgeführte Tests definiert und gegenüber den Integrations- und Systemtests abgegrenzt. Ob das Testen der Unit gegen die Spezifikation vom Entwickler oder einer anderen Person durchgeführt wird, sollte jedoch an den jeweiligen SW-Designprozess und Testprozess angepasst werden. Daher bevorzugt und verwendet der Autor die Definition des German Testing Board [HL14]. Wonach jeglicher Test der Unit einen Unittest darstellt, unabhängig davon, wer den Test erstellt und durchführt.

Integrationstest

Der Integrationstest, welcher üblicherweise nach einem ausgiebigen Unittest durchgeführt wird, soll sicherstellen, dass die einzelnen Units über deren Schnittstellen richtig miteinander kommunizieren. Dabei wird nicht nur die richtige Implementierung, sondern indirekt auch die Spezifikation der Schnittstellen getestet. Die zu integrierenden Units werden dazu in das bestehende Subsystem eingefügt und offene Schnittstellen zu noch nicht implementierten Units geschlossen. Der Testaufwand für den Integrationstest ist somit sehr stark von der Anzahl der Abhängigkeiten zwischen den Units und damit von der Systemarchitektur abhängig.

Systemtest

Der im V-Modell als letztes angeführte Systemtest soll sicherstellen, dass durch die Implementierung die Software-Requirements erfüllt werden, bevor die Software einem Akzeptanz-Test unterzogen bzw. die Software in das System integriert wird. Dabei sollte nicht nur die Einhaltung der funktionalen Anforderungen, sondern auch der nicht-funktionalen Anforderungen überprüft werden.

Akzeptanztest

Der Akzeptanztest erfolgt im Rahmen der Abnahme durch den Kunden. Je nachdem, ob es sich um ein reines Software- oder ein Hardware/Software-Projekt handelt, ist der Akzeptanz-Test der Softwareentwicklung oder der darüber liegenden Systementwicklung zuzuschreiben. Dieser Test ist notwendig, da es im Laufe der Erstellung der Systemanforderungen zu Missverständnissen zwischen Kunden und Auftragnehmer kommen kann.

2.1.3 Verifikation & Validierung

Unter Verifikation und Validierung versteht man Vorgänge in der SW-Entwicklung, welche sicherstellen sollen, dass die Software sowohl der Spezifikation als auch den Kundenvorstellungen entspricht. Sie sind damit Teil der Qualitätssicherung. Um differenziertere Aussagen treffen zu können, wurden im Rahmen der Qualitätssicherung die beiden Begriffe wie folgt definiert:

Verifikation

Doing things right. Bei der Verifikation wird die Software oder das Teilsystem der Software gegen die in der entsprechenden Spezifikation festgelegten Anforderungen geprüft um sicherzustellen, dass diese erfüllt werden.

Validierung

Doing the right things. Die Validierung ist ein Prozessschritt, bei dem festgestellt wird ob das Softwareprodukt die für den beabsichtigten Gebrauch benötigten Anforderungen erfüllt.

Da der Kunde meist mit einem zu lösenden Problem an die Systementwicklung herantritt, muss sichergestellt werden, dass das entwickelte System dieses Problem tatsächlich löst. Dazu enthält die Spezifikation des Systems die Anforderungen an das System um das bestehende Problem lösen zu können. In einem solchen Dokument wird absichtlich nicht darauf eingegangen, wie diese Anforderungen erfüllt werden. Diese Spezifikation dient als Grundlage für die weitere Entwicklung und schlussendlich für die Implementierung des Systems. Die Verifikation überprüft anders als die Validierung nicht, ob das System das Kundenproblem tatsächlich löst, sondern ob die Anforderungen der Spezifikation erfüllt werden. Da es bei der Erstellung der Anforderungen zu Fehlern kommen kann, garantiert eine positive Verifizierung der Anforderungen nicht zwangsläufig die Lösung des Kundenproblems. Erst die zusätzliche Validierung zeigt auf, ob das Kundenproblem tatsächlich gelöst wurde. Eine anschauliche Darstellung der abstrakten Beschreibungen bietet Abbildung 2.2.

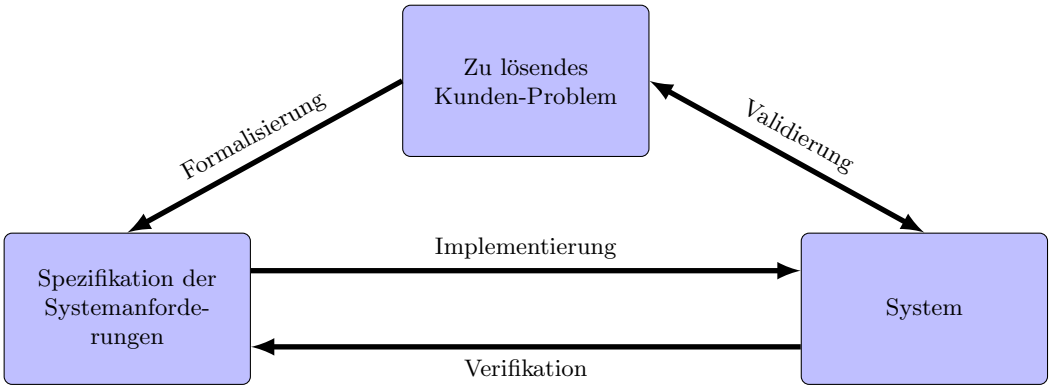

Abbildung 2.2: Veranschaulichung des Unterschiedes zwischen Verifikation und Validierung. Gezeichnet nach [8]

2.2 Model-driven Software Development

Dieses Unterkapitel behandelt das Thema Model-driven Software Development (MDSD) und gibt einen kurzen Überblick über die Funktionsweise sowie Vor- und Nachteile dieser Softwareentwicklungstechnik. Synonym zu MDSD wird oftmals Model-driven Engineering (MDE) verwendet, wobei MDE nahe legt, dass es auch für Bereiche außerhalb der Softwareentwicklung verwendet werden kann. Im Weiteren werden MDE und MDSD, soweit es sich im Kontext der Softwareentwicklung befindet, synonym verwendet.

2.2.1 Allgemeines zu MDSD

Als Model-driven Software Development (MDSD) wird der Vorgang bezeichnet, bei dem die Software als abstraktes Modell beschrieben wird und dieses Modell anschließend durch spätere Transformation in eine konkrete Implementierung übersetzt wird [FR07]. Die Herausforderung

für den Entwickler verlagert sich dabei von der Aufgabe des Programmierens hin zum Erstellen der Modelle [Som12]. Verfechter von MDSD argumentieren, dass die Modellierung die Abstraktionsebene anhebt und somit der Entwickler nicht mit den Eigenarten der Zielplattform bzw. einer Programmiersprache belastet wird [Som12].

Von MDSD unterschieden werden muss jedoch Model-driven Architecture (MDA). Dabei handelt es sich um ein von der Objekt Management Group entwickeltes Softwareparadigma [Som12] zur Trennung von Spezifikation der Software und ihrer Implementierung [Hie07]. Der Fokus von MDA wurde dabei auf die Entwurfs- und Entwicklungsphase gelegt. Im Vergleich zu MDA beinhaltet MDSD zusätzlich die Bereiche der Softwareprozesse zur modellbasierten Entwicklung, des modellbasierten Requirements Engineering und das Testen auf Modellbasis [Som12]. Da MDA ein Bestandteil von MDSD ist, wird im Folgenden auf die Abstraktionsebenen von MDA eingegangen:

Computation Independent Model, CIM

Das CIM dient zur abstrakten Modellierung des Gesamtsystems in seinem Anwendungsbereich. In diesem Modell wird viel mehr die Umwelt und die externe Umgebung des Systems spezifiziert als das System selbst [Hie07]. Damit lässt sich auch erklären, warum CIMs häufig als Domainmodell oder Geschäftsmodell bezeichnet werden [Som12].

Platform Independent Model, PIM

Auf diesem Level der Abstraktion werden bereits Details der Implementierung in das Modell eingebracht. Im PIM wird die Architektur des Systems mittels statischer Strukturen festgelegt. Dies kann unter anderem durch ein Klassendiagramm, welches die verwendeten Klassen und deren Attribute, Methoden und Datentypen festlegt, erfolgen.

Platform Specific Model, PSM

Gemäß [Som12] muss für jede Plattform ein separates PSM erstellt werden, weil in dieser Abstraktionsebene plattformabhängige Details zum Modell hinzugefügt werden. Theoretisch könnten mehrere Schichten an PSM existieren, wobei in jeder Schicht neue plattformabhängige Details hinzugefügt werden. Dies bietet sich besonders dort an, wo sich Plattformen teilweise ähneln, wie es bei einer typischen Produktlinie (Product Line) der Fall ist.

In Abbildung 2.3 werden die Abstraktionsebenen (blau) nochmals aufgegriffen und in Zusammenhang gebracht. Die Transformationen von einer höheren Abstraktionsebene auf eine niedrigere werden durch Generatoren (grün) durchgeführt. Je nachdem in welcher Ebene ein Generator zum Einsatz kommt, fließen bei der Entwicklung des Generators Details der Domain, plattformabhängige Details oder Details der Programmiersprache mit ein.

Die Grenzen zwischen den einzelnen Abstraktionsebenen sind leider nicht immer klar zu identifizieren. Ebenso existiert eine Adaption von MDA, in welcher der Code direkt aus dem PIM erzeugt wird [Som12].

2.2.2 Nutzen von MDSD

Einige der hier genannten Vorteile bzw. Ziele von MDSD leiten sich aus der Definition von MDA ab, andere ergeben sich aus der praktischen Anwendung.

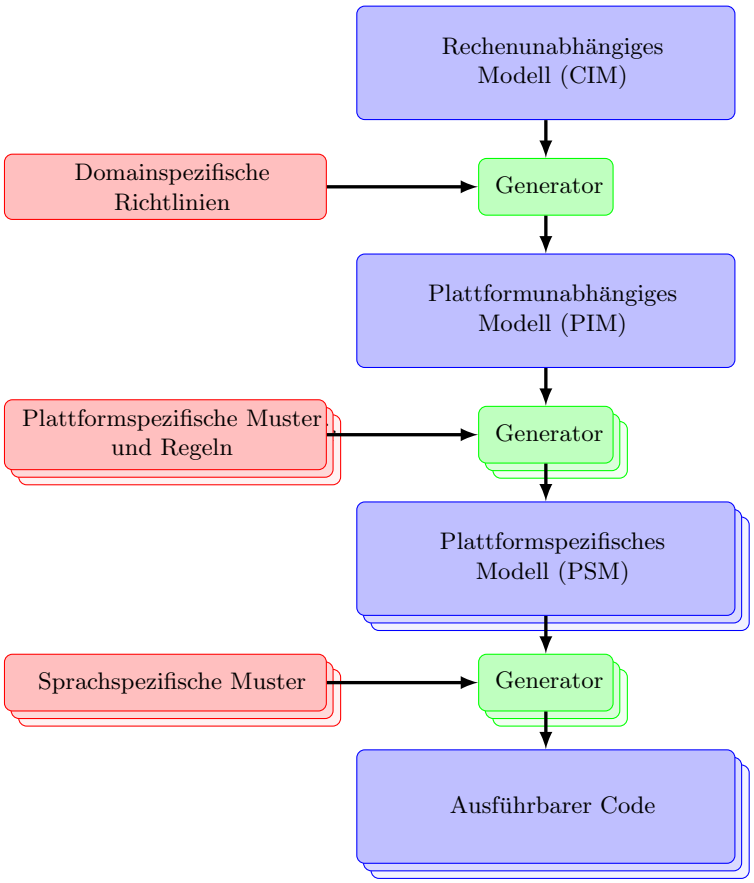

Abbildung 2.3: MDA-Transformationen; Angelehnt an [Som12, Abb. 5.19]

Höhere Abstraktionsebene

MDSD ermöglicht dem Entwickler durch seine höhere Abstraktionsebene und der damit verbundenen Vernachlässigung von Implementierungsdetails gegenüber Programmiersprachen, einen besseren Überblick über die Software und damit die Fokussierung auf die Funktionalität. Mit Hilfe von Generatoren wird aus dem Modell die Implementierung generiert. Dazu werden durch den Generator vorgefertigte Bausteine zusammengefügt.

Architektur

Je nachdem auf welchem Level der Abstraktion der Entwickler sein Modell erstellt, muss er sich keine Gedanken über die Architektur machen, wie das beim CIM der Fall ist, bzw. kann sich durch die Vernachlässigung der Implementierungsdetails bei Entwicklungen auf darunterliegenden Abstraktionsebenen wie PIM und PSM auf die Architektur konzentrieren. Wird das PIM aus dem CIM generiert, muss allerdings ebenso Designarbeit für die Architektur aufgewendet werden. Diese verlagert sich hin zur Erstellung des Generators zwischen CIM und PIM. Der Generator gibt somit einen universell gültigen Rahmen für die Architektur vor. Dadurch wird einerseits sichergestellt, dass die Architektur in der Software konsequent eingehalten wird und andererseits, dass diese schneller angepasst werden kann, da sie zentral verwaltet wird.

Wiederverwendung und Wiederverwendbarkeit

Wiederverwendung spielt in der Softwareentwicklung eine große Rolle, da sie die Softwareentwicklung erheblich beschleunigen kann. Die Wiederverwendbarkeit vorausgesetzt, werden zu diesem Zweck fertige Programmteile, z.B. Funktionen, anstatt Neuentwicklungen eingesetzt. In Bezug auf MDSD lässt sich dieser Gedanke auf die Generatoren erweitern, indem diese für mehrere Anwendungen definiert werden. Die Kosten der Erstellung der Generatoren können somit auf mehrere Projekte aufgeteilt werden.

Plattformunabhängigkeit

Mit Hilfe der Generatoren ist es möglich, ein plattformunabhängiges Modell für mehrere Plattformen zu verwenden. Soll eine neue Plattform unterstützt werden, so muss nur der entsprechende Generator geschrieben oder ein Generator einer anderen Plattform adaptiert werden. Leider ist es häufig der Fall, dass das Modell bereits plattformabhängig ist. Ist dies der Fall, wird bei der Portierung einer Anwendung auf eine andere Plattform eine Anpassung des Modells notwendig.[Hie07]

Zeit- und Kostenersparnis

MDSD wird von vielen Firmen eingesetzt, weil dadurch erhebliche Kosten- und Zeitersparnisse bei der Entwicklung von Funktionen mit hoher Komplexität erhofft werden. Dies ist auch durchaus der Fall, allerdings sollte der hohe Zeitaufwand und die damit verbundenen hohen Kosten der Einführung nicht unterschätzt werden. So sind im ersten Jahr nach der Einführung Kostensteigerungen zu erwarten. Erst nachdem das System etabliert ist, kommen die Ersparnisse richtig zum Tragen und es können laut der Studie [BKZK11] im Durchschnitt 27 % der Kosten und 36 % der Zeit eingespart werden.

All diese hier angeführten Vorteile des MDSD und die zunehmende Toolunterstützung tragen zur immer größeren Verbreitung in der Industrie bei. In der ISO26262 wird diesem Umstand bereits Rechnung getragen, in dem neben der Programmiersprache (Programming Language) auch von einer Modellierungssprache (Modelling Language) die Rede ist. Dabei sollten aber keinesfalls die zusätzlichen Kosten und die Probleme bei der Einführung unterschätzt werden.

2.3 ISO26262

Der Standard ISO26262 befasst sich mit der funktionalen Sicherheit von Straßenfahrzeugen bis zu einem max. Gesamtgewicht von 3,5t. Er wurde im November 2011 von der International Organization for Standardization herausgebracht und stellt seit diesem Zeitpunkt die zentrale Norm der Automobilindustrie dar.

Dieses Unterkapitel gibt zunächst einige allgemeine Informationen zum Standard, um danach eine Möglichkeit der standardkonformen Softwareentwicklung mit MDSD aufzuzeigen. Der letzte Abschnitt befasst sich mit Kritik aus rechtlicher Sicht.

2.3.1 Allgemeines zu ISO26262

Unter funktionaler Sicherheit wird in der ISO26262 die Vermeidung von Fehlfunktionen oder Systemausfällen durch systematische Fehler oder Komponentenausfälle verstanden. Ziel der definierten Maßnahmen ist es, bevor die Produktion des Produktes begonnen hat, alle Gefahren und Risiken zu erkennen [Reu14] und nach dem aktuellen Stand der Technik angemessene Gegenmaßnahmen vorzunehmen. Ein Großteil der Anforderungen in der ISO26262 beschreiben die Dokumentationspflichten und systematische Vorgehensweisen beim Entwicklungsprozess. Ein großes Augenmerk wird dabei auf die Rückverfolgbarkeit gelegt.

Dass dieser Ansatz nicht ganz neu ist, sieht man anhand der IEC61508. Ausgehend von der IEC61508 wurde die ISO26262 als Anpassung für die Automobilindustrie entwickelt. Dabei unterscheiden sich die beiden Standards in wesentlichen Punkten. Neu ist, dass das Endprodukt, also das Fahrzeug, im Fokus aller Betrachtungen steht und nicht wie bisher die einzelnen Systeme. Bislang standen Hardware, Software und die Interaktion zwischen Systemen im Zentrum der sicherheitstechnischen Betrachtungen [Reu14]. Die ISO26262 bezieht den gesamten Sicherheitslebenszyklus, welcher Management, Entwicklung, Produktion, Betrieb, Service und Außerbetriebnahme beinhaltet [Hil12, Kap. 3], mit ein.

Die ISO26262 ist in zehn Dokumente unterteilt, wobei sich einzelne Teile auf jeweils spezielle Phasen des Lebenszyklus beziehen und andere allgemeine Gültigkeit besitzen. Die Gliederung ist wie folgt:

- Part1: Vocabulary
- Part2: Management of functional safety
- Part3: Concept phase
- Part4: Product development at the system level
- Part5: Product development at the hardware level
- Part6: Product development at the software level
- Part7: Production and operation
- Part8: Supporting processes
- Part9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analysis
- Part10: Guideline on ISO 26262

Vor allem Part6 und die allgemeinen Teile sind für die Softwareentwicklung und damit für diese Arbeit von Bedeutung. Jedoch sollten die Schnittstellen zu anderen Phasen nicht ganz außer Acht gelassen werden. So wird in Part6 explizit darauf hingewiesen, dass die SW-Entwicklung mit der System- und Hardware-Entwicklung koordiniert werden muss.

2.3.2 ISO26262-konformes MDSD

Die modellbasierte Entwicklung von Software ist in der Automobilindustrie weit verbreitet [Kur13, Kap. 1.1] [AWS14, Table 12] daher wird dies in der ISO speziell berücksichtigt. So ist des Öfteren von einer Modellierungssprache (Modelling Language) die Rede. An diese werden weitgehend dieselben Anforderungen wie an die Programmiersprache (Programming Language) gestellt.

Da ein Standard üblicherweise sehr generisch verfasst ist und dies auch auf die ISO26262 zutrifft, gibt es mehrere Möglichkeiten diesen umzusetzen. Ein im Rahmen der Arbeit [Kur13] vorgestellter Referenzworkflow (Abbildung 2.4), bezüglich der Verifikation von mit MDSD entwickelter Software, wird im Folgenden kurz erläutert.

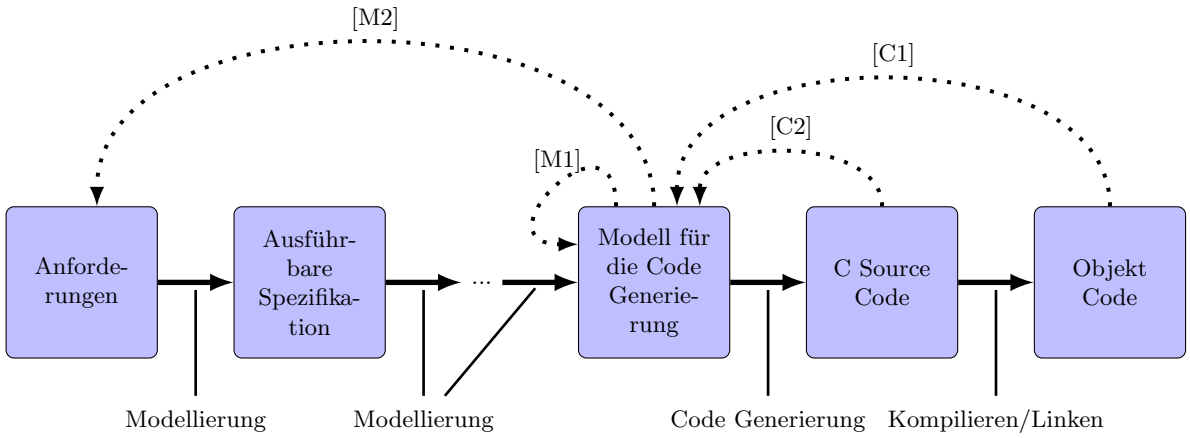

Abbildung 2.4: Workflow nach [Kur13]

[M1] Review und automatische statische Analyse auf Modellebene

Die modellierte Unit wird einem Review unterzogen. Dabei sollte überprüft werden, ob das Modell den Modellierungsrichtlinien wie zum Beispiel der Namenskonvention entspricht und keine unerlaubten Konstrukte zum Einsatz kommen. Wo möglich, ist der Einsatz eines statischen Analyse Tools dem manuellen Review vorzuziehen.

[M2] Funktionale Unit- und Integrationstests auf Modellebene

Aus der Spezifikation der Unit systematisch abgeleitete Testfälle werden auf dem Modell zur Ausführung gebracht. Ziel dieses Verifizierungsschrittes ist es, eine objektive Aussage darüber treffen zu können, ob die Unit die Spezifikation erfüllt und kein unerwünschtes Verhalten zeigt. Wurde jede Unit auf diese Weise überprüft, wird das System schrittweise integriert. Bei jeder Integrationsstufe wird mit zusätzlichen Integrationstests überprüft, ob das Verhalten im Zusammenspiel der Spezifikation entspricht.

Am Ende der beiden Verifikationsschritte [M1] und [M2] erhält man ein gegen die Spezifikation verifiziertes Modell.

[C1] Back- to Back-Test zwischen Modell und Objekt-Code

Der Referenzworkflow sieht vor, dass das Modell und der Objekt-Code mit identischen Testfällen stimuliert werden. Die Auswertung erfolgt dann über den direkten Vergleich des jeweiligen Verhaltens. Der Referenzworkflow gibt aber keine Information über die Art bzw. das Erstellungsverfahren der Testfälle.

[C2] Zusätzliche Maßnahmen zur Vermeidung unerwünschter Funktionalität

In diesem Schritt wird der generierte C-Code auf unerwünschte Funktionalität hin überprüft. Der Workflow schlägt dazu Coverageanalysen sowohl auf C-Code- als auch Modellebene vor. Anschließend werden die beiden Werte miteinander verglichen. Gibt es Abweichungen, sollten Traceabilityanalysen durchgeführt werden. Mit deren Hilfe kann nachvollzogen werden, ob jedes Element im C-Code auf das bereits gegen die Spezifikation verifizierte Modell zurückzuführen ist und kein unerwünschtes Verhalten zu erwarten ist.

Dieser Referenzworkflow ist so nur mit geeigneter Toolunterstützung wirtschaftlich umsetzbar. Ein kleines Hindernis könnte hier die Coverageanalyse auf Modellebene sein, da in einigen Modellierungsumgebungen eine derartige Analyse noch nicht umgesetzt wurde.

2.3.3 Rechtliche Aspekte[1]

Für Entwicklungen im Automobilbereich sind hauptsächlich die Bestimmungen des Vertragsrechts und der deliktischen Produkthaftung von Bedeutung. Kundenanforderungen und -zufriedenheit sind ohnehin hohe Anreize, technische Normen in der Entwicklung zu beachten. Die Produkthaftung liefert aber ein weiteres Argument, seine Produkte entsprechend der Normen mit Sorgfalt zu entwickeln. Durch vermeidbare Fehler verursachte Unfälle können im Extremfall sogar zu einer strafrechtlichen Verurteilung führen. Die funktionale Sicherheit sollte daher im Interesse der Kunden aber auch der Hersteller höchste Priorität besitzen.

Standards sind unverbindliche Empfehlungen, stellen aber in vielen Rechtsbereichen einen Maßstab für den Stand der Technik dar. Im Rahmen des deutschen Vertragsrechts wird die ISO26262 als zentrale Sicherheitsnorm der Automobilindustrie für E/E-Syteme[2] angesehen und davon ausgegangen, dass sie Bestandteil des Vertrages ist. Wird sie nicht explizit im Vertrag erwähnt, übernimmt der Auftragnehmer die Verantwortung für die ordnungsgemäße Umsetzung. Kommt es zu einem Unfall, muss der Hersteller im Rahmen der Produkthaftung nachweisen, dass sein Produkt dem zum Zeitpunkt des Inverkehrbringens aktuellen Stand der Technik entspricht. Da eine Norm einen Maßstab für den Stand der Technik darstellt, sollte zumindest die Erfüllung der anwendbaren Normen nachgewiesen werden können.

Die ISO26262 ist aus technischer Sicht zukunftsweisend, was einige rechtliche Unsicherheiten für die Hersteller mit sich bringt. Eine Norm sollte alles das dokumentieren, was ein verantwortungsvoller Fachmann ohnehin tut. Leider gibt es für viele der in der ISO26262 definierten Prozesse noch keine „best practice“, welche auf langjährigen Erfahrungen basieren. Da für einige Prozesse

[1] Dieser Abschnitt basiert auf dem Paper [Reu14], welches sich mit dem deutschen Recht befasst. Da die Produkthaftung in Rahmen der EU Richtlinie 85/374/EWG geregelt ist und diese in nationalen Gesetzen umgesetzt wurde, sind die diesbezüglich angestellten Überlegungen so oder so ähnlich in den restlichen EU-Mitgliedsstaaten gültig.

[2] Elektrik- und Elektronik-Systeme

noch gar keine Umsetzung existiert, sind die Hersteller in der Bedrängnis zu begründen, warum diese Prozesse nicht in den Entwicklungsprozess einfließen. Den Herstellern von E/E-Systemen in der Automobilbranche ist daher anzuraten sorgfältig zu dokumentieren, warum einzelne Prozesse zu einem bestimmten Zeitpunkt noch nicht oder nur teilweise umgesetzt werden.

Das in Kraft treten der ISO26262 hätte für die Automobilindustrie bedeutet, dass alle E/E-Systeme den aktuellen Stand der Technik nicht mehr erfüllt hätten und somit nicht mehr ausgeliefert werden hätten dürfen. Da die nachträgliche Erfüllung der Anforderungen der ISO26262 quasi unmöglich ist, wurde eine zusätzliche Regelung für Alt-Systeme getroffen, welche dies verhindert hat. Vor dem Inkrafttreten der ISO26262 entwickelte Systeme werden dabei als ISO-konform angesehen, wenn sie nach den bisherigen Qualitätsmanagement-Anforderungen entwickelt wurde. Nach der unglücklichen Formulierung des entsprechenden Absatzes fallen die Alt-Systeme unter die IEC61508, was so nicht gewollt war. Bei einem Prozess ist der Hersteller nun gezwungen zu begründen, warum die IEC61508 auf sein System nicht anzuwenden war.

Um nach dem in Kraft treten der ISO26262 Änderungen an einem Alt-System vornehmen zu können, wurden Delta-Betrachtungen zugelassen. Bei Delta-Betrachtungen werden nur die Änderungen zum Alt-System betrachtet. Diese erlauben Fehler auszubessern oder neue Funktionen zu integrieren ohne gleich das ganze System mit dem vollen Umfang der ISO26262 betrachten zu müssen. Allerdings müssen die Änderungen mit allen Regeln der ISO26262 entwickelt werden. Zusätzlich muss die Plausibilität der Delta-Betrachtung durch einen unabhängigen Experten bestätigt werden. Dieser Experte darf dabei aus dem eigenen Haus stammen, darf aber nicht in die Entwicklung des Produktes involviert sein. Für den Hersteller ist es sehr wichtig sorgfältig zu dokumentieren, auf welchem Alt-System aufgebaut und welche Änderungen durchgeführt wurden. Damit wird es möglich im Nachhinein schlüssig zu argumentieren, warum bei der Entwicklung eines bestimmten Teiles des Produktes einzelne Prozessschritte nicht notwendig waren.

Die größte Herausforderung für die Hersteller stellt die Qualifizierung der Komponenten dar. Das gängige Qualifikationsverfahren für Bauelemente in der Automobilindustrie (AEC Q) genügt den hohen Ansprüchen der Norm leider nicht. Somit ist in der Praxis der in der Norm geforderte Nachweis auf Tauglichkeit der Komponenten nicht oder nur sehr eingeschränkt möglich, da keine belastbaren Daten über die Zuverlässigkeit der einzelnen Komponenten vorhanden sind.

2.4 Zusammenspiel mit anderen Verifikationsmethoden

Da Testen nur eine von mehreren Verifikationsmethoden darstellt, werden in diesem Unterkapitel neben den Tests weitere Verifikationsmethoden vorgestellt. Am Ende des Kapitels wird auf die Einsatzgebiete eingegangen und begründet, warum sich diese Arbeit mit der Verifikation durch Testen beschäftigt.

Die Verifikationsmethoden können in zwei Kategorien unterteilt werden. Bei statischen Analysen muss das Programm nicht ausgeführt werden und somit muss auch kein ablauffähiges Programm vorhanden sein. Auf der anderen Seite stehen die ablaufbasierten Analysemethoden, für die sehr wohl ein ablauffähiges Programm zur Verfügung stehen muss. Die in diesem Abschnitt beschriebenen Methoden Review, automatische statische Analyse und formale Methoden gehören zur ersten und die Tests zur zweiten Kategorie.

2.4.1 Tests

Unter Testen von Software versteht man das Ausführen der Software unter bestimmten Randbedingungen. Dabei werden stichprobenartig Daten an die Eingänge des zu testenden Objektes angelegt und die Ausgangsgrößen gemessen. Das Testobjekt erfüllt die geforderten Eigenschaften dann, wenn die Ist- mit den Sollausgangsgrößen übereinstimmen.[Vig10, S. 26]

Da die Software ausgeführt werden muss, ist beim Testen ein erhöhter Aufwand gegenüber anderen Verifikationsmethoden zu betreiben. Der Programmcode oder bei MDSD das Modell muss in eine ablauffähige Form gebracht oder simuliert werden. Im Idealfall sollten Tests auf der Zielplattform ausgeführt werden, allerdings ist dies insbesondere bei Embedded Systems nicht immer oder nur mit extrem erhöhtem Aufwand möglich. Um eine Unit auf der Zielplattform ablauffähig zu machen und damit die Unittests zu ermöglichen, muss in den meisten Fällen sehr viel zusätzlicher Code ergänzt werden. Unabhängig davon kann es nötig sein, für einzelne Tests Testtreiber und Teststubs zu schreiben um die Umgebung des zu testenden Objektes zu modellieren. Ebenso gibt es Testmethoden, bei denen es notwendig ist, zusätzlichen Code in das Programm einzufügen[3].

Bei komplexen Systemen wird kaum ein vollständiger Test, bei dem alle möglichen Eingangskombinationen an das System Under Test (SUT) angelegt werden, durchgeführt, da der Testaufwand und die Kosten wirtschaftlich nur in den wenigsten Fällen vertretbar sind. Daher wird bereits in der Definition von Testen auf die stichprobenartige Natur dieses Vorgangs hingewiesen. Gegenüber einem vollständigen besitzt der stichprobenartige Test allerdings den Nachteil, dass die Abwesenheit von Fehlern nicht bewiesen werden kann. Ein stichprobenartiger Test kann somit nur zum Auffinden von Fehlern eingesetzt werden.

2.4.2 Review

Der Review ist ein Vorgang bei dem das Ergebnis eines Arbeitsschrittes durch möglichst unvoreingenommene Personen bewertet wird, um Fehler bei der Erstellung zu finden. Somit ist ein Review als Aktivität der Qualitätssicherung einzuordnen, welcher die Qualität des Projektes sicherstellen soll [Som12]. Dazu wird je nach Lage des Reviews im Design-Prozess zwischen mehreren Review-Typen unterschieden. Darunter befinden sich unter anderem Software-Requirements-, Modell-, Code-, aber auch Testdesign-Reviews.

Ein Review-Prozess kann grundsätzlich in drei Phasen von Aktivitäten unterteilt werden. Aktivitäten vor dem Review, dem Review selbst und den Aktivitäten nach dem Review. Im Folgenden werden die einzelnen Review-Aktivitäten aus der Abbildung 2.5 genauer beschrieben [Som12]:

- Aktivitäten vor dem Review: Bei der Review-Planung werden die Teams zusammengestellt, die zu prüfenden Dokumente ausgegeben und der Ort und die Zeit des Reviews festgelegt. Die Gruppenvorbereitung bietet den Rahmen, in dem sich die Teammitglieder einen Überblick über die zu prüfenden Dokumente und die anzuwendenden Standards machen sollen. In einem weiteren Schritt der individuellen Vorbereitung prüft jeder unabhängig auf Versäumnisse, Fehler und Abweichungen von den Standards.
- Review-Sitzung: In der Review-Sitzung werden die Dokumente im Beisein des Autors durchgegangen und die Kommentare des Review-Teams festgehalten. Während der Sitzung wird durch den Sitzungsleiter ein Review-Protokoll verfasst, in dem die getroffenen Entscheidungen und vorzunehmenden Maßnahmen dokumentiert werden.

[3] Zum Beispiel ist zur Messung der Code Coverage zusätzlicher Code notwendig.

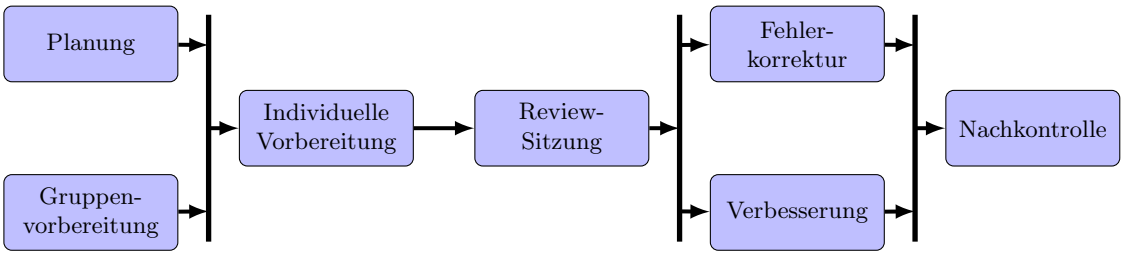

Abbildung 2.5: Prozessschritte bei der Durchführung eines Reviews. Gezeichnet nach [Som12].

- Aktivitäten nach dem Review: Nach der Review-Sitzung werden Lösungen für die aufgedeckten Probleme gesucht und die entsprechenden Überarbeitungen durchgeführt.

Dabei können die einzelnen Phasen je nach Wahl der Review-Strategie und des Review-Typs stark unterschiedlich gewichtet werden. Tendenziell können Reviews, die weiter unten im V-Modell angesiedelt sind, informeller gestaltet werden als solche, die weiter oben angesiedelt sind. Dies spiegelt sich in der Tatsache wieder, dass die Kernmannschaft eines Reviews aus drei bis vier Personen bestehen sollte [Som12], aber bei einem Code-Review meistens nur aus einer Person besteht.

Obwohl im Zuge eines Reviews Fehler aufgedeckt werden, sollte dem Projektverantwortlichen klar sein, dass ein Review kein Werkzeug zur Mitarbeiterbewertung darstellt, sondern einzig zur Verbesserung der Softwarequalität dient [Som12]. Gerade bei Reviews durch einen gleichgestellten Kollegen, bei sogenannten Peer-Reviews, besteht ansonsten die Gefahr, dass ein Reviewer den gefundenen Fehler nicht dokumentiert, damit ihm der Kollege bei einem Fehler seinerseits denselben Gefallen tut. Für diesen nicht dokumentierten Fehler kann dadurch im Weiteren auch keine Strategie entwickelt werden um diesen zukünftig zu vermeiden. Andererseits darf kein Umfeld entstehen, in dem im Falle von Fehlern Schuldzuweisungen erfolgen. Vielmehr sollte dem entsprechenden Ersteller Hilfe bei der Lösung angeboten werden. Darüber hinaus sind regelmäßige Reviews ein wichtiges Hilfsmittel zur Verbreitung von Know-how innerhalb eines Teams, wodurch sich das Projektrisiko im Krankheits- oder Kündigungsfall reduziert [Grü13, Kap. 5].

2.4.3 Automatische statische Analyse

Automatische statische Analysen sind ein Werkzeug um den Review, bei dem es sich ebenfalls um eine statische Analyse handelt, durch Automatisierung zu unterstützen. Bei einem klassischem Review Prozess werden die zu prüfenden Dokumente meist anhand von Checklisten auf Fehler überprüft. Mit zunehmender Länge dieser Liste vergrößert sich der Aufwand für den Review. Um diesen Aufwand in Grenzen zu halten, wird der gut automatisierbare Teil dieser Liste mittels eines Tools oder mehrerer Tools zur statischen Analyse abgearbeitet. Diese Tools können die Checkliste nicht nur schneller abarbeiten als es ein Mensch tun könnte, sondern besitzen den Vorteil, dass die Qualität des Reviews über alle Dokumente gleich bleibt.

Obwohl Tools dieser Art grundsätzlich in allen Ebenen des V-Modells zum Einsatz kommen können, wurden diese bis jetzt weitgehend nur für die Codeebene entwickelt. Daher wird in der Literatur sehr oft die automatische statische Analyse der automatischen Code-Analyse gleichgesetzt. Mit der zunehmenden Verbreitung von MDSD steigt aber auch das Interesse der Hersteller diese Tools für die Modellebenen, wie PIM und PSM, bereitzustellen.

Nach [Som12, S. 450] können in einem statischen Code-Analyse-Tool drei verschiedene Prüfstufen implementiert sein. Im Folgenden werden die einzelnen Stufen aufgegriffen und für die Anwendung in MDSD erweitert.

- Prüfen auf typische Fehler: Eine Untersuchung von großen Codemengen ergab, dass ca. 90% der Programmierfehler auf 10 typische Fehlerarten zurückzuführen sind. Daher bildet das Prüfen auf solche Fehler die Grundlage für viele Code-Analyse-Tools. Den Entwicklern von Analyse-Tools auf Modellebene ist daher anzuraten, Untersuchungen an einer großen Anzahl an Modellen durchzuführen um typische Fehler identifizieren zu können.
- Prüfen auf benutzerdefinierte Fehler: Bei diesem Ansatz werden durch den Benutzer Fehlermuster vorgegeben, nach denen der Code bzw. das Modell durchsucht wird. So können Tools, welche diese Funktion anbieten, z.B. zur Überprüfung von Namens- und Style-Konventionen aber auch für komplexere benutzerdefinierte Analysen, eingesetzt werden.
- Prüfen auf Zusicherungen: Das statische Überprüfen auf Zusicherungen stellt den allgemeinsten und damit den mächtigsten Ansatz dar. Das Analyse-Tool überprüft dabei, ob das Programm bei einem bestimmten Programmpunkt durch den Entwickler zuvor festgelegte Beziehungen erfüllt. Dazu ist es meist notwendig, die Beziehungen als speziell formatierten Kommentar in den Code einzufügen. Das statische Analyse-Tool analysiert den Code mit Hilfe eines komplexen Algorithmus, um alle möglichen Ausführungspfade ausfindig zu machen. Das Tool untersucht anschließend, ob die Beziehung zu dem bestimmten Programmpunkt für jeden dieser Ausführungspfade erfüllt wird. So kann z.B. sichergestellt werden, dass eine Variable an einem bestimmten Programmpunkt in einem definierten Wertebereich liegt.

Automatische statische Analyse stellt daher nicht nur eine Entlastung für im Review beteiligte Personen dar, sondern kann die Tiefe der Analyse in bestimmten Punkten enorm erhöhen. Besonders eine Überprüfung von Zusicherungen wäre ohne Unterstützung von derartigen Analysewerkzeugen aus Kosten- aber auch Zeitgründen kaum möglich.

2.4.4 Formale Methoden

Formale Methoden sind mathematische Methoden, mit denen formale Modelle von Software erstellt und überprüft werden können. Als Ausgangspunkt der Modellierung dienen oftmals natürliche Sprache, Diagramme und Tabellen. Diese informell verfassten Spezifikationen müssen anschließend in die entsprechende mathematische Beschreibungsform gebracht werden. Das so entstandene formale Modell dient als Grundlage für die Spezifikation der Software. Zusätzliche Analysen des Modells können eingesetzt werden um Fehler in der Spezifikation zu vermeiden bzw. Eigenschaften der Spezifikation zu überprüfen. Nach der Erstellung des Programms wird die formale Spezifikation als Referenz der Überprüfung der Software verwendet. Dazu können manuelle oder Tool-unterstützte Methoden zum Einsatz kommen. Ziel dabei ist es, einen Beweis dafür zu erhalten, dass das Verhalten der Software mit der formalen Spezifikation übereinstimmt. Die Entwicklung von Tools geht sogar in die Richtung, die Software aus der formalen Spezifikation zu generieren.[Som12]

Die formale Spezifikation gliedert sich beim Designprozess normalerweise zwischen der Spezifikation der Software Requirements und dem ausführlichen Systementwurf ein. Der in Abbildung 2.6

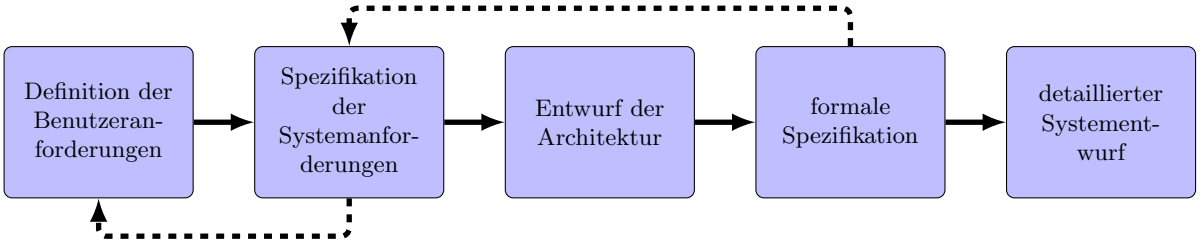

Abbildung 2.6: Designprozess unter Verwendung einer formalen Spezifikation. Gezeichnet nach [Som12]

vorgestellte Designprozess sieht mehrere Schritte zwischen der Definition der Benutzeranforderungen und dem detaillierten Systementwurf vor. Eine Überarbeitung von vorhergehenden Prozessschritten ist meist dann notwendig, wenn Fehler in einem nachgelagerten Prozessschritt aufgedeckt werden. In der Abbildung werden die eventuell notwendigen Rückschritte durch die strichlierten Linien dargestellt.

Wichtige Gründe, warum sich formale Methoden in der Praxis (noch) nicht durchgesetzt haben [Som12]:

- Domain-Experten können eine formale Spezifikation nicht auf die Einhaltung der Anforderungen überprüfen, weil sie die Spezifikation nicht verstehen.
- Entwickler müssen eine neue Sprache lernen, um Spezifikationen erstellen bzw. lesen zu können.
- Die Kosten für die Einführung sind sehr gut, aber die erzielbaren Ersparnisse sind im Vorfeld nur sehr schwer abschätzbar.
- Die aktuellen Ansätze sind nur schlecht skalierbar.
- Formale Spezifikation ist nicht mit der sogenannten agilen Softwareentwicklung kompatibel.

Der Einsatz von formalen Methoden sollte aber besonders für die Entwicklung von sicherheitsrelevanten[4] (Teil-)Systemen in Erwägung gezogen werden. Formale Methoden haben gegenüber Tests den entscheidenden Vorteil, dass sie die Abwesenheit von Fehlern bis zu einem gewissen Grad beweisen können [Grü13, Kap. 8.13]. Während einige Ansätze, wie symbolische Tests, zur formalen Verifikation in der Praxis nur sehr schwierig anwendbar sind, gibt es mit *Model Checking* eine gut automatisierbare Methode. Als Grundlage der Überprüfungen dient ein formales Modell des Systems, welches im besten Fall direkt aus dem Programmcode erzeugt wird. Dieses Modell wird dann mit Hilfe eines Model Checkers auf die Erfüllung formal definierter *Properties* überprüft. Verfügbare Model Checker verwenden häufig die Computation Tree Logic (CTL), mit der es möglich ist Variablenzustände über die Zeit zu spezifizieren [Grü13, Kap. 8.13.3].

2.4.5 Erläuterungen

In der Praxis sollte immer eine Kombination von mehreren Verifikationsmethoden angewendet werden, da jede dieser Methoden bestimmte Stärken aber auch Schwächen mit sich bringt. Die

[4] Sicherheit in Bezug auf den Schaden, den das System der Umwelt zufügen kann. (Safety)

Gewichtung der einzelnen Methoden ist dabei sehr stark vom Projekt abhängig. Dabei ist es wichtig, diese gut und zum richtigen Zeitpunkt in den Designprozess zu integrieren.

Formale Methoden können zwar für einzelne Komponenten eingesetzt werden, sind aber nur sehr schwer auf das gesamte Softwareprojekt anzuwenden. Aus diesem Grund und den vergleichsweise hohen Kosten werden formale Methoden in der Praxis fast ausschließlich zur Verifikation von hoch sicherheitskritischen Systemkomponenten eingesetzt. Diesbezüglich empfiehlt die ISO26262 den Einsatz von formalen Methoden zur Verifikation von Architektur und Units erst ab dem Sicherheitslevel ASIL C. Da sich diese Arbeit mit der Anwendersoftware des Motorsteuergerätes befasst und die meisten sicherheitskritischen Funktionen in der Basissoftware gebündelt sind, wird auf den Einsatz von formalen Methoden verzichtet.

Bei weniger sicherheitskritischen Funktionen kann eine Prüfung auf Zusicherungen als Alternative zu formalen Methoden angesehen werden. Die entsprechende Toolunterstützung vorausgesetzt, können Bedingungen sehr einfach in den Quellcode integriert werden, um anschließend überprüft zu werden. Leider existieren diese Tools vorläufig nur für die Codeebene. Das in der Entwicklung verwendete Programm ASCET[5] enthält allerdings Funktionen, die der Prüfung auf Zusicherungen sehr nahe kommen. So kann unter anderem für jede Variable ein gültiger Wertebereich definiert werden und eine zusätzliche Einstellung ermöglicht es, den Ausgangswert auf diesen Wertebereich zu beschränken. Ein stark vereinfachtes Beispiel dient dazu, die Funktion des Wertebereichs zu erklären. Eine Funktion soll die Multiplikation von zwei Zahlen, jeweils im Wertebereich 0 bis 10, realisieren. Setzt man nun für die Ausgangsvariable eine Zusicherung, welche fordert, dass die Ausgangsvariable ≤ 50 sein muss, würde durch ein statisches Analyse-Tool durch Prüfung auf Zusicherungen eine Fehlermeldung ausgegeben. Wird im Vergleich dazu in ASCET der Wertebereich der Ausgangsvariable von 0 bis 50 festgelegt, erfolgt durch ASCET bei der Codegenerierung eine Analyse auf Einhaltung des Wertebereichs. Je nachdem, ob die Begrenzung aktiviert ist oder nicht, erfolgt eine Fehlermeldung bzw. Information. Die Fehlermeldung tritt immer dann auf, wenn der von ASCET berechnete mögliche Ausgangsbereich nicht im zuvor vom Benutzer festgelegten Ausgangsbereich liegt und der Ausgangswert nicht begrenzt wird. Wurde die Begrenzung aktiviert, erfolgt eine Information an der Benutzer und der Ausgangswert wird durch den generierten Code so begrenzt, dass dieser immer im definierten Ausgangsbereich liegt.

Da es momentan noch keine Tools von Anbietern gibt, wird parallel zu dieser Diplomarbeit ein Programm entwickelt, welches die Überprüfung des ASCET-Modells auf verschiedene Konventionen erlaubt. Dieses Programm ermöglicht es, das Modell unter anderem auf das Einhalten der Namenskonventionen zu prüfen. Vorausgesetzt es bestehen einheitliche Regelungen bezüglich der Begrenzung von Ausgangsgrößen, könnte dieses Tool so erweitert werden, dass die Einstellung diesbezüglich überprüft wird. Wie aus den Erläuterungen zu den einzelnen Verifikationsmethoden herausgeht, sollte eine automatische statische Analyse aber niemals einen Review ersetzen, da diese keine Prüfung gegen die Spezifikation durchführt und somit nicht deren Einhaltung garantiert.

Dass Testen eine spezielle Form der Verifikation darstellt, lässt sich bereits aus dem V-Modell in Abbildung 2.1 erkennen, da es explizit angeführt wird und Tests verschiedener Art auf mehreren Ebenen stattfinden. Gegenüber den anderen Verifikationsmethoden bietet Testen den entscheidenden Vorteil, dass alle darunter liegenden Abstraktionsebenen bei der Ausführung eines Tests mitverifiziert werden. Angenommen es würde ein vollständiger Systemtest durchgeführt, was aus Kosten- und Zeitgründen leider nicht möglich ist, würde dieser nur ohne Fehler durchlaufen, wenn

[5] Tool zum MDSD. Für nähere Informationen siehe 3.2.2.

keine Fehler bei der Erstellung der Systemarchitektur, der Spezifikation der Units, der Implementierung der Units, der Übersetzung des C-Codes in Maschinensprache usw., gemacht wurden. Wie bereits in Abschnitt 2.4.1 hervorgehoben wurde, ist ein Test stichprobenartiger Natur und sollte daher nur in Kombination mit anderen Verifikationsmethoden in der Softwareentwicklung eingesetzt werden.

Da Reviews bereits einen festen Bestandteil des Design-Prozesses darstellen und geeignete Tools zur automatischen statischen Analyse zur Verfügung stehen bzw. in Entwicklung sind, wurde für diese Arbeit das Thema Testen von Units in den Mittelpunkt gestellt.

3 Ausgangspunkt der Arbeit

In diesem Kapitel wird ein Eindruck gegeben, in welchem Umfeld die Arbeit durchgeführt wurde, wie die zu diesem Zeitpunkt in Verwendung befindliche Toollandschaft ausgesehen hat und Tests durchgeführt wurden.

3.1 Aufbau der Motorsteuergeräte-Software

In Abbildung 3.1 ist der Aufbau der Software schemenhaft dargestellt. Im Folgenden wird etwas genauer auf die einzelnen Softwarekomponenten eingegangen.

PVER

Der PVER (Programm Version) stellt das gesamte Softwaresystem dar und enthält zusätzlich zu den in Abbildung 3.1 angeführten Komponenten das Scheduling, Komponenten zum Monitoring der Systemfunktionen und viele weitere Komponenten. Für das Testen von Units sind in erster Linie die Scheduling-Informationen notwendig. Besonders wichtig ist dabei das Intervall zwischen zwei Ausführungen einer Unit. Diese Zykluszeit wird in einem statischen Scheduling festgelegt und bestimmt, wie schnell auf Änderungen reagiert werden kann und im Sinne der Realtimeanforderungen auch reagiert werden muss. Im System wird zwischen mehreren Task-Typen, unter anderem den segmentsynchronen Tasks, 10 ms-Tasks und 100 ms-Tasks unterschieden. Die Zeit zwischen zwei Ausführungen eines segmentsynchronen Tasks ist von der Motordrehzahl abhängig und erreicht sein Minimum bei der maximalen Motordrehzahl.

Basissoftware

Die Kommunikation mit der Umwelt erfolgt über die Basissoftware. Die Basissoftware enthält dazu als unterste Schicht die Hardwareabstraktionsschicht, welche der restlichen Software ermöglicht, von der Hardware unabhängig zu sein. Ebenso ist dadurch eine einfache Anpassung der Software auf geänderte Hardware möglich. Um Schnittstellen mit höherer Abstraktion zur Verfügung stellen zu können, werden in der Basissoftware Teile verschiedenster Protokollstacks implementiert, wobei die Grenze zwischen Hardware- und Softwareimplementierung des Stacks von der jeweiligen Zielhardware abhängig ist. Ein in der Automobilindustrie sehr häufig benötigter Protokollstack ist jener von CAN (Controller Area Network). Neben diesen Funktionen übernimmt die Basissoftware das Fehlermanagement und bringt das Automobil bei Bedarf in einen sicheren Zustand.

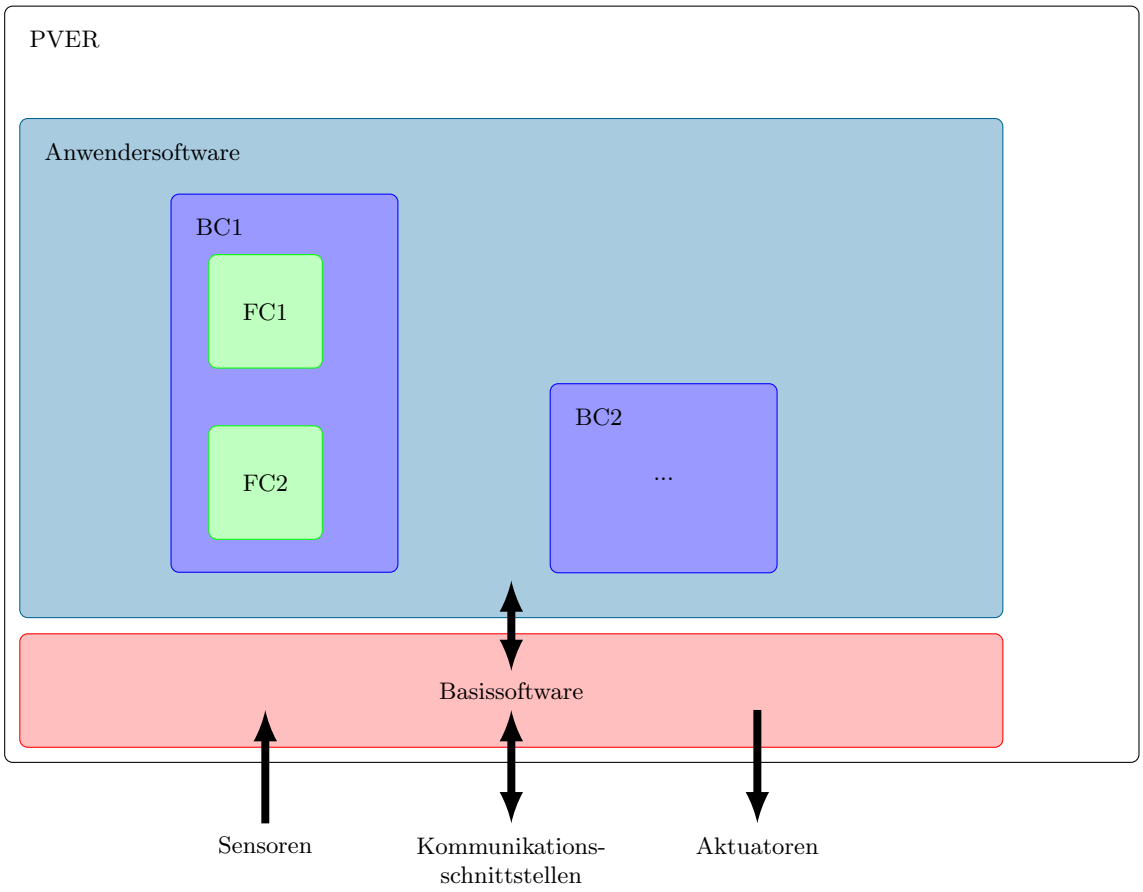

Abbildung 3.1: Diese Abbildung zeigt die grobe Gliederung der Software und die Kommunikationspfade[1] zwischen verschiedenen Softwareteilen. Eine Funktion (FC) entspricht dabei einer Unit aus den V-Modell. Der PVER ist das auf der Hardware lauffähige Programm und ist damit als System anzusehen.

Anwendersoftware

Die Anwendersoftware enthält die eigentlichen Funktionen der Software, wie das Management der Einspritzung. Sie wird speziell nach den Wünschen des Kunden angepasst und das Verhalten kann durch Einstellparameter auch nach der Auslieferung der Software verändert werden. Eine BC (Base Component) dient als Kapselung von FCs (Functional Components). Eine BC beinhaltet meist alle Funktionen zu einem bestimmten Themengebiet, wie zum Beispiel Abgassteuerung.

3.2 Verwendete Tools

Dieser Abschnitt gibt einen kurzen Einblick in die für den Entwicklungs- und den Testprozess verwendeten kommerziellen Tools. Zusätzlich zu diesen Tools werden Entwickler und Tester bei ihrer Arbeit durch verschiedenste Skripts und hauseigene Tools unterstützt.

[1] Kommunikationspfade innerhalb der Anwendersoftware wurden nicht explizit abgebildet.

3.2.1 TPT

TPT (Time Partition Testing) [9] ist ein von PikTec entwickeltes Testwerkzeug für den Test eingebetteter Systeme. TPT unterstützt mehrerer Phasen des Testprozesses. Neben der Möglichkeit Testfälle modellbasiert zu erstellen unterstützt TPT den Tester bei Testverwaltung, Testausführung, Testauswertung und Testdokumentation.

Das Testtool ist so gestaltet, dass Testausführung und Testauswertung von der Testmodellierung unabhängig sind. Testfälle können somit leicht auf andere Plattformen übertragen werden. Durch eine hohe Anzahl von unterstützten Plattformen können unter anderem folgende Tests ausgeführt werden.

- Tests von ASCET-Modellen[2]
- Tests von C-Code
- Tests von AUTOSAR[3] Software
- Tests mit Anbindung an INCA[4]
- Tests mit Hardware in the Loop

Des Weiteren wurde durch die Robert Bosch AG die Entwicklung von weiteren Plattformen veranlasst, um TPT in die bestehenden Testsysteme integrieren zu können bzw. mit speziell entwickelte Testsystemen arbeiten zu können.

Eine für die Automobilindustrie sehr wichtige Eigenschaft von TPT ist die ISO26262 Qualifizierbarkeit. Die tatsächliche Qualifizierung des Tools kann allerdings nur durch den Anwender erfolgen, da dazu die Einbettung in den Safety Lifecycle betrachtet werden muss. Zu diesem Zweck wird jeder Release der Software durch die Robert Bosch AG mit dem vorgesehenen Prozess qualifiziert.[10]

3.2.2 ASCET

ASCET steht für Advanced Simulation and Control Engineering Tool und wurde von der ETAS Group für die modellbasierte Entwicklung von Automobilsoftware entwickelt. Zu den Funktionen gehören unter anderem [3][4]:

- Modellbasierte Entwicklung von Automotive Software mit Echtzeit-Anforderungen
- Automatische Generierung des Steuergeräte-Codes
- Entwicklung von AUTOSAR-konformen Komponenten
- Einfache Integration von Bestandscode (Library-Funktionen und C-Code)
- ISO26262-zertifizierter Codegenerator

[2] Für genauere Informationen zu ASCET siehe Abschnitt 3.2.2.
[3] Automotive Open System Architecture ist eine Entwicklungspartnerschaft von Firmen in der Automobilindustrie.
[4] Für genauere Informationen zu INCA siehe Abschnitt 3.2.3.

- Dokumentationsfunktionen
- Import von Simulink- und UML-Modellen

Durch den Softwareentwickler werden die Komponenten mittels Blockdiagrammen und Zustandsautomaten definiert. Die automatischen Codegeneratoren erzeugen aus dem Modell C-Code, welcher mit einem entsprechenden Compiler weiter in den Maschinencode übersetzt werden kann.

Bei Betrachtung aus der Sichtweise des Model-driven Software Development wird bei ASCET auf der Ebene des PIM entwickelt. Mit Hilfe eines plattformabhängigen Generators wird der C-Code direkt aus dem PIM erzeugt. Dabei werden die einzelnen Blöcke durch entsprechende Codeblöcke im Ausgabefile repräsentiert. Steht dem Entwickler kein geeigneter Funktionsblock zur Verfügung bietet ASCET die Möglichkeit, Funktionen innerhalb des User-Interfaces zu definieren. Dabei muss für jede benötigte Plattform der entsprechende C-Code händisch verfasst werden. Nachdem der C-Code generiert wurde, kann dieser durch einen C-Compiler in den Maschinencode der gewählten Plattform übersetzt werden.

3.2.3 INCA

Bei INCA (Integrated Calibration and Acquisition System) [5] handelt es sich um eine aus mehreren Modulen zusammengesetzte Software welche das Testen auf der Hardware unterstützen soll. Je nach Anforderungen des Anwenders können vom Hersteller ETAS Module zur Basissoftware hinzu gekauft werden um den Funktionsumfang zu vergrößern. Wobei das Messdatenanalysewerkzeug MDA, welches in der Basissoftware enthalten ist, auch Standalone erhältlich ist. In der Basissoftware sind ebenfalls ein Steuergeräteflash-Programmiertool PROF, welches den Upload der Software auf die Zielhardware ermöglicht, und ein Werkzeug zur Verwaltung von applikationsspezifischen Parametern namens Applikationsdatenverwaltung (ADM) enthalten. Im Weiteren werden die erwähnten Softwareteile MDA und PROF kurz beschrieben:

MDA

MDA (Measurement Data Analyzer) ist ein Werkzeug zur Darstellung und Analyse von Messdaten im MDF- oder ASCII-Format. Dazu können die Messdaten in verschiedenen Darstellungen angezeigt werden um eine Auswertung durch den Nutzer zu vereinfachen. Im Zusammenspiel mit INCA bereits in der Basissoftware, ermöglicht das Tool die nahtlose Auswertung der Messdaten nach der Testausführung.

PROF

Das Steuergeräteflash-Programmiertool PROF ist für das Flashen des zuvor generierten HEX-Files der Steuergeräte Software gedacht. Die Integration dieses Tools in die Basissoftware hat den entscheidenden Vorteil, dass bei der Testausführung jeder Schritt, vom Flashen der Software bis zur Auswertung in MDA, mit INCA durchgeführt werden kann.

Die Basissoftware bietet somit eine Bedienoberfläche mit benutzerspezifischen Anzeige- und Bedienelementen zu Messzwecken, zur Verwaltung der integrierten Datenbanken und zur Messwertanzeige. Eine integrierte Datenbank stellt dabei die konsistente Verwaltung von Projekten, Hardwarekonfigurationen und Testumgebungen sicher. Die einfache Integration von INCA in ein

bestehendes Testsystem und die Kommunikation mit dem Steuergerät wird durch die Konformität mit mehreren ASAM-Standards (Association for Standardisation of Automation and Measuring Systems) erzielt. Über die Kommunikationsschnittstelle mit dem Steuergerät können aus INCA heraus steuergeräte-interne Daten gelesen und geschrieben werden, wodurch die Testbarkeit des Systems erhöht wird.

3.2.4 LABCAR

LABCAR-Systemkomponenten sind von ETAS produzierte Hardwarekomponenten und die dazugehörige Software zur Ansteuerung, welche verwendet werden, um offene und skalierbare Hardware-in-the-Loop (HIL) Testsysteme aufzubauen. Das Basissystem besteht dabei aus mindestens drei Komponenten [6]:

- LABCAR System Base: Neben der Aufgabe als Verbindungselement zwischen ergänzenden LABCAR-Komponenten und Komponenten von Drittanbietern enthält diese Komponente eine Spannungsversorgung und zwei CAN-BUS-Controller.
- LABCAR System Software Pack: Softwarekomponente zur Ansteuerung der einzelnen Hardware-Komponenten.
- PT-LABCAR Engine ECU I/O: Steckkarten dieser Art führen die Signalgenerierung bzw. Signaldigitalisierung durch. Drüber hinaus verfügt das Modul über digitale und analoge Ein- und Ausgänge und einen Lastsimulator.

Das Zusammenspiel dieser Komponenten ermöglicht dem Anwender das Anlegen von Mixed-Signal Signalen an die Ports des Testobjektes. Somit stellt LABCAR eine Ergänzung des Funktionsumfanges von INCA dar, bei dem es möglich ist steuergeräteinterne Signale auszulesen bzw. zu schreiben. Mit dem System Add-On LABCAR System M/C ist es außerdem möglich, das LABCAR System aus INCA heraus zu steuern und die Messdatenauswertung durchzuführen.

3.2.5 eHooks

eHooks [7] ist ein weiteres Tool, das sich nahtlos in die Toollandschaft von ETAS eingliedert. eHooks erlaubt den Bypass von Variablen. Damit wird es möglich einzelne Werte von Variablen innerhalb des Systems zu manipulieren. Die Abbildung 3.2 dient zur Veranschaulichung eines Bypass.

Das Besondere an eHooks ist, dass das Einfügen eines Bypass erst nach der Generierung des Objektcodes erfolgt. Das dazu benötigte HEX-File wird dazu direkt manipuliert. Dies bietet den Vorteil, dass beim Hinzufügen eines neuen Bypass das System nicht neu kompiliert werden muss.

Nach dem Flashen des generierten HEX-Files können die generierten Bypass-Variablen über INCA stimuliert werden. Um den Wert der Bypass Quelle an die Funktion anlegen zu können, muss der Bypass aktiviert werden. Somit ist es möglich, einen beliebigen Wert an die Funktion anzulegen, ohne von anderen Berechnungen im Vorfeld abhängig zu sein.

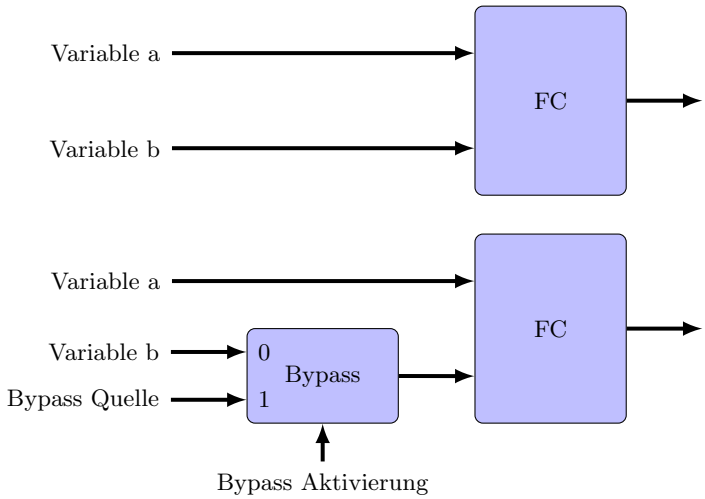

Abbildung 3.2: Schematische Darstellung eines Bypass. Oben: Funktion ohne Bypass. Unten: Funktion mit Bypass der Variable b

3.3 Konzept des Testrahmens

Dieses Unterkapitel geht auf den für die Durchführung der Unittests notwendigen Testrahmen ein.

Der Testrahmen für einen Unittest besteht aus einem vorhergehenden Release des Softwaresystems. Die zu testende Unit wird auf Ebene des C-Codes in das Programm integriert, indem der entsprechende C-Code des Releases durch den C-Code der neuen Implementierung ersetzt wird. Der darauf folgende Kompiliervorgang erzeugt ein auf der Zielhardware lauffähiges Programm, welches die neue Unit enthält.

Das so erstellte Programm erlaubt allerdings noch keine Manipulierung der Eingangsgrößen der Unit und muss daher mit eHooks modifiziert werden. Diese Modifizierung erfolgt in zwei Schritten. Der erste Schritt integriert zusätzliche Prozesse in das System, welche den späteren Zugriff auf interne Größen erlauben. Dieser Schritt ist dabei nicht von der zu testenden Unit, sondern vom verwendeten Release der Software abhängig. In einem zweiten Schritt müssen all jene Signale mit einem Bypass versehen werden, welche beim Testvorgang stimuliert werden müssen. Dieser Schritt ist also von der zu testenden Unit, genauer gesagt sogar von den Testfällen, abhängig.

Da die Signalpfade sozusagen aufgeschnitten werden, um den Zugang frei zu machen, werden die im zweiten Schritt durchgeführten Anpassungen im folgenden *Freischnitt* genannt. Zudem wird dieser Begriff auch vom Toolhersteller ETAS benützt [7].

Die Stimulation der Eingangsgrößen und das Auslesen erfolgt durch das Programm INCA am Hostrechner. Dazu kann die Schnittstelle CAN oder die Serviceschnittstelle ETK, welche einen direkten Zugriff zu Variablen und Parameter im Steuergerät zulässt, genutzt werden.

Die Vorgabe der anzulegenden Werte und die Auswertung erfolgt in INCA manuell.

3.4 Testausführung

Dieses Unterkapitel vermittelt einen Eindruck, wie Tests auf der Zielhardware durchgeführt werden können. Dabei wird nicht auf Details eingegangen, sondern ein grober Überblick gegeben, wie die einzelnen Komponenten interagieren.

3.4.1 Open-Loop Hardware Test

Bei einem Open-Loop Test werden die Steuergeräte-Eingänge mit Signalen stimuliert und die daraus resultierende Ansteuerung der Aktuatoren gemessen. Ein Motormodell, welches die Reaktion des Motors auf diese Ansteuerung simuliert, kommt dabei nicht zum Einsatz.

Zur Veranschaulichung sind in Abbildung 3.3 die Kommunikationspfade zwischen den einzelnen Tools abgebildet. Regelkreise sind nur so weit geschlossen, dass die Diagnose keinen Fehler erkennt und sich das Steuergerät in einem normalen Betriebszustand befindet. In diesem Aufbau kann die Software in einer Konfiguration getestet werden, in welcher der Regelkreis der Drosselklappe geschlossen und mehrere Aktuatoren durch Ersatzlasten simuliert werden müssen. Wird eine andere Konfiguration gewählt, könnte der Testaufbau auch ohne Drosselklappe und/oder Ersatzlasten erfolgen.

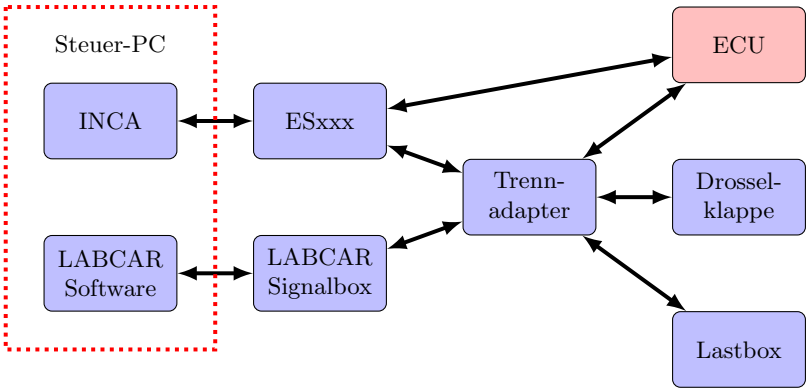

Abbildung 3.3: Schematische Darstellung des Aufbaus eines Open-Loop Testplatzes. Bei ESxxx handelt es sich um ein Hardwaredevice, welches als Adapter zwischen Software und Schnittstellen wie zum Beispiel CAN und ETK dient.

3.4.2 Closed-Loop Hardware Test

Es gibt unzählige Varianten von Closed-Loop Tests. Einige basieren auf Modellen der Umgebung des zu testenden Systems. Im Fall eines Steuergerätes ist zweifellos das Motormodell eines der wichtigsten Modelle. Natürlich ist es auch möglich, den tatsächlichen Motor in Betrieb zu nehmen und bestimmten Umwelteinflüssen auszusetzen. Diese Tests sind aber mit einem viel höheren Aufwand verbunden als eine Simulation des Motors am PC. Die Erstellung von Modellen kann aber mitunter sehr kostspielig sein, weshalb immer genau abgewogen werden sollte, welche Art der Closed-Loop Testausführung für das vorliegende Projekt passend ist.

Vergleicht man die Abbildungen 3.4 des Closed-Loop mit der Abbildung 3.3 des Open-Loop Testsystems, ist erkennbar, dass der Hardwareaufwand um einiges größer ist. Zusätzlich wird ein Motormodell benötigt, dessen Erstellung mit Zeit- und Kostenaufwand verbunden ist.

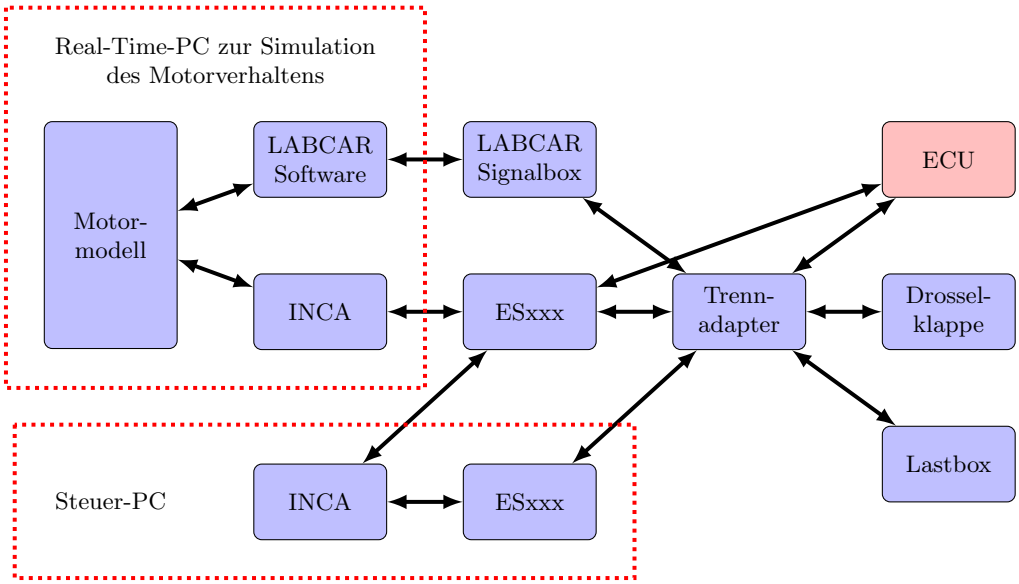

Abbildung 3.4: Schematische Darstellung des Aufbaus eines Closed-Loop Testplatzes. Bei ESxxx handelt es sich um ein Hardwaredevice welches als Adapter zwischen Software und Schnittstellen wie zum Beispiel CAN und ETK dient.

4 Stand der Technik

Dieses Kapitel vertieft die Grundlagen und vermittelt den aktuellen Stand der Technik wie er momentan in der Industrie eingesetzt wird. Der Fokus liegt dabei auf der Automatisierung von Tests, Erstellung von Testfällen und dem Zusammenspiel von Test- und Designprozess.

4.1 Automatisierung von Tests

Die Automatisierung von Abläufen ist ein großer Trend in der Industrie. Sie ermöglicht den effizienten Einsatz der Ressource Mensch und kann viele Arbeitsabläufe extrem beschleunigen.

Unter Testautomatisierung versteht man den Einsatz von Hilfsmitteln zur Durchführung oder Unterstützung der Testaktivitäten [HL14, Testautomatisierung]. Zu diesen Testaktivitäten zählen Testfallerstellung, Testauswertung, Testdokumentation und Testmanagement. In diesem Unterkapitel werden Gründe für die Automatisierung und Anwendungsgebiete aufgezeigt.

4.1.1 Gründe für die Automatisierung von Tests

Je nach Art des Softwareprojekts können verschieden Gründe für die Testautomatisierung sprechen. Einige wichtige Gründe für den Einsatz in der Industrie werden im Folgenden aufgezeigt [11][12][IST14]:

Eindeutige Wiederholbarkeit

Manuelles Testen ist stark von der Erfahrung des Testers abhängig. Auch bei guter Dokumentation der Testausführung kann es vorkommen, dass eine andere Person die Testausführung nicht nachvollziehen kann. Dies kann zu Problemen führen, wenn ein Mitarbeiter, zum Beispiel durch Krankheit, ausfällt. Automatisierte Tests sind im Idealfall so verfasst, dass die Ausführung über einen einfachen Befehl gestartet werden kann und der gesamte Ablauf bis zur abschließenden Dokumentation keine zusätzliche Interaktion mit dem Tester benötigt. Ein solcher Test ist damit unabhängig von bestimmten Personen immer gleich durchführbar.

Objektive Bewertung

Dieser Punkt ist ähnlich gelagert wie die Wiederholbarkeit, allerdings bezieht er sich auf die Auswertung der Testergebnisse. Durch eine automatische Auswertung wird die Qualität der Software messbar und das Vertrauen der Entwickler gegenüber den Testern steigt. Es sollte aber nicht der Eindruck entstehen, dass manuelle Tests nicht objektiv bewertet werden können. Bei manuellen Tests ist der Tester aber dazu verleitet, die Kriterien nicht so genau zu verfassen, wie es für automatisierte Tests notwendig wäre. Damit entsteht unter Umständen ein Interpretationsspielraum bei der Bewertung, ob der Test bestanden wurde oder nicht.

Nachvollziehbarkeit von Fehlern

Bei der Ausführung von manuellen Tests bleibt immer eine kleine Restunsicherheit, ob der Test richtig ausgeführt wurde. Bei einigen Fehlertypen, zum Beispiel sporadisch auftretenden Fehlern, kann dies zum Problem werden. Wenn der Test schon mehrere Male ohne Fehler durchgeführt wurde, ist es sehr wahrscheinlich, dass der Tester an der korrekten Durchführung seinerseits zweifelt und ihn im schlechtesten Fall so lange durchführt, bis er bestanden wird. Eine Aufzeichnung des gesamten Testablaufes könnte diese Unsicherheit ausräumen, ist aber bei Tests mit einem hohen manuellen Anteil meist nur sehr schwierig umsetzbar.

Zeitlich besser planbar

Die automatische Ausführung einer Testsuite dauert bei den einzelnen Testläufen immer etwa gleich lange. Die Dauer bei manueller Ausführung kann jedoch erheblich schwanken. Automatisierte Tests sind also zeitlich besser planbar als manuelle Tests. Gleichzeitig ist der Entwicklungsabteilung besser abschätzbar, wie lange vor dem Release-Termin der Software mit den Tests begonnen werden muss um anschließend die Fehler noch ausbessern zu können.

Ausnutzung der verfügbaren IT und Hardware erhöhen

Das Testen beschränkt sich zeitlich nicht mehr wie bei manuellen Tests auf die Arbeitszeit der Mitarbeiter. Werden die Tests nachts oder über das Wochenende durchgeführt, erhöht sich die Auslastung der Hardware und der IT deutlich. Die Gesamtkosten für die Testaufbauten reduzieren sich, da weniger Testplätze benötigt werden.

Verkürzung der Testphase

In der Testautomatisierung gibt es vielzählige Ansätze die benötigte Zeit zu verkürzen. So kann bereits die automatische Erstellung der Dokumentation eine erhebliche Zeitersparnis mit sich bringen. Bezüglich der Testdurchführung können automatisierte Tests in der Regel schneller ablaufen als manuelle.

Von Menschen nicht durchführbare Tests

Es gibt verschiedenste Gründe, warum die Durchführung eines Tests nicht durch einen Menschen erfolgen kann. Hier seien nur zwei Gründe kurz erwähnt. Die Durchführung kann für den Menschen zu gefährlich sein, wenn zum Beispiel Sicherheitssysteme gezielt abgeschaltet werden um einen Test zu ermöglichen. Die Beschränktheit der menschlichen Fähigkeiten kann es ebenfalls unmöglich machen einen manuellen Test durchzuführen. Bei dem Test eines Protokolls kann es unter anderem nötig sein, sehr schnell auf Nachrichten des System Under Test (SUT) zu reagieren. Die menschliche Reaktionszeit wird im Allgemeinen zu lange sein um rechtzeitig zu reagieren.

Kostenreduktion

Die zu erwartende Kostenreduktion ist vermutlich der Haupttreiber für den verstärkten Einsatz von Testautomatisierung in der Industrie. Die Kosten für die Einführung und die ständige Weiterentwicklung der Testautomatisierung sollten dabei aber nicht außer Acht gelassen werden.

Rechtliche Absicherung

Besonders die Entwicklung von sicherheitskritischen Systemen sollte sich am Stand der Technik orientieren. Da Testautomatisierung als Stand der Technik angesehen werden muss, kann es bei einem etwaigen Gerichtsprozess schwierig werden zu begründen, warum Tests manuell ausgeführt wurden, wenn eine Automatisierung möglich wäre.

Wie bereits erwähnt, können diese Gründe für den Einsatz sehr unterschiedlich gewichtet sein. Bei sicherheitsrelevanten Systemen spielt die rechtliche Absicherung eine höhere Rolle als bei Standard-PC-Software. Bei dieser wiederum können, durch die immer weiter verbreitete agile Softwareentwicklung bei solchen Projekten, die Planbarkeit und eine möglichst kurze Ausführungszeit der Tests wichtige Rollen spielen.

4.1.2 Anwendungsgebiete von automatisierten Tests

Die Automatisierung von Tests kann grundsätzlich fast überall vorgenommen werden. Es haben sich aber besondere Einsatzgebiete herauskristallisiert, in denen die automatischen gegenüber den manuellen Tests signifikante Vorteile besitzen [1][11].

Regressionstest

Das erneute Ausführen eines Tests zur Prüfung eines bereits überprüften SUT wird Regressionstest genannt. Ein Regressionstest wird nach einer Änderung am SUT durchgeführt um sicherzustellen, dass durch eine Änderungen oder das Hinzufügen eines neuen Programmteils kein unerwünschtes Verhalten anderer Programmteile ausgelöst wird.

Performanztest

Performanz ist der „Grad, in dem ein System oder eine Komponente seine vorgesehenen Funktionen innerhalb vorgegebener Bedingungen (z.B. konstanter Last) hinsichtlich Verarbeitungszeit und Durchsatzleistung erbringt“ [HL14, Performanz]. Bei einem Performanztest wird überprüft, ob das zu prüfende System unter den definierten Bedingungen die geforderte Performanz erbringt.

Lasttest

Ein Lasttest ist eine Art Performanztest, bei dem das Systemverhalten in Abhängigkeit von der Systemlast beobachtet wird. Das Ziel des Lasttests ist es zu bestimmen, bis zu welcher Last das System bzw. die Komponente die geforderte Performanz liefert.[HL14, Lasttest][Grü13, Kap. 8.3]

Zusammenfassend lässt sich sagen, dass in der Praxis die Automatisierung von Tests immer dann angestrebt wird, wenn diese oft ausgeführt werden müssen. Dabei spielt es eine untergeordnete Rolle, ob diese Ausführungen hintereinander, wie bei Regressionstests, oder gleichzeitig, wie bei Performanz- und Lasttests, stattfinden.

4.1.3 Erläuterungen

Für die Ausführung von Last- und Performanztests wird üblicherweise ein lauffähiges Programm mit den dazugehörigen Scheduling-Informationen benötigt. Dieses liegt bei Units-Tests nicht vor. Bei der Automatisierung von Unittests steht somit der Gedanke des Regressionstests im Mittelpunkt.

Im konkreten Fall dieser Arbeit ist bei der Entwicklung der FC (Unit) allerdings bereits bekannt, in welchem Intervall die FC durch den Scheduler aufgerufen wird. Auf Unit-Ebene kann daher überprüft werden, ob die Laufzeit der Funktion kürzer als die Zeit eines Intervalls ist.

Studien zeigen, dass sich die Automatisierung eines Tests erst ab ca. 5 Ausführungen finanziell rechnet. Bei sicherheitskritischen Systemen sollten jedoch andere Gründe für die Automatisierung im Vordergrund stehen.

4.2 Erstellung von Testfällen

Dieser Abschnitt stellt zunächst eine Klassifizierung von Tests vor, um anschließend mehrere Methoden zur Erstellung von Testfällen aufzuzeigen.

In der Literatur werden sogenannte abstrakte Testfälle (High Level Testcases) und konkrete Testfälle (Low Level Testcases) unterschieden. Wie die englischen Bezeichnungen vermuten lassen, handelt es sich dabei um unterschiedliche Abstraktionsebenen zur Beschreibung von Testfällen [HL14].

- Abstrakte Testfälle beinhalten noch keine expliziten Eingabewerte und die dazugehörigen Ausgabewerte. Das Verhalten wird über logische Operatoren beschrieben.
- Beim konkreten Testfall werden die logischen Operatoren der abstrakten Testfälle durch konkrete Werte ersetzt.

Einige der hier vorgestellten Methoden zur Testfallerstellung eigenen sich besser zur Erstellung von abstrakten Testfällen, andere eher zur Konkretisierung der selbigen.

4.2.1 Black-, White- und Gray-Box Test

Die Tests können über die Informationen, die zur Testfallerstellung benötigt werden, klassifiziert werden. Die genaue Methode, wie Testfälle aus diesen Informationen abgeleitet werden, ist dabei nicht relevant.

White-Box Test

Die Auswahl der Testdaten erfolgt bei einem White-Box Test mit der Kenntnis über die innere Struktur des zu testenden Objekts. Bezüglich Model-driven Software Development (MDSD) [Kap. 2.2] lässt sich dieser Gedanke auf Modelle zur Softwareentwicklung ausweiten.

Black-Box Test

White-Box Tests stellen bezüglich Testen gegen die Spezifikation ein Problem dar. Die Kenntnisse über die innere Struktur können zu einer Voreingenommenheit des Erstellers der Testdaten führen. Bei Black-Box Tests werden die Testdaten daher nur aus der Spezifikation ohne Kenntnis der inneren Struktur der zu prüfenden Einheit abgeleitet [Vig10]. Aus der Definition lässt sich ableiten, dass ein sinnvoller Black-Box Test nicht vom Entwickler der Einheit erstellt werden kann. Eine Einbeziehung einer zweiten Person als Testentwickler bringt darüber hinaus den Vorteil einer zweiten unabhängigen Interpretation der Spezifikation mit sich. Wurde diese vom Entwickler falsch interpretiert, können Fehler diesbezüglich somit im Test aufgefunden werden.

Gray-Box Test

Der Gray-Box Test stellt eine Kombination von beiden Ansätzen dar. Zum einen werden Testfälle direkt aus der Spezifikation abgeleitet und sind somit von der konkreten Implementierung unabhängig, zum andern werden aber auch Testfälle mit der Kenntnis über die innere Struktur erstellt. Ein typisches Beispiel für einen Gray-Box Test ist der Integrationstest, bei dem die Schnittstellenspezifikationen von und die Kommunikationspfade zwischen Units innerhalb der Systemarchitektur des Systems bekannt sind, aber nicht dessen konkrete Realisierung.

Die Klassifizierung von Tests in Unit-, Integrations-, System- und Akzeptanz-Tests ist davon völlig unabhängig. Theoretisch sind alle Kombinationen möglich, allerdings machen einige mehr Sinn als andere. Als Beispiel seien hier die Akzeptanz-Tests angeführt. Da sich der Kunde im Allgemeinen nicht für die interne Struktur der Software interessiert, wird es sich dabei meist um Black-Box Tests handeln. Der Programmierer einer Unit, welcher kleine Fehler in seinem Code auffinden will, wird hingegen White-Box Tests durchführen, um gezielt den Bereich des Codes zu testen, in dem er einen Fehler vermutet. Tendenziell lässt sich sagen, je weiter unten im V-Modell (Abbildung 2.1) sich der Test befindet, desto mehr Informationen stehen bei der Testfallerstellung zur Verfügung. Akzeptanz- und Systemtests als Black-Box Tests zu konzipieren hat darüber hinaus den Vorteil, dass diese nicht von der konkreten Implementierung der Software abhängig sind und dadurch sofort nach der Spezifikation der SW-Anforderungen mit der Testfallerstellung begonnen werden kann.

4.2.2 Model-Based Testing

Unter Model-Based Testing versteht man das Testen auf Grundlage eines Modells des zu testenden Systems [HL14]. Dabei ist vorgesehen, dass Testfälle aus einem Modell der Software abgeleitet werden [Vig10, Kap. 4.2]. Es existieren aber auch Ansätze für Model-Based Testing, welche auf Modellen von Interaktionen basieren[1].

Bedingt durch die immer besser werdende Toolunterstützung findet Model-Based Testing in der Industrie immer öfter Anwendung. Die Studie [AWS14] gibt dazu an, dass 35,96 % der teilnehmenden Unternehmen bei der Serien-Entwicklung Model-Based Testing anwenden. Allerdings besagt dieselbe Studie, dass nur 8,16 % der Serien Entwickler Testfallgenerierung (Test Case Generation) einsetzen. Model-Based Testing wird also in erster Linie zur Modellierung der Testumgebung und Automatisierung der Testausführung verwendet und weniger zur automatischen Generierung von Testfällen.

Der restliche Abschnitt befasst sich ausschließlich mit der automatischen Generierung von Testfällen aus einem Modell. Dafür gibt es zwei Ansätze.

Generierung aus dem Modell der Codegenerierung

Besonders bei MDSD könnte man verleitet werden, die Testfälle aus dem für die Codegenerierung verwendeten Modell abzuleiten. Da für die Generierung der Testfälle und der Implementierung dasselbe Modell verwendet wird, können damit lediglich Aussagen darüber treffen werden, ob das Verhalten der Implementierung jenem des Modells entspricht. Es erfolgt also keine Verifizierung gegen die Spezifikation, sondern lediglich eine indirekte Verifizierung von Codegenerator und Compiler.

Generierung aus einem unabhängigen Modell

Soll die Implementierung allerdings gegen die Spezifikation der zu testenden Einheit geprüft werden, muss ein zweites Modell erstellt werden. Die aus diesem unabhängigen Modell generierten Testfälle werden dann gegen die aus dem anderen Modell generierte Implementierung ausgeführt und die Implementierung somit gegen die Spezifikation verifiziert.

Abbildung 4.1 veranschaulicht das unterschiedliche Vorgehen bei der Testfallerstellung und vergleicht dieses mit händisch generierten Black-Box Tests. Aus dieser Abbildung lässt sich sehr schön erkennen, dass für die ersten beiden Methoden (a und b) ein höherer manueller Aufwand für die Erstellung der Testfälle notwendig ist, dieser allerdings mit der Verifikation gegen die Spezifikation belohnt wird. Bei der Generierung der Testfälle aus dem Implementierungsmodell (c) verringert sich dieser Aufwand, allerdings muss das Modell mit anderen Verifikationsmethoden gegen die Spezifikation geprüft werden, was mitunter zu zusätzlichen Aufwand außerhalb der hier betrachteten Testfallerstellung führen kann.

[1] Siehe dazu unter anderem [BHK09].

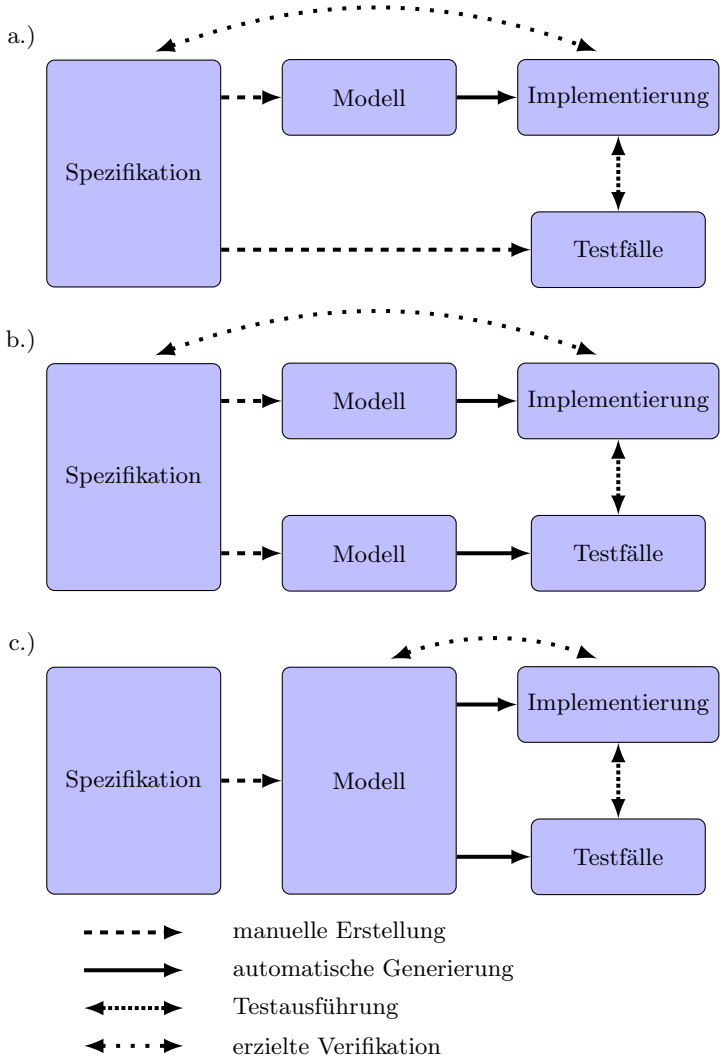

Abbildung 4.1: a.) Manuelle Erstellung der Testfälle ohne Kenntnis der Implementierung; b.) Generierung der Testfälle aus einem unabhängigen Modell; c.) Generierung aus dem Implementierungsmodell; Gezeichnet nach [Som12].

4.2.3 Use Case Testing

Use Case Testing wird im deutschsprachigen Raum auch szenariobasiertes oder anwendungsfallbasiertes Testen genannt. Dabei handelt es sich um ein Black-Box Testverfahren, bei dem die Testfälle auf der Grundlage von Szenarien von Anwendungsfällen erstellt werden [HL14]. Ein Szenario ist dabei eine hypothetische Folge von Ereignissen [2, Szenario], welches so gut wie möglich einem Anwendungsfall des Systems entsprechen soll und ist eine Möglichkeit, wie ein System benutzt werden kann. Dabei ist es wichtig darauf zu achten, dass das Szenario realistisch und nachvollziehbar ist [Som12, Kap. 8.3.2]. Für die Erstellung der Szenarien und die Anwendung dieser Methode zur Testfallerstellung ist es unerheblich, ob die zugrunde liegende Interaktion mit einem menschlichen Benutzer oder einem anderen System erfolgt [HL14, Anwendungsfall]. Allerdings kann die Motivation der Entwickler und die Nachvollziehbarkeit davon abhängig sein.

Use Case Testing kann eng mit dem Requirements Engineering verwoben werden, falls Szenarios erstellt werden, um die Requirements daraus abzuleiten bzw. um direkt als Spezifikation herangezogen zu werden [Som12, Kap. 8.3.2]. Mit Hilfe gut designter Szenarien können z.B. einander widersprechende Requirements, schon in der Frühphase des Projekts aufgedeckt werden. Vermutlich werden aus diesem Grund Szenario-Tester in [Kan03] als Frühwarnsystem für Fehler im Requirements Engineering bezeichnet.

4.2.4 Classifikation Tree

Das systematische Vorgehen bei der Classifikation Tree Methode (CTM) ermöglicht eine effiziente Erstellung von Testfällen. Das Ergebnis dieser Methode ist dabei in den wenigsten Fällen ein einziger Testfall, viel mehr erhält man eine ganze Testsuite bestehend aus mehreren Testfällen. Der dazugehörige Ablauf kann in mehrere Schritte unterteilt werden, die bei Bedarf rekursiv ausgeführt werden [Vig10, Kap. 10.4.3].

- Identifizierung von Aspekten: Aspekte sind dabei als Einflüsse auf die Datenverarbeitung zu verstehen.
- Auffinden von Äquivalenzklassen: Partitionieren des Eingabebereiches. Das System verhält sich für jede Eingabe aus einer Äquivalenzklasse gleich.
- Kombination: Auswahl je einer Klasse für jeden Aspekt. Die Testsuite sollte dabei alle im realen Betrieb möglichen und unmöglichen Kombinationen berücksichtigen.
- Konkretisierung: Auswahl eines konkreten Repräsentators pro Klasse.

Ein einfaches Beispiel soll die Methode veranschaulichen. Dazu wird ein Modul mit der folgenden Spezifikation definiert und die Methode darauf angewendet.

"Das Modul soll die Möglichkeit bieten das Vorzeichen des Eingangswertes IN des Datentyps int8, zu invertieren. Über einen zusätzlichen Eingang INVERT, bool, soll bestimmt werden können, ob das Vorzeichen invertiert ($INVERT = true$) oder der Wert unverändert ($INVERT = false$) an den Ausgang weitergeleitet wird."

- Aspekte: $INVERT$, IN, OUT
- Äquivalenzklassen:
  $INVERT$: false, true; IN: negativ (-128 bis -1), positiv (0 bis 127)
- Kombinationen:
  Testfall 1: $INVERT = false$, $IN = negativ$
  Testfall 2: $INVERT = false$, $IN = positiv$
  Testfall 3: $INVERT = true$, $IN = negativ$
  Testfall 4: $INVERT = true$, $IN = positiv$
- Konkretisierung: Ersetze $negativ$ durch -50, $positiv$ durch 50.

Die einzelnen Testfälle müssen nun um die dazugehörigen Ausgangswerte ergänzt werden um einen ausführbaren Testfall zu erhalten. Dabei ist es auch denkbar, dass die möglichen Ausgangswerte ebenfalls in Äquivalenzklassen unterteilt werden und im Testfall lediglich überprüft wird, ob sich der Ausgangswert in einer festgelegten Klasse befindet.

4.2.5 Boundary Value Testing

Die Erfahrung hat gezeigt, dass die Testentwickler dazu neigen die Software mit typischen Eingabewerten zu testen. Beim Boundary Value Testing werden gezielt Extremwerte der entsprechenden Eingangsvariable als Eingabe genutzt um zusätzliche Fehler aufzudecken. Betrachtet man unter diesem Aspekt das Modul aus Abschnitt 4.2.4, erhält man folgende Extremwerte für die Eingangsvariable IN.

- $IN_{min} = -128$
- $IN_{max} = 127$

Setzt man nun in die oben definierten Testfälle jeweils IN_{min} anstelle von *negativ* und IN_{max} statt *positiv* ein und ergänzt den erwarteten Ausgangswert, erhält man folgende Testfälle:

- Testfall 1: $INVERT = false$, $IN = -128$, $OUT = -128$
- Testfall 2: $INVERT = false$, $IN = 127$, $OUT = 127$
- Testfall 3: $INVERT = true$, $IN = -128$, $OUT = 127$
- Testfall 4: $INVERT = true$, $IN = 127$, $OUT = -127$

Bereits bei der Durchsicht der Testfälle fällt auf, dass der Testfall 3 heraussticht. Dieser Testfall zeichnet sich dadurch aus, dass sich der Betrag des Ausgangswertes vom Betrag des Eingangswertes unterscheidet. Dies begründet sich aus dem Wertebereich des Typs INT8. Wurde das Modul ohne die Berücksichtigung des INT8 Wertebereiches implementiert kann es vorkommen, dass es bei diesen Eingabewerten zu einem Fehler kommt. Eine Beispiel-Implementierung kann dem Listing 4.1 entnommen werden.

```
#include <stdbool.h>

int8 function( bool invert, int8 in )
{
        if (invert == false) {
                return in;
        } else {
                return -in;
        }
}
```

Listing 4.1: Beispiel-Implementierung des Moduls

Bei den Eingabewerten $INVERT = true$ und $IN = -128$ kann es je nach Übersetzung des C-Codes aus Listing 4.1 in Maschinensprache zu einem Überlauf kommen oder nicht. Ein Überlauf der Variable hätte ein Abweichen des Ausgangswertes vom Sollwert zur Folge. Es sei darauf hingewiesen, dass viele Compiler solche Fehler aufdecken und eine entsprechende Meldung ausgeben bzw. entsprechende Maßnahmen in den Maschinencode integrieren. Andererseits fordert die ISO26262 ein Vorgehen bei der Verifikation und Validierung der Software unabhängig vom Entwicklungsprozess und Einsatz der Tool-Kette [Kur13, Kap. 1.3]. Alternativ zu der Verifikation durch Testen könnte dieser Fehler im Code durch ein entsprechend ausgereiftes Tool zur automatischen statischen Codeanalyse sehr einfach bzw. durch einen manuellen Code-Review etwas schwieriger aufgedeckt werden.

4.2.6 Erläuterungen

Um einen Testfall effektiv für Regressionstests einsetzen zu können, sollte dieser so weit wie möglich von der konkreten Implementierung unabhängig sein. Die Betrachtung von außen, also ohne den inneren Aufbau, ermöglicht es, Änderungen in der Unit vorzunehmen ohne den entsprechenden Test anpassen zu müssen. Voraussetzung dafür ist, dass sich die zu Grunde liegende Anforderung in der Spezifikation nicht verändert hat. Zusätzliche Anforderungen an die Unit können somit beliebig in der Unit umgesetzt werden ohne die Ausführbarkeit von Tests zur Überprüfung von anderen Anforderungen zu beeinträchtigen. Beim Testen von Units kann es aber durchaus notwendig sein, White-Box und/oder Grey-Box Tests durchzuführen. Ob diese in die Testsuite für die Regressionstests aufgenommen werden sollen, muss allerdings individuell entschieden werden.

Die ISO26262 [ISO11] empfiehlt abhängig vom ASIL Level eine bestimmte Testabdeckung. Üblicherweise erfolgt die Messung der Testabdeckung auf Ebene des C-Codes mit der sogenannten Codeüberdeckung (Code Coverage). In der modellbasierten Softwareentwicklung wird aus dem Modell meist sehr stark optimierter Code generiert, wodurch die Tester mit den vorgestellten Methoden zur Testfallerstellung Schwierigkeiten haben die geforderte Testabdeckung zu erreichen. Die Arbeit [KM14] zu effektivem Unit-Testen in modellbasierter Softwareentwicklung befasst sich mit dieser Problematik und stellt diesbezüglich eine neue Methode zur Testfallerstellung vor. Die vorgestellte Methode MCDC mit Grenzwertanalyse analysiert den Code und leitet daraus die Testfälle ab.

4.3 Integration von Testen in den Designprozess

Testen ist eng mit dem Designprozess verwoben. Es gibt unzählige Möglichkeiten diese beiden Aspekte der Softwareentwicklung miteinander zu verknüpfen. In diesem Kapitel wird auf verschiedene Möglichkeiten eingegangen Testen in den Designprozess einzubinden.

4.3.1 Demonstratives/Destruktives Testen

Testen ist ein Prozess, bei dem man versucht Schwachpunkte bzw. Fehler in der Software zu finden. Eine Möglichkeit zu Klassifizierung von Tests ist die Aufteilung in demonstrative und destruktive Tests [Vig10, Kap. 3.4.1].

Demonstratives Testen

Bei demonstrativen Tests wird das System systematisch mit dem Ziel ausgeführt die Qualität der Software zu erhöhen. Bei dieser Art des Testens steht eine konstruktive Haltung im Vordergrund. Der Entwickler sucht unterbewusst die Bestätigung, dass seine Implementierung den Anforderungen entspricht. Leider werden mit dieser Herangehensweise nur sehr wenige Fehler gefunden.

Destruktives Testen

Destruktives Testen ist ein Prozess, bei dem die Software während der Ausführung an seine Grenzen gebracht wird, um Fehler aufzudecken. Hier steht also nicht die Bestätigung der Implementierung im Vordergrund, sondern eine Art von destruktivem Denken. Dem Programmierer selbst kann es allerdings sehr schwer fallen diese destruktive Haltung einzunehmen, alleine schon aus der Tatsache heraus, dass er bei der Implementierung eine konstruktive Haltung annehmen muss. Sein eigenes Werk zu zerstören fällt umso schwerer.

Der Autor ist der Meinung, dass im Testprozess das destruktive Testen im Vordergrund stehen sollte, da dadurch mehr Fehler aufgedeckt werden können.

4.3.2 Test Driven Development

Test Driven Development (TDD) wird verstärkt in der Agilen Softwareentwicklung angewandt, kann aber auch sinnvoll in andere Entwicklungsprozesse eingebunden werden. Bei TDD wird der Test gewissermaßen vor der Implementierung verfasst. Der Vorteil liegt darin, dass sich der Entwickler, welcher auch die Tests schreibt, sich vor der Implementierung Gedanken über die Testbarkeit der Unit machen muss. Der Ansatz verfolgt zudem die Strategie Tests und Implementierung immer Zug um Zug zu erstellen. Schlägt also ein Test fehl, kann der Fehler nur im neu hinzugekommenen Code liegen. Der Suchbereich wird beim Debuggen dementsprechend eingeschränkt und führt im Allgemeinen zum schnelleren Auffinden der Fehlerquelle.

TDD macht bei hardwarenaher Software den Einsatz von Simulatoren notwendig. Zum einen kann es vorkommen, dass bei der Programmierung der Software noch keine Hardware zur Verfügung steht, zum anderen ist es sehr aufwendig die Software in jeder Iteration auf die Zielhardware zu bringen. Programme zum Model-driven Software Development (MDSD) bieten diesbezüglich meist eine Simulationsumgebung um Tests durchführen zu können.

Die Testausführung bei TDD sollte einen hohen Automatisierungsgrad aufweisen. Die hohe Anzahl an Iterationen von Implementierung und Testen führt zu einer entsprechend hohen Zahl an Ausführungen eines Testfalles.

4.3.3 Dokumentation von Tests

Es ist notwendig, die Durchführung der Tests und den Testprozess zu dokumentieren. Wie bereits in Abschnitt 2.3.3 angesprochen, ist die Dokumentation ein wesentlicher Bestandteil der Beweisführung vor Gericht. Grünfelder geht in seinem Buch [Grü13] sogar soweit, dass er einen nicht dokumentierten Test einem nicht durchgeführten Test gleichsetzt.

Da der Testprozess viel mehr als die eigentliche Testausführung umfasst, müssen die Entscheidungen bezüglich des Testmanagements ähnlich jenen des Designprozesses dokumentiert werden. Die Testplanung und deren Festhaltung unterstützt zudem während der Testphase ein einheitliches Verständnis und Vorgehen einzelner Teammitglieder. Nur so ist davon auszugehen, dass die Testspezifikation, -durchführung und -dokumentation den Vorgaben und damit, bei entsprechender Testplanung, dem Stand der Technik entsprechen.

In der Industrie wird die Automatisierung der Dokumentation vorangetrieben. Zum einen kann damit die Dokumentation sehr viel schneller erstellt werden und zum anderen wird die Einheitlichkeit gefördert. Das Testteam kann sich somit auf die wesentlichen Arbeiten, wie zum Beispiel die Testfallspezifikation, konzentrieren.

5 Analyse der für Unit-Testen relevanten Subprozesse

Dieses Kapitel befasst sich mit der Analyse der für das Testen der Units relevanten Subprozesse Functional Component Implementation Test (FCI) und Functional Component Functional Test (FCF).

Zunächst wird ein Einblick in die Subprozesse, im Speziellen auf den Workflow, gegeben und anschließend das Verbesserungspotenzial aus Sicht des Autors aufgezeigt.

5.1 Beschreibung der Subprozesse

Dieses Unterkapitel nennt zunächst die Ziele der Subprozesse FCI und FCF und geht dann etwas näher auf den Workflow innerhalb dieser Subprozesse ein.

5.1.1 Beschreibung der Testziele

FCI und FCF unterscheiden sich bezüglich des Testzieles. Das Testziel legt fest, welche Art von Fehler im entsprechenden Subprozess aufgedeckt werden sollen. Im Folgenden werden die Testziele der einzelnen Subprozesse kurz erläutert.

Functional Component Implementation Test

Ziel ist es Fehler in der Implementierung des Codes zu finden. Also die inneren Strukturen und Algorithmen auf Implementierungsfehler zu überprüfen. Bezüglich der Testfallerstellung kann der FCI als White-Box Test angesehen werden. Der Fokus der Tests liegt auf den Änderungen auf C-Code Ebene gegenüber der vorherigen Version der Unit (FC).

Functional Component Functional Test

Ziel des Functional Component Functional Test (FCF) ist die Verifizierung der Funktionalität gegen die Spezifikation. Wechselwirkungen mit anderen Anforderungen sind zu beachten, allerdings liegt das Hauptaugenmerk auf den funktionalen Änderungen. Da die Testfallerstellung nur mit der Kenntnis der Spezifikation erfolgt, handelt es sich um einen Black-Box Test.

5.1.2 Workflow der Subprozesse

Da sich die Workflows der einzelnen Subprozesse stark ähneln und die Testanweisungen wo möglich eine parallele Ausführung von FCI und FCF vorschlägt, wird nur ein Workflow beschrieben.

Die einzelnen Aktivitäten des in Abbildung 5.1 dargestellten Workflows werden im Folgenden kurz beschrieben und wo nötig auf unterschiedliche Definitionen der einzelnen Aktivitäten hingewiesen.

Testumgebung festlegen

Diese Aktivität dient zur Festlegung der Testumgebung. Als solche stehen die PC-basierte Testumgebung, die beiden bereits in Unterkapitel 3.4 beschriebenen Testumgebungen für Tests auf der Zielhardware und das Auto als Testumgebung zur Verfügung. Der Tester ist dazu angehalten, die Testumgebung entsprechend des Umfangs der Änderungen an der FC, der ASIL-Einstufung der FC und der Verfügbarkeit der einzelnen Testumgebungen auszuwählen.

Statische Analysen

In dieser Aktivität werden Tool-gestützte Code-Analysen durchgeführt. Für die Zukunft sind auch Analysen auf Ebene des Modells geplant. Die Testanweisungen von FCI und FCF sehen vor, dass diese Aktivität nur beim FCI vorkommt. Aus Sicht des Autors begründet sich dies nur durch die Tatsache, dass der FCI im Testprozess vor dem FCF eingeordnet ist und die erneute Durchführung keinen Mehrwert bringt.

Testfallerstellung

Die Testfallerstellung ist jene Aktivität, welche zwischen FCI und FCF am stärksten abweichend definiert ist. Wie bereits beschrieben, handelt es sich beim FCI um einen White-Box Test. Die Testfälle werden daher mit der Kenntnis der Implementierung erstellt. Dabei ist die Erreichung einer bestimmten Testabdeckung vorgesehen. Die Erstellung der Testfälle im FCF basieren auf der Spezifikation der FC. Dabei werden nicht nur funktionale, sondern auch nicht funktionale Anforderungen berücksichtigt.

Testdurchführung

Diese Aktivität beinhaltet all jene Maßnahmen, welche eine Testdurchführung auf der gewählten Testumgebung vorbereiten und die Testfalldurchführung selbst. Je nach Wahl der Testumgebung kann der Arbeitsaufwand dieser Aktivität stark abweichen. Tests auf dem Simulator sind als jene Tests mit dem diesbezüglich geringsten Aufwand anzusehen.

Testauswertung und -dokumentation

Wie der Name dieser Aktivität vermuten lässt, werden in diesem Punkt des Subprozesses die Ergebnisse der durchgeführten Tests bewertet und die Testdokumentation erstellt. Letztere beinhaltet dabei nicht nur die Testergebnisse, sondern auch die Testdurchführung.

Abbildung 5.1: Workflow der Subprozesse FCI und FCF

Bei der Wahl der Testmittel wird dem Tester weitgehend freie Hand gelassen. Für sicherheitskritische Funktionen ab ASIL B sind allerdings Tests auf der Zielhardware in einem vorhergehenden Release des Systems vorgesehen. Bei mittleren bis großen Änderungen an einer Funktion sind ebenfalls Tests auf der Zielhardware vorgesehen. In der Praxis hat dies und die mangelnde Erfahrung mit der Simulationsumgebung dazu geführt, dass fast ausschließlich Tests auf der Zielhardware durchgeführt werden.

Der bestehende Workflow ist durch sehr lange Zyklen gekennzeichnet. Die Tests auf der Zielhardware sind mit langen Vorlaufzeiten verbunden. Wird beim Testen ein Fehler gefunden, kann dieser relativ schnell ausgebessert werden, allerdings ist der erneute Test wieder mit dieser langen Vorlaufzeit verbunden. Bei Termindruck kann dies zu unerwünschten Verzögerungen führen, welche im Vorfeld nur schwer einplanbar sind.

5.2 Verbesserungspotenzial innerhalb der Subprozesse

Im Folgenden wird das Verbesserungspotenzial des Workflows aufgezeigt und kurz beschrieben. Dabei kann es auch vorkommen, dass sich einzelne Verbesserungen nicht auf eine Aktivität aus dem Workflow einschränken lassen.

Automatisierung der Testvorbereitung für Tests auf der Zielhardware

Wird die Testumgebung so gewählt, dass Tests auf der Zielhardware durchgeführt werden, muss vor der eigentlichen Testdurchführung zunächst der C-Code der Unit in den Testrahmen, welcher größtenteils aus einem vorhergehenden Release der Software besteht, integriert werden. Für den Build der Test-Software existieren Skripts, welche den Entwickler unterstützen. Der Build dauert, abhängig von der Hardwarekonfiguration des Rechners und vom Umfang des Projektes, 30 min bis 2 h. Das Flashen der Software auf das Steuergerät muss größtenteils manuell durchgeführt werden und erfordert mehrere projektabhängige Einstellungen. Zusätzlich muss der Testrahmen vor dem Flashen an die jeweilige Unit angepasst werden. Dieser Arbeitsschritt ist wiederum mit einer manuellen Erstellung einer Konfigurationsdatei verbunden. Eine automatische Generierung der Konfigurationsdatei würde den Tester entlasten und den Workflow beschleunigen. Bei entsprechendem Reifegrad der Automatisierung könnten zusätzlich Konfigurationsfehler vermieden werden.

Verstärkter Einsatz der Simulation

ASCET bietet die Möglichkeit das Verhalten der Modelle zu simulieren. Die Simulation hat den Vorteil, dass eine erste Verifikation der Funktionalität bereits am Hostrechner möglich ist. Diese

Funktion wird durch die Tester bisher aber nur sehr eingeschränkt verwendet, weil die Simulation zusätzliche Einstellungen in ASCET benötigt und die entsprechende Erfahrung damit fehlt. Die Simulation könnte eine erste Instanz der Verifikation, vor dem Zielhardwaretest, darstellen. Der Vorteil diesbezüglich wäre die geringere Vorlaufzeit.

Einführung von Regressionstests

Die betrachteten Subprozesse sehen nur in Ausnahmefällen vor, die gesamte Funktionalität der Unit zu testen. Es ist die Aufgabe des Testers zu bewerten, ob eine neue Anforderung bzw. deren Implementierung Auswirkungen auf andere Anforderungen haben kann. Liegt eine Fehleinschätzung durch den Tester vor, können Querwirkungen eventuell nicht aufgedeckt werden. Regressionstests wären in der Lage einige dieser Querwirkungen zu entdecken.

Einheitliche Testfallerstellung

Im vorliegenden Workflow ist keine spezielle Methode zur Testfallerstellung vorgeschrieben. Bei FCI und FCF wird der Tester jedoch durch Checklisten bei der Testfallfindung unterstützt. Ein einheitliches Vorgehen mit entsprechender Toolunterstützung würde den Tester bei der Testfallerstellung entlasten und weitgehend einheitliche Ergebnisse liefern.

Erhöhung des Automatisierungsgrades von Testausführung und -dokumentation

Besonders im Hinblick auf Regressionstests ist ein höherer Automatisierungsgrad der Testausführung und -dokumentation anzustreben. Auf Argumente, welche für eine Automatisierung sprechen, wurde bereits in Abschnitt 4.1.1 eingegangen. TPT bietet diesbezüglich gute Möglichkeiten.

Mit der flächendeckenden Einführung von TPT sollte sich die Situation bezüglich des Automatisierungsgrades und der Akzeptanz der Simulation verbessern. TPT bietet dazu eine einheitliche Oberfläche zur Erstellung von Testfällen, sowohl für Tests am Simulator als auch der Zielhardware.

6 Analyse der Simulationsumgebung

Dieses Kapitel analysiert die Funktion zur Simulation von Modellen in ASCET und erarbeitet Gründe, warum trotz Simulation Tests auf der Zielhardware ratsam sind.

6.1 Simulation in ASCET

Da die Implementierung der Software-Units auf Modellebene mit dem Programm ASCET durchgeführt wird, ist es zunächst notwendig, die von ASCET zur Verfügung gestellten Funktionen zur Simulation und Codegenerierung genauer zu betrachten. Die Abbildung 6.1 und die folgenden Beschreibungen dienen der Veranschaulichung.

Nach der Modellierung der Unit kann diese direkt in ASCET simuliert werden. In der ASCET-Konfiguration muss dazu allerdings die Plattform von der Implementierungs- auf die Simulationsplattform geändert werden. Zu jeder Zielhardware existiert in ASCET jeweils eine Implementierungs- und eine Simulationsplattform. Die Simulationsplattform ist speziell für die Simulation konfiguriert und erlaubt keine Generierung des Implementierungscodes. Umgekehrt kann die Implementierungsplattform nicht für die Simulation genutzt werden. Um den generierten Simulationscode auf dem PC ablaufen zu lassen wird dieser durch einen plattformspezifischen Compiler übersetzt. Da unterschiedliche Plattformen für die Simulation und Implementierung verwendet werden, kommt je nachdem der Host- oder der Targetcompiler zum Einsatz.

Um die Simulation auf dem Host zu ermöglichen, muss der aus dem Modell generierte Code um Funktionen erweitert werden, welche die Simulation ermöglichen. Betrachtet man jeweils nur die Elemente, welche die gewünschte Funktion implementieren, fällt als aller Erstes auf, dass in ASCET als Systemkonstanten definierte Größen unterschiedlich behandelt werden. In der Implementierung werden diese durch Präprozessor Anweisungen geklammert, wohingegen der Simulationscode C-konforme if-Statements nutzt. Damit ergeben sich unter anderem Unterschiede bei der Auswertung der Testabdeckung, wenn Tests nur bei einer bestimmten Kombination von Werten der Systemkonstanten ausgeführt werden.

Einen Teil des Testrahmens für die Simulation bildet die Automotive Library. Dabei handelt es sich um eine Library bestehend aus vielen Grundfunktionen, welche im PVER integriert sind und für die Simulation benötigt werden. Da die Architektur der Anwendersoftware so aufgebaut ist, dass keine Funktion der Anwendersoftware eine andere aufruft, sind für die Simulation meist keine zusätzlichen Stubs notwendig. Die notwendigen Treiber werden von ASCET generiert.

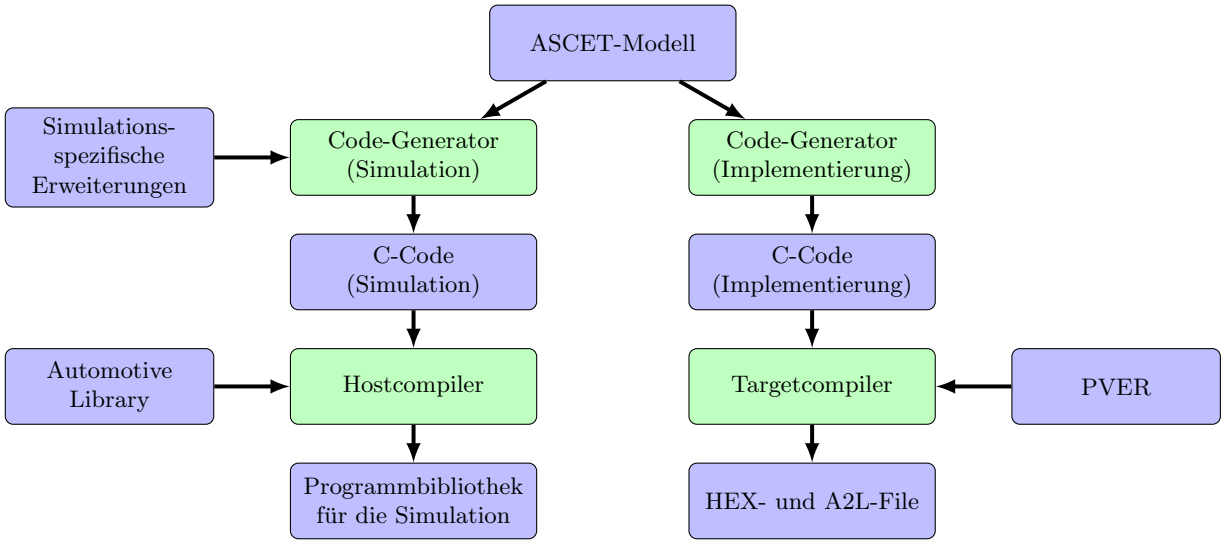

Abbildung 6.1: Diese Abbildung veranschaulicht wie aus dem ASCET-Modell die Implementierung und die Simulationsbibliothek abgeleitet werden. Bei der Simulationsbibliothek handelt es sich um eine Hostbibliothek, welche das Verhalten der Unit nachstellt.

6.2 Gründe für das Testen auf der Zielhardware trotz Simulation

In diesem Unterkapitel werden Argumente angeführt, warum Tests auf der Zielhardware anzuraten sind, auch wenn die Simulation positive Ergebnisse liefert.

6.2.1 Allgemeine Gründe

Testen auf der Zielhardware bietet mehrere Vorteile gegenüber der Simulation auf dem Hostsystem. Grünfelder [Grü13, Kap. 6.8] hat fünf Vorteile eines Target-Tests, also dem Test auf der Zielhardware, in seinem Buch zusammengefasst:

- Compilerfehler können nur bei Tests auf der Zielhardware aufgefunden werden. Voraussetzung dafür ist, dass dieselben Compilereinstellungen wie für die Auslieferung des Systems verwendet werden.
- Durch ungenügende Definitionen wird dem Hersteller von Compilern bei der Übersetzung von C-Code in die jeweilige Maschinensprache ein Interpretationsspielraum gegeben. Zusätzlich können Zahlendarstellungen und Datenbreiten bei Host- und Zielsystem unterschiedlich sein.
- Assembler Code ist nur mit Hilfe einer Emulation des Prozessors der Zielhardware am Host testbar. Ein fehlerhafter Emulator könnte dabei zu abweichendem Verhalten führen.
- Am Zielsystem müssen Betriebssystemroutinen eventuell nicht durch Stubs ersetzt werden.
- Die Norm ISO26262 rät zu Tests auf der Zielhardware.

Ein völliger Verzicht auf Target-Tests ist aus dieser Sicht nicht anzuraten. Dennoch macht es Sinn, Tests zuerst auf dem Host ablaufen zu lassen, da die Ressourcen der Zielhardware begrenzt sind und damit die Testsoftware in ihrem Umfang eingeschränkt werden könnte.

6.2.2 Für diese Umgebung spezifische Gründe

Wie daraus und aus der Abbildung 6.1 hervorgeht, ist die Ausgangsbasis für die Simulation und die Implementierung gleich. Aber bereits auf der Ebene des C-Codes ergeben sich Unterschiede.

Ein weiteres Problem bezüglich der Simulation stellen die zwischen Host und Target stark unterschiedlichen Befehlssätze dar. Dieses Problem könnte nur mit der Emulierung der Zielhardware und der Verwendung des Targetcompilers umgangen werden.

Legt man den in Abschnitt 2.3.2 vorgestellten Workflow zu Grunde und bezieht dabei die Erkenntnisse aus 6.1 mit ein ergibt sich der Workflow nach Abbildung 6.2. Auffällig dabei ist, dass das Modell nur indirekt über das Testen der Simulationsbibliothek verifiziert werden kann. Der Review findet allerdings direkt auf dem Modell statt.

Wurde die Simulation erfolgreich durchgeführt, gibt es zwei Möglichkeiten das Verhalten der Software auf der Zielhardware zu verifizieren. Zum einen ist ein Back-to-Back Test zwischen dem Verhalten der Simulation und der Zielhardware möglich. Die zweite Möglichkeit stellt die Verifikation direkt gegenüber den Anforderungen dar.

Ein weiterer Grund für die Ausführung von Tests auf der Zielhardware ist die für die Simulation verwendete Automotive-Library. Diese Library kann leider nicht alle Funktionen der Basissoftware im notwendigen Ausmaß nachbilden. So kann zum Beispiel kein EPROM-Zugriff simuliert werden. Einige Tests sind somit nur auf der Zielhardware möglich.

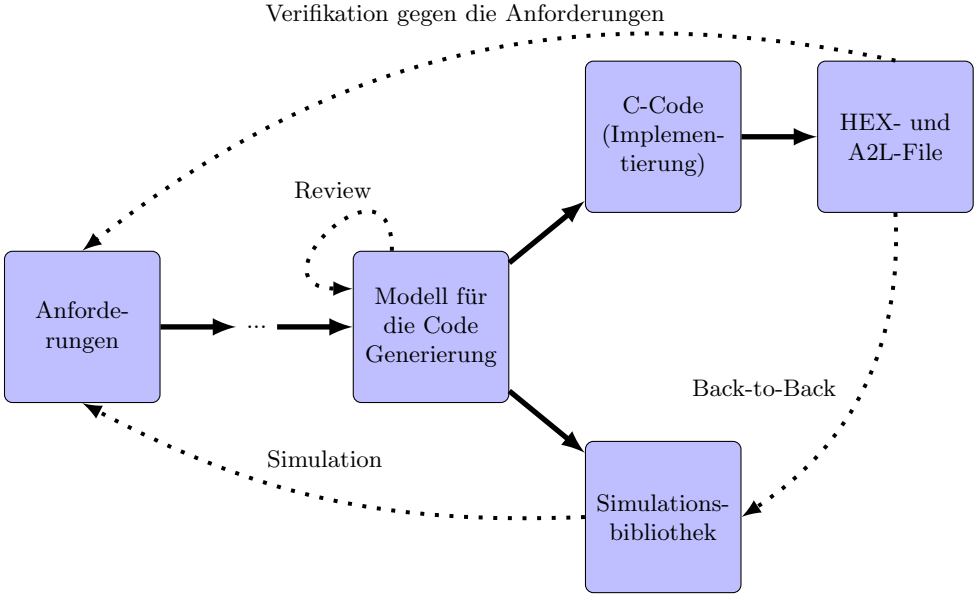

Abbildung 6.2: Workflow nach [Kur13] angepasst an die in Verwendung befindliche Entwicklungsumgebung ASCET. Vergleiche dazu Abbildung 2.4.

7 Erweiterung der bestehenden Subprozesse

In diesem Kapitel wird auf ein im Rahmen der Arbeit entwickeltes Konzept eingegangen, welches den Workflow der Subprozesse FCI und FCF erweitert.

7.1 Ziele der Erweiterung

Der Hauptfokus des Konzeptes war die Beschleunigung der Subprozesse FCI und FCF. Der stetig steigende Kosten- und Termindruck hat die Fokussierung maßgeblich beeinflusst. Zudem soll die Dauer der Subprozesse besser abschätzbar und damit planbarer werden.

Im Workflow sollen die Schleifen, welche beim Auffinden eines Fehlers durchlaufen werden müssen, so kurz wie möglich gehalten werden.

Langfristig wird eine effizientere Nutzung der zur Verfügung stehenden Zielhardware angestrebt.

Des Weiteren soll der Workflow durch Tools unterstützt werden.

7.2 Festlegung des neuen Workflows

Dieses Unterkapitel beschreibt den im Zuge der Arbeit entwickelten Workflow, welcher für den FCI und FCF gleichermaßen Anwendung finden kann. Zusätzlich zu den folgenden Beschreibungen bietet Abbildung 7.1 eine Veranschaulichung des Workflows.

Der Workflow basiert auf dem verstärkten Einsatz der Simulation von ASCET-Modellen und erlaubt damit eine erste schnelle Überprüfung durch Testen auf dem Simulator. So können Fehler schon vor den Tests auf der Zielhardware, welche mit einer hohen Vorbereitungszeit verbunden sind, gefunden werden.

Im Folgenden werden die einzelnen Aktivitäten kurz erläutert und es wird beschrieben an welchen Stellen der Workflow durch vom Autor erstellte Tools unterstützt wird. Dabei wird davon ausgegangen, dass der Tester die zur Verfügung gestellte Möglichkeit zur frühen Fehlerfindung nutzt.

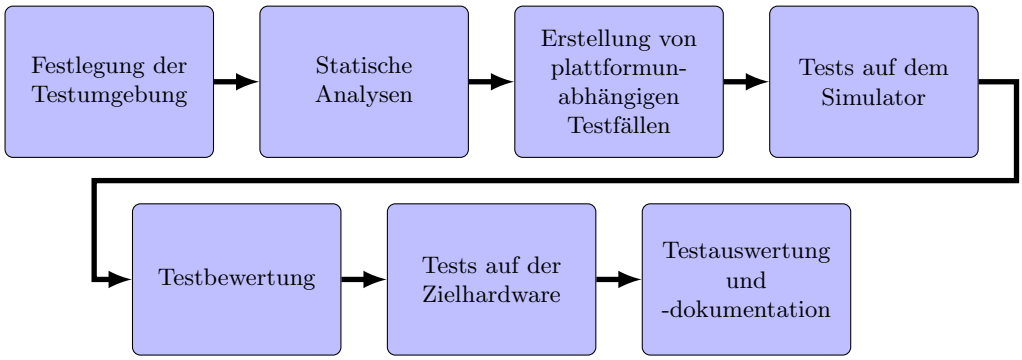

Abbildung 7.1: Erweiterter Workflow

Festlegung der Testumgebung

In dieser Aktivität wird festgelegt welche Testumgebung zum Einsatz kommt. Bevorzugt sollte der Workflow ausgeschöpft werden, indem die Kombination aus Simulation und Open-Loop Testplatz gewählt wird. In einigen Fällen kann es aber dazu kommen, dass eine Simulation der FC nicht möglich ist. Dies ist zum Beispiel der Fall wenn die Automotive Library die durch die FC aufgerufene Grundfunktion für die Simulation nicht bereitstellt.

Statische Analyse

Im bestehenden Workflow hat sich gezeigt, dass einige Fehler im Zuge des Build-Vorganges entdeckt werden. Bei diesen Fehlern handelt es sich häufig um Fehler bezüglich der Definition von Signalen der Schnittstelle der Unit. Dabei ist das Signal in dem PVER, in dem die Funktion integriert werden soll anders definiert als in dem ASCET-Modell der Funktion. Da der Build-Vorgang der Aktivität Testen auf der Zielhardware zugeordnet ist und diese erst spät im erweiterten Workflow angesiedelt ist, könnten Fehler in der Spezifikation der Unit-Schnittstellen erst spät entdeckt werden. Um dies zu vermeiden, ist im vorgeschlagenen Workflow eine erweiterte statische Analyse vorgesehen. Bei der statischen Analyse sollte das Modell zusätzlich darauf überprüft werden, ob die Definitionen der Signale der Schnittstelle im ASCET-Modell und PVER gleich sind.

Erstellung der plattformunabhängigen Testfälle

In dieser Aktivität werden die Testfälle entsprechend der Testziele des Subprozesses spezifiziert und mit Hilfe von TPT plattformunabhängig modelliert. Der Workflow sieht diesbezüglich vor, dass der Tester ein TPT-Template als Ausgangsbasis für die Modellierung nutzt. Die Testfälle müssen dabei so konzipiert sein, dass die zu stimulierenden Signale sowohl am Simulator als auch auf der Zielhardware zugänglich sind.

Tests auf dem Simulator

Die Tests werden auf dem Simulator zur Ausführung gebracht und die entsprechenden Ergebnisse aufgezeichnet.

Testbewertung

Die Ergebnisse der Tests auf dem Simulator werden darauf beurteilt, ob eine Änderung am Modell vorgenommen werden muss. Liegt ein Fehler vor, muss das Modell überarbeitet werden und der Workflow beginnt von neuem.

Tests auf der Zielhardware

Tests auf der Zielhardware sind mit einer langen Vorbereitungszeit verbunden und sollten daher im Idealfall nur einmal durchgeführt werden. Unter anderem dient die Simulation im Vorfeld dazu, dies so oft wie möglich zu erreichen. Um die Vorbereitungszeit zu verkürzen und Fehler bei der Konfigurierung des Testrahmens weitgehend auszuschließen, wird der Tester durch das Freischnitt-Tool[1] unterstützt.

Testauswertung und -dokumentation

Die Testauswertung und -dokumentation kann durch die Verwendung von TPT sehr gut automatisiert werden. Um die Erzielung einer einheitlichen Erscheinungsform der Dokumentation zu ermöglichen sieht der Workflow vor, dass im TPT-Template die diesbezüglichen Einstellungen bereits getroffen sind.

Betrachtet man die beiden Workflows in Abbildung 5.1 und Abbildung 7.1 könnte man zunächst annehmen, dass der erweiterte Workflow mehr Zeit in Anspruch nimmt. Es ist aber zu erwarten, dass durch die frühe Auffindung der Fehler weniger Tests auf der Zielhardware notwendig sind und dadurch der erweiterte Workflow einen Zeitvorteil besitzt.

Zudem begünstigt die konsequente Nutzung von TPT zur Modellierung der Testfälle die Nachvollziehbarkeit. Der Tester ist in TPT diesbezüglich gezwungen den Testfall sehr viel genauer zu spezifizieren als es bei der im derzeitigen Workflow oft gewählten manuellen Testdurchführung notwendig ist.

7.3 Anwendbarkeitsanalyse des festgelegten Freischnitts

Dieses Kapitel befasst sich mit der Festlegung eines Testrahmens für Unit-Tests auf der Zielhardware. Nach der Festlegung einer Konvention für den Freischnitt, wird diese auf ihre Anwendbarkeit bei bestehenden Kundenprojekten überprüft.

7.3.1 Festlegung des Freischnitts

Da die Tests möglichst unabhängig von der Plattform verfasst werden sollen, müssen auf der Zielhardware dieselben Signale stimuliert werden wie bei der Simulation. Da in ASCET die Unit ohne dessen Umgebung modelliert wird, ist in ASCET die Simulation nur über die Schnittstellen der Unit möglich. Für Tests auf der Zielhardware müssen die Signale dieser Schnittstelle mit einem Bypass versehen werden. Dazu ist es notwendig, den Testrahmen für die Tests auf der Zielhardware mit eHooks anzupassen.

[1] Für eine detailliertere Beschreibung des Freischnitt-Tools siehe Unterkapitel 8.1

Diese Festlegung bietet gegenüber anderen den Vorteil, völlig unabhängig von anderen Units zu sein. Eine Änderung in einer anderen Unit hat somit keinen Einfluss auf die Durchführung eines Tests.

7.3.2 Anwendbarkeit des Freischnitts für Tests auf der Zielhardware

eHooks erlaubt für einige Signale kein Setzen eines Bypass. Darüber hinaus hat sich bei verschiedensten Testläufen herausgestellt, dass ein Bypass bei einigen Signalen bzw. Signalgruppen zu Problemen führen kann. Diese Signale sollten nicht mit einem Bypass versehen werden. Einige dieser Signale lassen sich aber physikalisch direkt stimulieren. Diese Stimulation erfolgt dann anders als die Stimulation mittels Bypass nicht über INCA, sondern über das LABCAR.

Um zu überprüfen, wie viele der verwendeten Funktionen mit diesem Freischnitt auf der Hardware testbar sind, wurde durch den Autor ein kleines Programm entwickelt. Dieses liest Daten aus dem A2L-File aus und bewertet jede einzelne Funktion (FC) anhand der Schnittstellen auf die Möglichkeit, diese mit dem festgelegten Freischnitt zu testen.

Zur Analyse, deren Ergebnis in Tabelle 7.1 einsehbar ist, wurden vier verschiedene auf dem Steuergerät ablauffähige Programme (PVERs) herangezogen. Die PVERs unterscheiden sich bezüglich der enthaltenen Funktionen und der Systemkonstanteneinstellung. Beide Merkmale sind vom konkreten Kundenprojekt abhängig. Eine der gröbsten Unterscheidungsmerkmale ist dabei, ob die Steuerung eines Diesel-, Benzin- oder Hybrid-Motors durchgeführt werden soll. Die Spalten der Tabelle geben an, welche Signale über das LABCAR stimuliert werden. Die erste Spalte steht dabei für eine reine Stimulation der Unit über INCA. Bei T15_st handelt es sich um das Signal des Zündschlosses. Epm_nEng kann über das LABCAR stimuliert werden und repräsentiert die Motordrehzahl. Epm_nEng10ms repräsentiert ebenfalls die Motordrehzahl, allerdings in leicht veränderter Form. Dazu wird das Signal Epm_nEng durch die Software im 10 ms Takt abgetastet.

	INCA	T15_st	T15_st, Epm_nEng	T15_st, Epm_nEng, Epm_nEng10ms
PVER 1	76,13	77,72	87,47	88,10
PVER 2	77,43	78,95	88,52	89,15
PVER 3	62,90	67,35	86,93	87,13
PVER 4	55,52	61,52	84,57	85,05

Tabelle 7.1: Bewertung des gewählten Freischnitts. Die Zahlen geben an, wieviel Prozent der Funktionen eines Systems mit der zusätzlichen LABCAR Stimulation getestet werden können.

Aus der Tabelle lässt sich erkennen, dass bereits bei zwei zusätzlich über das LABCAR stimulierten Signalen ein sehr hoher Anteil an Funktionen mit einem Freischnitt der Schnittstellen getestet werden können. Für den Großteil der Anwendersoftware ist daher zu erwarten, dass ein für die Simulation verwendeter Test ohne größere Probleme auch auf die Zielhardware portiert werden kann.

8 Umsetzung der Tools

Dieses Kapitel gibt einen Einblick in die Umsetzung der beiden Tools, welche den definierten Workflow unterstützen sollen.

8.1 Umsetzung des Freischnitt-Tools

Da als übergeordnetes Ziel die Beschleunigung des Workflows definiert wurde, wurde im Rahmen der Arbeit ein Tool zur automatischen Generierung des Freischnitts entwickelt. Dieses Unterkapitel geht auf die Implementierung dieses Tools ein.

8.1.1 Use Cases

Vor Beginn der Entwicklung des Tools, wurden Use Cases festgelegt. Vorbedingung für beide Use Cases ist das Vorhandensein von A2L- und HEX-File eines ablauffähigen Systems. Dieses System sollte die zu testende Unit aber noch keine Bypässe enthalten.

Freischnitt anhand des Funktionsnamens

Nach Aufruf des Tools durch den Benutzer und des entsprechenden Menüpunktes wird dieser durch das Tool aufgefordert den Namen der freizuschneidenden Funktion (Unit) einzugeben. Das Tool führt die Vorbereitungen und den Freischnitt selbst durch. Alternativ besteht die Möglichkeit, die in der automatisch generierten Konfigurationsdatei getroffenen Einstellungen in der eHooks GUI manuell zu überprüfen bevor der Freischnitt durchgeführt wird.

Freischnitt anhand eines TPT-Files

Dem Freischnitt-Tool wird der Pfad zu einem TPT-File übergeben. Das Tool liest aus dem Signal-Mapping aus, welche Signale für den Test mit einem Bypass versehen werden müssen. Gleichzeitig erfolgt eine Überprüfung, ob die Signale im vorliegenden System zur Verfügung stehen. Nach der Vorbereitung des HEX-Files für den Freischnitt besteht auch hier die Möglichkeit die in der Konfigurationsdatei gespeicherten Einstellungen vor der Durchführung der Manipulationen am HEX-File zu überprüfen.

Beide Use Cases nehmen dem Tester die Arbeit der manuellen Erstellung der Konfigurationsdatei ab. Die dazu benötigten Daten werden entweder aus dem TPT- oder dem A2L-File ausgelesen. Dies verringert die Gefahr von Fehlern bei den vorgenommenen Einstellungen, belastet Tester nicht mit monotonen Arbeiten und beschleunigt den Vorgang.

8.1.2 Integration in den Workflow

Das Tool kommt nach der Kompilierung des ablauffähigen Systems für die Zielplattform zum Einsatz. Wie in Abschnitt 3.2.5 beschrieben, ist es mit eHooks möglich, einen Freischnitt durch direkte Manipulation der HEX-Datei einzufügen. eHooks bietet dazu eine eigene grafische Oberfläche an. In diesem Programm kann der Freischnitt festgelegt und durchgeführt werden. Bei Bedarf können die Konfigurationen im XML-Format gespeichert werden. Ein späterer Kommandozeilenaufruf des Programms mit der Konfigurationsdatei löst die Ausführung des Freischnitts aus. Das Freischnitt-Tool setzt bei der Generierung der Konfigurationsdatei an und umgeht dabei die aufwendige händische Festlegung.

Abbildung 8.1 zeigt schematisch, wie das Tool in die bestehende Toollandschaft integriert wird, auf welche Dateien es zugreift und welche Dateien generiert werden. Bevor ein Bypass in das System eingefügt werden kann, muss der Output des Compilers mit eHooks auf die Manipulation vorbereitet werden. Dazu wird eine projektabhängige Konfigurationsdatei benötigt. Diese Datei wird im Folgenden auch zur Erstellung der eHooks-Dev-Konfigurationsdatei benötigt. Unter anderem wird das darin definierte Passwort für das Einfügen eines Bypasses benötigt. Das Freischnitt-Tool benötigt somit Zugriff auf die darin enthaltenen Daten. Zusätzlich muss je nach Use Case nur auf das A2L-File oder zusätzlich auch auf das TPT-File zugegriffen werden. Die generierte Dev-Konfigurationsdatei dient anschließend als Input für die eigentliche Durchführung des Freischnitts durch das Programm eHooks-Dev. Das Freischnitt-Tool koordiniert dabei die einzelnen Abläufe und dient somit gewissermaßen als Steuerzentrale. Es versorgt dazu die anderen Programme mit den Pfaden zu den benötigten Dateien und triggert deren Ausführung.

8.1.3 Klassendiagramm

Bei dem Freischnitt-Tool handelt es sich um ein Skript, welches mit Übergabeparametern aufgerufen werden kann. Ein Übergabeparameter legt fest, ob der Freischnitt durch ein TPT-File oder einen Funktionsnamen erfolgt. Ein weiterer Übergabeparameter versorgt das Skript mit dem Pfad zum TPT-File bzw. mit dem Namen der Funktion. Darüber hinaus müssen noch weitere Übergabeparameter übergeben werden, auf diese wird jedoch nicht genauer eingegangen, da sie in erster Linie dazu dienen das Skript mit den nötigen Pfaden zu den Testordnern und Konfigurationsdateien zu versorgen.

Die Klasse TPTParser im Kassendiagramm aus Abbildung 8.2 dient dazu Daten aus dem TPT-File auszulesen und aufzubereiten. Der TPTParser erlaubt es unter anderem alle definierten Signale auszulesen.

Die Klasse EHOOKSParser ist für die Erstellung der eHooks-Konfigurationsdatei notwendig. Die Funktion addHooks(signal_name:String) ermöglicht das Hinzufügen von Bypässen. Zunächst werden diese noch nicht in die Konfiguration eingefügt, sondern intern separat verwaltet. Bevor die Konfiguration mit der Funktion write() in die Konfigurationsdatei geschrieben wird, muss die Funktion refreshTree() aufgerufen werden, um die Bypässe in die Konfiguration einzufügen.

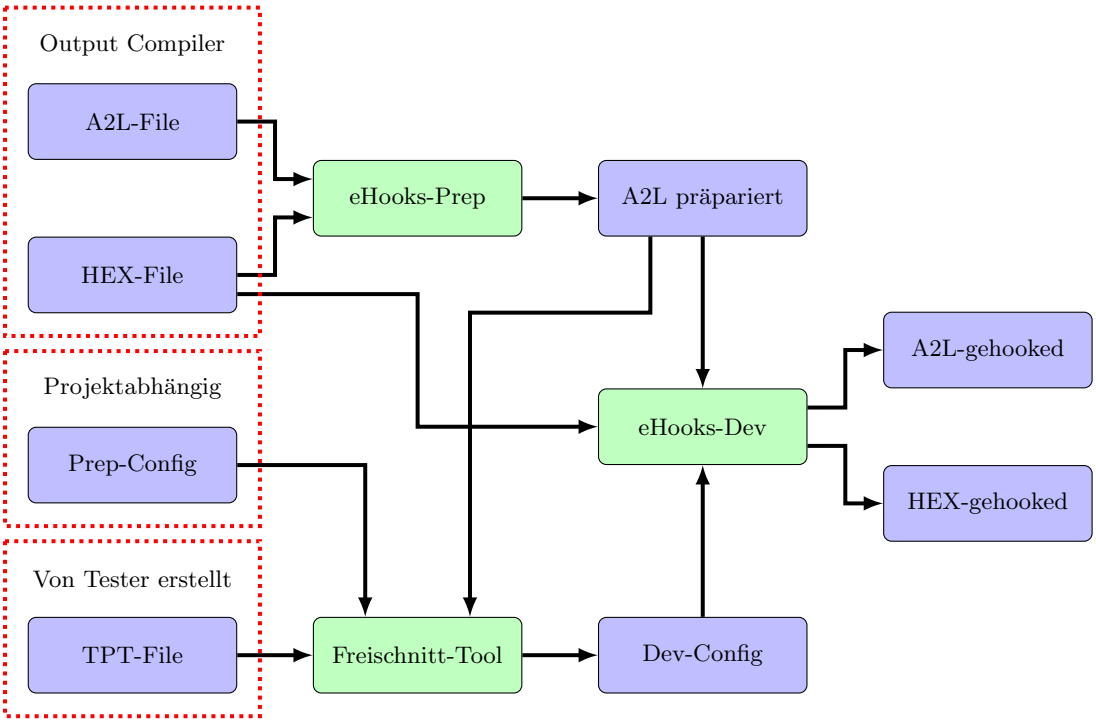

Abbildung 8.1: Schematische Darstellung des Datenflusses. blau: Dateien; grün: Programme

EasyA2L ist jene Klasse, welche einen Zugriff auf die in einem A2L-File gespeicherten Daten erlaubt. Das File wird beim Initialisierungsvorgang geparst. Die Übergabeparameter dienen dabei zur Festlegung, welche Daten aus dem File benötigt werden. Der Zugriff auf die geparsten Daten erfolgt im Anschluss direkt über die entsprechenden Klassenvariablen. Diese Klasse konnte aus einem anderen Projekt wiederverwendet werden. Der Autor dieser Diplomarbeit ergänzte die Klasse lediglich um die Möglichkeit die Header-Informationen auszulesen, da diese Daten für die Erstellung der Konfigurationsdatei notwendig sind.

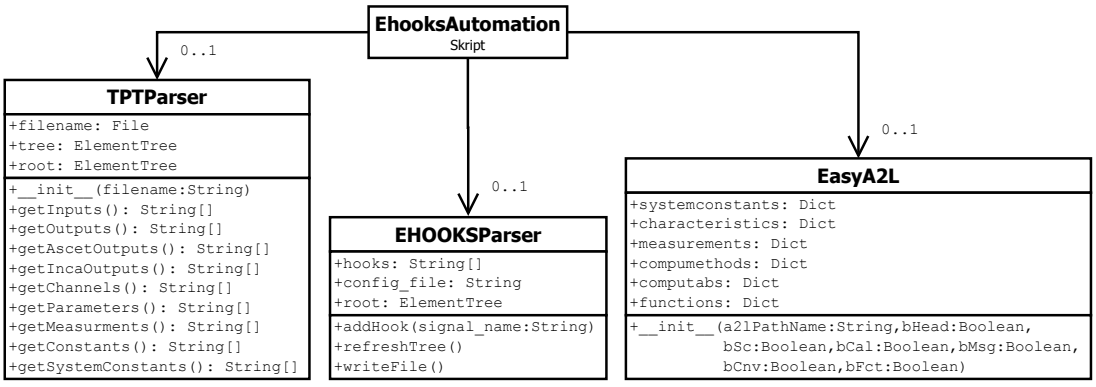

Abbildung 8.2: Klassendiagramm des Freischnitt-Tools

8.2 Möglichkeiten zur Realisierung der Plattformunabhängigkeit

Der in Unterkapitel 7.2 definierte Workflow sieht plattformunabhängige Tests vor. Dieses Unterkapitel befasst sich daher mit verschiedenen Ansätzen zur Umsetzung der Plattformunabhängigkeit von in TPT modellierten Testfällen.

Für die Umsetzung der Plattformunabhängigkeit wurde aus drei Ansätzen, welche im Folgenden kurz beschrieben werden, eine Auswahl getroffen.

8.2.1 Generierung mehrerer TPT-Files

Aus einem TPT-File mit den plattformunabhängigen Testfällen werden mit Hilfe eines externen Programms mehrere TPT-Files generiert, welche jeweils für eine spezielle Plattform ausgelegt sind. Für die Durchführung der Tests wird das entsprechende plattformspezifische TPT-File herangezogen.

Aus einem plattformunabhängigen TPT-File entstehen mehrere plattformabhängige TPT-Files, wobei die Übersetzung auf einem plattformabhängigen Template basieren könnte. Dieses Template könnte anschließend mit den für die Plattform übersetzten Tests gefüllt werden. Die eigentliche Übersetzung würde sich sehr einfach gestalten, indem Signalnamen und -definitionen aus dem plattformunabhängigen TPT-File durch dazugehörige plattformabhängige ausgetauscht und in das zu generierende TPT-File eingetragen werden.

Da die Übersetzung vor der eigentlichen Durchführung der Tests durchgeführt wird, bestehen keine besonderen zeitlichen Anforderungen an das Programm zur Übersetzung.

8.2.2 TPT-interne Lösung

TPT bietet zu vielen Plattformen geeignete Schnittstellen an. Über Plattformkonfigrationen können innerhalb eines TPT-Files mehrere Plattformen definiert werden. Zu jeder dieser Plattformen werden ein oder mehrere Mappings benötigt. Ein Mapping definiert dabei, an welche der Schnittstellen zu externen Programmen das Signal weitergeleitet wird. Dabei ist es auch möglich einen externen Namen für ein Signal zu definieren. Innerhalb des TPT-Files ist es somit möglich, beliebige Namen für ein externes Signal zu verwenden, was sich bei geeigneter Namensgebung positiv auf die Verständlichkeit des Testfalls auswirken kann. Zusätzlich kann so ein gewisser Grad an Plattformunabhängigkeit erreicht werden. Leider sind dazu viele Einstellungen an verschiedenen Stellen des Programms notwendig und eine völlige Plattformunabhängigkeit ist nicht erreichbar, wenn die Datentypen von externem und internem Signal nicht übereinstimmen. Für das angestrebte Ziel, denselben Testfall für die Simulation und den Test auf der Hardware zu verwenden, ist dies leider der Fall. Es ist jedoch möglich in jeden Testfall einen parallelen Pfad einzufügen, in dem eine Übersetzung in den jeweils anderen Typ erfolgt. Der Aufwand diese Lösung in das TPT-File zu integrieren ist allerdings sehr hoch.

Die Übersetzung wird zur Laufzeit des Tests und unabhängig davon, ob diese tatsächlich benötigt wird, durchgeführt. Da diese Lösung direkt auf dem Funktionsumfang von TPT basiert, kann angenommen werden, dass es zu keinen Laufzeitproblemen bei der Übersetzung kommen kann.

8.2.3 Externer Übersetzer

Dieser Ansatz verfolgt die Strategie, den in TPT verfassten Testfall mit einem externen Programm für die jeweilige Plattform zu übersetzen. Dazu wird in TPT lediglich eine Plattform definiert, welche die Signale an ein zusätzliches externes Programm weiterleitet. Dieses Programm dient zur Abstrahierung der für den Testlauf auf der jeweiligen Plattform notwendigen Ansteuerung. Der Übersetzer benötigt dazu plattformspezifische Datensätze, welche zuvor festgelegt werden müssen.

Die Übersetzung erfolgt dabei zur Laufzeit des Tests. Dadurch entstehen zusätzliche Anforderungen an das zeitliche Verhalten des Übersetzers.

8.2.4 Erläuterungen

Der Autor hat sich für die TPT interne Lösung entschieden, da ihm diese als die geeignetste für die Umsetzung erschien. Im Folgenden wird die Entscheidung kurz begründet.

Ziel der Abteilung in den letzten Jahren war die Units so zu gestalten, dass alle möglichen Varianten der Unit in einem Modell vereint werden können. Erst bei der Generierung des C-Codes wird durch Systemkonstanten festgelegt, welche Variante zum Einsatz kommt. Es wird also das Ziel verfolgt alle Daten an einem Ort zu bündeln. Die Generierung von mehreren TPT-Files würde diesem Ziel widersprechen und kann zusätzlich zu Konsistenzproblemen bei Änderungen führen. Die interne Lösung kann so gestaltet werden, dass nur ein TPT-File für alle Plattformen und Varianten existiert.

Die ISO26262 sieht vor, dass alle für die Testausführung verwendeten Tools entsprechend der Norm zertifiziert werden müssen. Da es sich bei einem externen Übersetzer um ein Programm handeln würde, welches direkt an der Testausführung beteiligt ist, müsste dieses ebenfalls zertifiziert werden. Diese Zertifizierung wäre mit einem zusätzlichen Zeit- und Kostenaufwand verbunden. Darüber hinaus müsste der externe Übersetzer zahlreiche Schnittstellen zu anderen Programmen implementieren. Da TPT diese Schnittstellen bereits anbietet, würde nach Meinung des Autors ein erheblicher Anteil des Entwicklungsaufwandes unnötig aufgewandt. Dieser und jener Aufwand für die Zertifizierung fallen bei einer TPT-internen Lösung weg.

Ziel ist es also den Entwickler der Tests bei der Konfigurierung der TPT-Files so zu unterstützen, dass diese fehlerfrei und schnell von statten geht. Das TPT-File wird dazu mit einem externen Tool modifiziert. Da dieselben Einstellungen auch händisch vorgenommen werden können, ist eine Zertifizierung des externen Tools nicht notwendig.

8.3 Umsetzung des Setup-Tools

Da es sich bei dem Übersetzer um eine TPT-interne Lösung handelt, wird ein Tool benötigt, welches die notwendigen Einstellungen in TPT vornimmt. Da ein Teil der Testumgebung und damit das TPT-File unabhängig von der zu testenden Unit sind, können Teile der Konfigurierung über einer Vorlage (Template) abgedeckt werden. Das externe Tool muss anschließend nur noch Unit-spezifische Einstellungen vornehmen. Da es sich dabei gewissermaßen um ein Setup des TPT-Files für die eigentliche Eingabe der Testfälle handelt, wird dieses Tool im Folgenden in Unterkapitel Setuper genannt.

8.3.1 Use Cases

Um die Anforderungen an den Setuper genauer zu spezifizieren, werden zunächst Use Cases formuliert.

Erstellung eines leeren Templates

Ein Tester möchte für eine Unit ein neues TPT-File generieren. Dazu wird der Setuper gestartet und der entsprechende Menüpunkt ausgewählt. Das System fragt nach dem Ort, an dem das File angelegt werden soll. Nach Angabe des Zielordners wird durch den Setuper die notwendige Ordnerstruktur und das TPT-File erzeugt.

Initial Setup

Ein zuvor leeres Template wird zunächst mit der Schnittstellendefinition befüllt. Die von der Unit abhängigen Einstellungen sollen durch das Tool soweit ergänzt werden, dass ein für den Open-Loop Testplatz erstellter Testfall ohne größeren Aufwand auch für die Simulation verwendet werden kann.

Änderungen der Schnittstelle

Existiert bereits ein TPT-File und kommt es im Laufe der Weiterentwicklung der Unit zu einer Änderung der Schnittstelle, muss das TPT-File modifiziert werden. Dazu muss dem Setuper das TPT-File bekannt sein. Durch den Aufruf der entsprechenden Funktion des Setupers wird die alte Schnittstellendefinition gelöscht und der User aufgefordert die neue einzugeben bzw. zu importieren. Nach dem Speichern und Schließen des TPT-Files werden die Einstellungen im TPT-File an die geänderte Schnittstellendefinition angepasst.

Auslesen von Systemkonstanten

Bei der Konfigurierung der Plattformen in TPT ist es in vielen Fällen notwendig, eine Konfigurierung der Systemkonstanten vorzunehmen. Es werden zwar die Standardwerte aus dem ASCET-Modell ausgelesen, allerdings können diese von jenen Werten im Projekt abweichen. Das Tool soll daher eine Möglichkeit bieten, die Werte der Systemkonstanten aus einem A2L-File auszulesen. Beim Aufruf dieser Funktion wird zur Eingabe des Pfades zum zugrunde liegenden A2L-File aufgefordert. Anschließend werden die Werte der benötigten Konstanten ausgelesen und das TPT-File zur Eingabe durch den User geöffnet.

Diese Use Cases dienen als Grundlage für eine weiterführende Spezifikation der Software.

8.3.2 Integration in den Arbeitsablauf

Da es sich um Unittests handelt, wird davon ausgegangen, dass ein ASCET-Modell der Unit oder zumindest die Spezifikation der Unit-Schnittstellen vorhanden ist. Ausgehend davon soll das Aufsetzen der Testumgebung ohne großen zusätzlichen Aufwand für den Ersteller möglich sein. Um dieses Ziel zu erreichen wurde ein Arbeitsablauf definiert, welcher in Abbildung 8.3 ersichtlich ist.

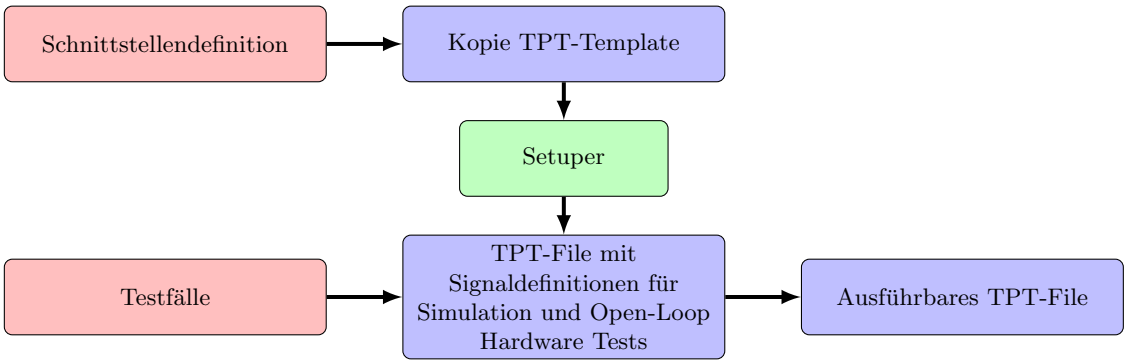

Abbildung 8.3: Arbeitsablauf zur Erstellung eines plattformunabhängigen TPT-Files. Durch die Integrierung des Setupers in den Arbeitsablauf erhöht sich der Arbeitsaufwand im Vergleich zur Erstellung eines Testfalles für den Open-Loop Testplatz nur minimal.

Die erweiterten Subprozesse (siehe Kapitel 7) wären mit erhöhtem Aufwand für die Modellierung verbunden, da der Testfall jeweils für die Simulation und für Tests auf der Zielhardware modelliert werden müsste. Dies galt es zu vermeiden. Mit Unterstützung des Setupers verfasste Testfälle besitzen diesbezüglich den Vorteil, dass nur ein Testfall modelliert werden muss. Das Einbinden der Simulation in den Workflow ist damit nur mit sehr geringem zusätzlichen Aufwand verbunden.

Da das Programm TPT keine API anbietet, muss dazu das TPT-File direkt modifiziert werden. Glücklicherweise erfolgt die Speicherung in dem für Menschen grundsätzlich lesbaren XML-Format. Im Zuge der Implementierung war es unter anderem notwendig, die Struktur des TPT-Files zu analysieren.

Bevor mit der Implementierung begonnen wurde, wurde das Sequence-Diagramm (Abbildung 8.4) des Use Cases Initial Setup angefertigt und diente im weiteren Entwicklungsprozess als Anhaltspunkt.

Nach dem Start des Setupers und der Auswahl des entsprechenden Menüpunktes wird der User aufgefordert, einen Pfad zu einem Ordner anzugeben. In diesem Ordner wird ein TPT-File mit dem ebenfalls vom User spezifizierten Filenamen und eine Ordnerstruktur angelegt. Nach diesem Vorgang wird das TPT-File geöffnet und um die Eingabe der Schnittstellendefinition gebeten. Dieser Schritt kann durch manuelle Eingabe oder die in TPT integrierte Funktion zum Import der Signaldefinitionen aus ASCET-Modellen erfolgen. Das Use Case Diagramm bildet allerdings nur die zweite Möglichkeit ab. Nach dem Schließen des TPT-Files gibt der Setuper sein Menü wieder frei. Der User kann nun mit der Auswahl des Menüpunktes Initial Setup die Modifizierung des TPT-Files veranlassen. Nach der Modifizierung ist der Setuper nicht mehr notwendig, es sei denn, es gibt Änderungen an der Definition der Schnittstelle bzw. es sollen die Werte der Systemkonstanten ausgelesen werden.

Darauf folgend können die Testfälle wie gewohnt erstellt werden. Diese Testfälle sind anschließend sowohl für die Simulation als auch für den Test auf der Zielhardware verwendbar.

8.3.3 Klassendiagramme

Bei der für die Umsetzung gewählten Programmiersprache Python handelt es sich um eine objektorientierte Sprache. Um die Übersichtlichkeit und die Wiederverwendbarkeit von Softwareteilen

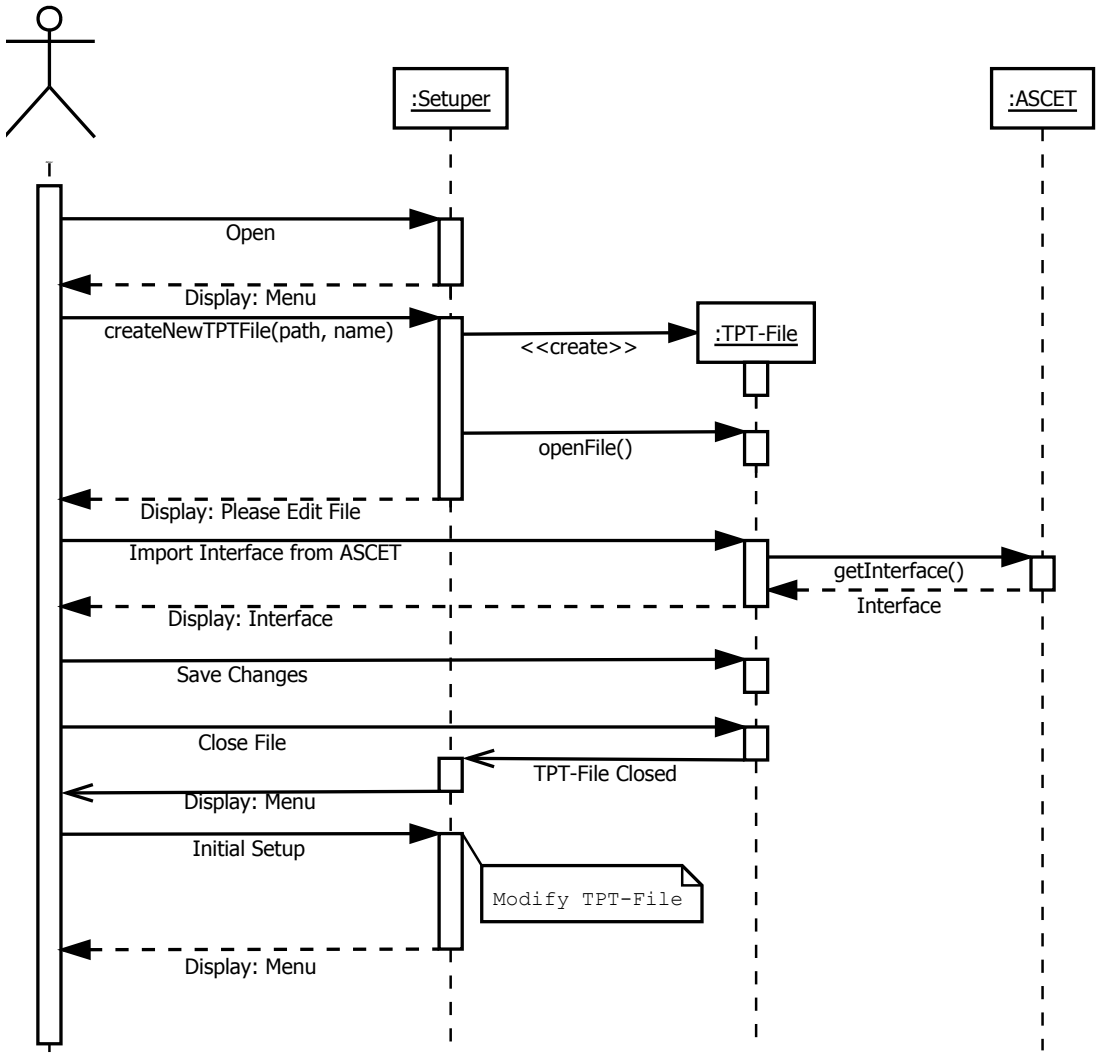

Abbildung 8.4: Sequence-Diagramm eines initialen Setup des TPT-Files.

sicherzustellen, wurden die Funktionen in mehreren Klassen gekapselt. Abbildung 8.5 enthält dazu das Klassendiagramm.

Der TPTSetuper koordiniert das Zusammenspiel der einzelnen Klassen und stellt ein User Interface zur Verfügung.

Die Klasse TPTParser wurde gegenüber 8.1.3 um Funktionen zur Modifizierung und zum Schreiben von TPT-Files erweitert. Die Modifizierungen werden im Zuge der Unterstützung des Testers bei der Erstellung der plattformunabhängigen Tests notwendig.

Die Klasse EasyA2L ermöglicht den Zugriff auf Daten im A2L-File und wurde bereits in Abschnitt 8.1.3 etwas genauer beschrieben.

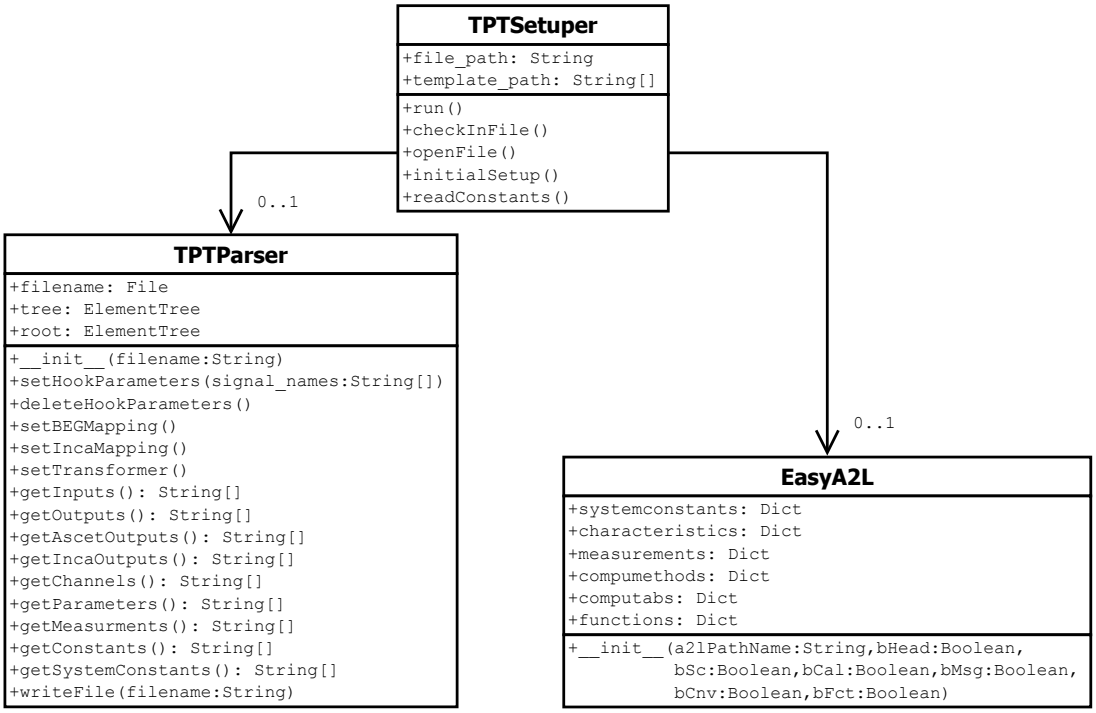

Abbildung 8.5: Klassendiagramm des TPTSetuper

9 Zusammenfassung und Ausblick

Dieses Kapitel gibt nach einer Zusammenfassung einen Ausblick auf zukünftig mögliche Entwicklungen im Bereich des Testprozesses.

9.1 Zusammenfassung

Diese Diplomarbeit hat sich mit der Verbesserung eines Verifikationsprozesses befasst, etwas genauer gesprochen mit den Subprozessen für Unittests des Testprozesses. In einem ersten Schritt wurden die betrachteten Subprozesse analysiert und deren Verbesserungspotenzial herausgearbeitet.

Der auf der Basis dieser Analyse neu definierte Workflow setzt auf den verstärkten Einsatz der Simulation von ASCET-Modellen und des Testautomatisierungsprogramms TPT. Dabei wurde der Workflow mit dem Ziel kürzer, besser abschätzbar und planbarer zu sein entwickelt. Da aus verschiedenen Gründen neben der Simulation zusätzliche Tests auf der Zielhardware notwendig sind, wurde im Zuge der Arbeit eine Methode zur Definition von plattformunabhängigen Tests erarbeitet. Zudem wurde eine Konvention bezüglich des Testrahmens getroffen um eine Grundlage für Regressionstests zu legen.

Um den Automatisierungsgrad des neu definierten Workflows zu erhöhen und den Tester von monotonen Tätigkeiten zu entlasten, wurden zwei Tools entwickelt. Nach der Erstellung der Konzepte wurden diese in prototypischen Implementierungen umgesetzt.

9.2 Ausblick

Der definierte Workflow der Subprozesse kann als Grundlage für die Einführung von Regressionstests genutzt werden. Da der Testrahmen für jeden Test einer Unit gleich definiert ist, kann eine Testsuite aufgebaut werden, welche nach jeder Änderung an der Unit ausgeführt werden sollte.

Mit der Überprüfung der Definition der Schnittstellen, wie sie im neuen Workflow vorgesehen ist, ist anzunehmen, dass der Kompiliervorgang künftig mit weniger Fehlern verbunden sein wird. Bei einem neuen Build können so mehrere neue bzw. überarbeitete Units in den Testrahmen, welcher aus dem vorhergehenden Release des Softwaresystems besteht, ohne größere Probleme integriert werden.

Mit Hilfe des Freischnitt-Tools und einer Erweiterung der Flash-Automatik könnte die bestehende Automatisierung der Testausführung dahingehend entwickelt werden, dass ausgewählte Unittests voll-automatisiert aufeinander folgend ausgeführt werden. Im Idealfall wären dazu nur der Software Build, in dem alle zu testenden Units enthalten sind, und die TPT-Files notwendig. Der Workflow könnte diesbezüglich so angepasst werden, dass dieser bis zur Testauswertung nach der Simulation für jede Unit getrennt und danach für mehrere Units zusammen ausgeführt wird.

Da das Freischnitt-Tool den Testrahmen für Tests auf der Zielhardware anhand der Signaldefinitionen im TPT-File anpassen kann, wären zusätzlich zu den Unittests auch automatische Integrationstests auf dieser Basis möglich.

Wissenschaftliche Literatur

[AWS14] Harald Altinger, Franz Wotawa, and Markus Schurius. Testing Methods Used in the Automotive Industry: Results from a Survey. In *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*, pages 1–6, New York, NY, USA, 2014. ACM.

[BHK09] F. Belli, A. Hollmann, and M. Kleinselbeck. A Graph-Model-Based Testing Method Compared with the Classification Tree Method for Test Case Generation. In *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pages 193–200, July 2009.

[BKZK11] Manfred Broy, Helmut Krcmar, Jens Zimmermann, and Sascha Kirstan. Einfluss des Software-Designs auf die Wirtschaftlichkeit von Software-Entwicklungen. *ATZelektronik*, 6(2):34–37, 2011.

[FR07] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[Grü13] Stefan Grünfelder. *Software-Test für Embedded Systems*. dpunkt.verlag GmbH, Heidelberg, 1.auflage edition, 2013.

[Hie07] Thomas Hiebler. Modellgetriebene Entwicklung von Webanwendungen aus Anforderungsspezifikationen. 2007.

[Hil12] Martin Hillenbrand. *Funktionale Sicherheit nach ISO 26262 in der Konzeptphase der Entwicklung von Elektrik / Elektronik Architekturen von Fahrzeugen*. KIT Scientific Publishing, Karlsruhe, 2012.

[HL14] Matthias Hamburg and Anke Löwer. ISTQB./GTB Standardglossar der Testbegriffe Deutsch - Englisch Version 2.3. *http://www.german-testing-board.info/fileadmin/gtb_repository/downloads/pdf/glossar/CT_Glossar_DE_EN_V23.pdf*, 2014.

[ISO11] [ISO 26262-6] ISO 26262 Road vehicles - Functional safety - Part 6: Product development at the software level. 2011.

[IST14] Certified Tester. Expert Level Syllabus. Test Automation - Engineering (Version 2014). International Software Testing Qualifications Board, 2014.

[Kan03] Cem Kaner. An Introduction to Scenario Testing. *http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf*, 2003.

[KM14] Damodaram Kamma and Pooja Maruthi. Effective Unit-testing in Model-based Software Development. In *Proceedings of the 9th International Workshop on Automation of Software Test*, AST 2014, pages 36–42, New York, NY, USA, 2014. ACM.

[Kur13] C. Kuratas. ISO 26262 konforme Software Entwicklung mit Model-Based Design.

Schritte in die künftige Moblität, Springer Fachmedien Wiesbaden, 2013.

[Reu14] Andreas Reuter. ISO 26262 - Funktionale Sicherheit im Automobil. *InTeR 3/14*, 2014.

[Som12] Ian Sommerville. *Software Engineering.* Pearson Deutschland GmbH, München, 9., aktualisierte auflage edition, 2012.

[Vig10] Uwe Vigenschow. *Testen von Software und Embedded Systems.* dpunkt.verlag, Heidelberg, zweite auflage edition, 2010.

Internet Referenzen

[1] Apica Inc., Santa Monica, US. Aug 2015. https://www.apicasystem.com/blog/automated-testing-vs-manual-testing/.

[2] Bibliographisches Institut GmbH, Berlin, Deutschland. Aug 2015. http://www.duden.de/.

[3] ETAS GmbH, Stuttgart, Deutschland. Sep 2015. http://www.etas.com/de/products/ascet_software_products.php.

[4] ETAS GmbH, Stuttgart, Deutschland. Sep 2015. http://www.etas.com/de/products/ascet_software_products-details.php.

[5] ETAS GmbH, Stuttgart, Deutschland. Sep 2015. http://www.etas.com/de/products/inca.php.

[6] ETAS GmbH, Stuttgart, Deutschland. Sep 2015. http://www.etas.com/de/products/labcar_system_components.php.

[7] ETAS GmbH, Stuttgart, Deutschland. Sep 2015. http://www.etas.com/de/products/ehooks.php.

[8] Freie Universität Berlin, Berlin, Deutschland. *Testen. Prinzipien und Methoden*, Aug 2015. http://www.inf.fu-berlin.de/lehre/SS02/alp2/folien_natalie/testen.pdf/.

[9] PikeTec GmbH, Berlin, Deutschland. Aug 2015. http://www.piketec.com/products/tpt.php?lang=de.

[10] Robert Bosch AG, Stuttgart, Deutschland. Aug 2015. https://inside-docupedia.bosch.com/confluence/x/j1gFCg.

[11] Smart Bear, Boston, USA. Aug 2015. http://support.smartbear.com/articles/testcomplete/manager-overview/.

[12] Zühlke Technology Group AG, Schlieren (Zürich), Schweiz. Aug 2015. http://blog.zuehlke.com/testautomatisierung-fuer-embedded-software-wozu-ist-das-gut/.